Semantic Search over XML Document Streams

Zografoula Vagena
Microsoft Research Cambridge
7 JJ Thomson Avenue
Cambridge, UK
zografv@microsoft.com

Mirella M. Moro*
Instituto de Informatica
Univ. Federal do Rio Grande do Sul - UFRGS
Porto Alegre, Brazil
mirella@inf.ufrgs.br

ABSTRACT

A large number of web data sources, such as blogs, news sites and podcast hosts, are currently disseminating their content in the form of streaming XML documents. The variability and heterogeneity of those sources make the employment of traditional querying schemes, which are based on structured query languages, cumbersome for the end user (those languages require precise knowledge of the underlying schema of each queried data source in order to be able to formulate meaningful queries). On the other hand, keyword search provides an alternative retrieval paradigm that is both simple and effective. Its importance for XML retrieval is well established and many specialized XML search engines have already appeared. Those engines support semantic search over XML documents within persistent environments, in which XML documents are permanently stored and can be indexed for efficient retrieval. Nevertheless, to the best of our knowledge, there is currently no published work that focuses on a streaming environment. In this paper, we attempt to fill this gap and study the problem of semantic keyword search over streaming XML documents. In particular, we build on previous work on semantic search over stored XML documents and propose a retrieval language that is simple and enables semantic search over XML documents. We then devise novel, online query processing algorithms that can answer semantic search queries over streaming XML data.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing, textual databases*

Keywords

XML streams, keyword search

1. INTRODUCTION

XML is rapidly becoming the *de-facto* standard for data dissemination and sharing on the web. In this environment, data from multiple heterogeneous data sources are retrieved, aggregated and pushed to the end user, usually in an online, streaming fashion. The XML data that originate from those data sources contain large amounts of plain text and are usually *schemaless* and *self-describing*, while exhibiting an inherent structure, which is defined by the different tags and tag attributes that may exist within an XML document. Due to the expressive power of XML (which enables the relaxation of many relational features and allows data with optional typing, sparse attributes, hierarchical representation, flexible schema, and optional identification keys) that structure can become arbitrary complex.

Structured XML query languages, such as XQuery and XPath, permit querying both on the data structure and on data values using predicates with exact semantics. As a result, deep knowledge of the structure of the data is required in order to formulate meaningful XML queries in those languages. Moreover, the existence of XML documents with large amounts of plain text within the web environment makes the retrieval of values based on predicates with exact semantics extremely difficult. Both problems (i.e. knowledge of the structure and retrieval with exact semantics) are aggravated with the existence of many disparate data sources. In this case, a knowledge of the data structure and the exact representation of values from multiple data sources and the formulation of an appropriate query for each data source is required.

EXAMPLE 1. *To better illustrate the problem, we present an example with two different bibliography data sources. Figure 1(a) illustrates a fraction of the first data source and Figure 1(b) a fraction of the second one. Both data sources represent information about books and authors. However,* Bib_1 *organizes the data grouped by* $book$*, while* Bib_2 *groups by* $author$*. If a user wants to retrieve information on* **books** *written by* **author** *"*l_2*", she needs to issue two structurally different queries, one for each data source:*

$/Bib/book[.//author = "l_2 f_2"]$
$/Bib/author[last = "l_2"]/book$

From the example, it becomes obvious that in order to retrieve the desired information the user has to (a) know the exact structure of each data source, and (b) the exact value of the textual content under the $author$ *tag. The usage of the* $contains$ *function that has been defined in XPath might waive the second requirement; however, its functionality is currently limited to very simple full-text retrieval tasks.*

To overcome the aforementioned problem, keyword search has been employed for XML data retrieval. The main advantage of keyword search is its simplicity: users do not have to know a complex

*Mirella Moro was supported by CNPq, Brazil - 151708/2007-0.

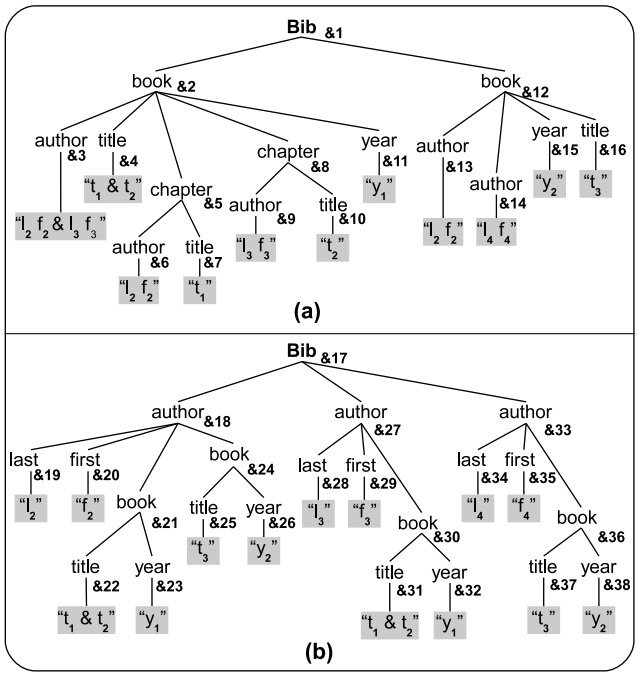

Figure 1: Tree Representation of XML Data Collections.

query language and can query any dataset without prior knowledge of the structure of the underlying data. So far, a number of specialized XML search engines have appeared (e.g. [5, 6, 12]). They leverage the power of pure, plain text-based, keyword search, in order to enable queries that convey semantic knowledge (e.g. that the user is looking for the $author$ of a particular $book$) and retrieve data at different granularities than a physical document. Nevertheless, those engines work over stored XML documents. Porting their functionality to a streaming environment is not straightforward, since the processing algorithms of those engines depend heavily on the existence of full-text indexes.

In this work, we tackle the problem of semantic XML search within streaming environments. In particular we (a) describe a retrieval language that is simple and enables semantic search over XML documents and (b) devise novel query processing algorithms that can efficiently answer semantic search queries (posed in the retrieval language that we employ) over streaming XML documents.

This paper is organized as follows. In Section 2, we overview the main modules of a system for XML semantic search over streams. In Section 3, we explain the syntax and the semantics of the retrieval language that we employ and in Section 4 we present the algorithms that perform semantic keyword search over streaming XML documents. Finally, we discuss some related work in Section 5 and conclude the paper in Section 6.

2. XML SEMANTIC SEARCH OVERVIEW

We assume that a number of web sources (or other data sources) are disseminating their content, which is encoded in XML, in an online fashion. Moreover, there is a mechanism for a user of the XML semantic search system to formulate (using the retrieval language that is described in Section 3) her queries and to register them with the system (i.e. to define how long the query is going to be active in the system). The registration mechanism resembles the query registration mechanisms that are supported in Data Stream Management Systems (DSMSs) and has been extensively studied elsewhere (e.g. [1]). At each point in time, there is a number of active semantic search queries within the system. In order to answer one such query, the query processing algorithm searches each document in the order of its arrival by performing a single sequential scan over its contents. The exact processing that takes places over the contents of the document is described in Section 4.

3. RETRIEVAL LANGUAGE

There are many works that tackle the problem of (semantic) XML search within persistent environments (i.e. environments where documents can be permanently stored, indexed and retrieved using random access patterns). The majority of those works (e.g. [6, 7, 17]) employs the *list of keywords* as its retrieval language. Upon receiving such a list, an XML (semantic) search system identifies the documents (or document fragments) that best satisfy this list. While processing the query, the system may choose to consider only the textual content, or both the textual content and the labels of each XML node in the searched collection. Alternatively, the query language can provide the user with better control on the document fragments to consider, by enabling the latter to explicitly specify constraints on the labels and/or the textual content if she so desires (e.g. [2, 16]). In our work, we opted for this latter option because of the better control that it provides while requiring minimal (or no) schema knowledge to formulate a query. Our language, whose syntax has been borrowed from [2, 16], is defined as follow:

DEFINITION 1. *(XML Semantic Search Query). Assume a(n) (infinite) stream of documents <d_i>, $i = 1, ...$ Moreover, let $\{N_i\}$ be the set of XML nodes within document d_i. An XML semantic search query Q over this XML document stream is a list of* **query 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 within document d_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 l.*
- *$::k$ if the tokenized 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.*

Each query term t_i results in a separate **input stream** *IS_i, which contains all the nodes that satisfy the term. The answer to Q consists of a stream of document fragments $< df_i >$, $i = 1, ...$ Each such document fragment contains at least one tuple $(n_1, .., n_m)$, $n_i \in IS_i$ with the additional constraint that the nodes in this tuple are meaningfully related.*

Node Relatedness Heuristics: The previous definition contains a requirement, namely, the *meaningful relatedness* of nodes, whose definition depends on user needs. Existing XML semantic search systems have employed a number of different heuristics to decide when a group of nodes contains meaningfully related nodes [2, 6, 9, 11, 14, 17, 18]. Moreover, very recently, a comparative survey of all those heuristics has appeared in [16]. The results of that work show that, among all the heuristics, the ones that are described in [6] and [17] (as well as their variants) return better quality results

(in terms of false positives and false negatives) over the rest of the heuristics.

The focus of our work is on how to support semantic search over streaming XML documents, and not on to devise new relatedness heuristic. Therefore, considering the results of the aforementioned survey (i.e. [16]), we decided to investigate whether the heuristics that were proposed in [2, 6, 17] can be efficiently employed in a streaming environment. Those heuristics, which employ the notion of the *Lowest Common Ancestor (LCA)* of two nodes n_1 and n_2 ([13]), are defined as follows. Assume that an XML semantic search query has produced two input streams IS_1 and IS_2:

DEFINITION 2. *(XRank Heuristic) The XRank system [6] asserts that two nodes n_1 and n_2 that belong to the same XML document d_i, with $n_1 \in IS_1$ and $n_2 \in IS_2$, are meaningfully related if there are no nodes $n_3 \in d_i$ and $n_4 \in d_i$ such that $LCA(n_1, n_3)$ is descendant of $LCA(n_1, n_2)$ or $LCA(n_2, n_4)$ is descendant of $LCA(n_1, n_2)$.*

DEFINITION 3. *(SLCA Heuristic) The SLCA heuristic [17] states that two nodes n_1 and n_2 that belong to the same document d_i, with $n_1 \in IS_1$ and $n_2 \in IS_2$, are meaningfully related unless there exist two other nodes n_3 and n_4 that also belong to d_i such that the $LCA(n_3, n_4)$ is a descendant of the $LCA(n_1, n_2)$. The intuition behind that (as well as the $XRank$) heuristics is that smaller trees contain more meaningfully related nodes. The difference between the $SLCA$ and the $XRank$ heuristics is that the first one disqualifies a pair of nodes n_1 and n_2 if there exists any other pair (n_3, n_4) with $LCA(n_3, n_4)$ being a descendant of $LCA(n_1, n_2)$, while $XRank$ has the additional constraint that $n_1 = n_3$ or $n_2 = n_4$.*

For ease of exposition, we provided the relatedness heuristics definitions in terms of only two nodes. However, our XML semantic search algorithms can handle any number of query terms. Before describing those algorithms, we provide an example of the employment of each of the above heuristics, in order to better illustrate their functionality.

EXAMPLE 2. *Assume a stream $< d_i > i = 1, ..$ of XML documents. Each document in the stream conforms to one of the two data sources from Example 1, which we described in Section 1. Assume also that a user is issuing the XML semantic search query: $author :: l_2$, $title ::$. Moreover, let's assume also that the first document to arrive at the search system is the one shown in Figure 1a. The input streams that correspond to the two query terms "$author :: l_2$" and "$title ::$" are: $< author_{\&3}, author_{\&6}, author_{\&13} >$ and $< title_{\&4}, title_{\&7}, title_{\&16} >$ respectively. The next step is to identify which of the pairs of those authors and titles are meaningfully related.*

The $XRank$ heuristic will accept the pairs $(author_{\&3}, title_{\&4})$, $(author_{\&6}, title_{\&7})$ and $(author_{\&13}, title_{\&16})$ as meaningfully related. However, it will reject the pair $(author_{\&3}, title_{\&7})$ since $LCA(author_{\&6}, title_{\&7}) = chapter_{\&5}$ is descendant of $LCA(author_{\&3}, title_{\&7}) = book_{\&2}$. It will also reject the pairs $(author_{\&3}, title_{\&16})$, $(author_{\&6}, title_{\&4})$, $(author_{\&6}, title_{\&16})$, $(author_{\&13}, title_{\&3})$ and $(author_{\&13}, title_{\&7})$ using similar arguments. Finally, it will return as answers the document fragment rooted at $book_{\&2}$, $chapter_{\&5}$ and $book_{\&12}$.

The $SLCA$ heuristic will work in a similar way. However, contrary to the $XRank$ heuristic, it will also reject the the pair $(author_{\&3}, title_{\&4})$, because $LCA(author_{\&6}, title_{\&7}) = chapter_{\&5}$ is descendant of $LCA(author_{\&3}, title_{\&4}) = book_{\&2}$. Thus, it will return as answers the document fragment rooted at $chapter_{\&5}$ and $book_{\&12}$.

Algorithm 1 Evaluate Search Heuristic

```
{Input: Set of Query Terms Q_t}
{       Stream of XML documents S_d}
{Output: Set of result XML nodes R_t}

R_t := ∅
for d ∈ S_d do
   s.clear() {initialize node stack}
   R_t := R_t ∪ SAX_Parse(d, Q_t, s, R_t)
end for
return R_t
```

Having described the retrieval language as well as the node relatedness heuristics that are employed by our XML semantic search system, we are now ready to describe how to efficiently process queries posed in that language in a streaming fashion.

4. QUERY EVALUATION

In order to support semantic search over XML documents, we need to devise algorithms that operate with a single scan (or at most small number of scans) over the incoming documents. Moreover their memory requirements should be bounded. In this section, we propose novel streaming algorithms that answer an arbitrary query expressed in the language previously introduced, while meeting the requirements that we just set.

Having a search query Q_t over a stream of documents S_d, the general search process is illustrated in Algorithm 1. As shown in the algorithm, each document d in the stream S_d is individually processed. The answers that exist within this document are collected and merged with the answers that are retrieved from the XML documents that have arrived previously, in R_t.

Each document is processed in S_d sequential fashion by a SAX parser, which returns five types of events for a document: $startDocument()$, $startElement(tag)$, $characters(text)$, $endElement(tag)$, and $endDocument()$. Our search algorithms then work through the SAX call back functions for those events.

Having described the base process in Algorithm 1, we start presenting the search algorithms by describing the search process in each XML document for the case when the $XRank$ heuristic is used. We then continue with the incorporation of the $SLCA$ heuristic in the search process.

4.1 Employing the XRank Heuristic

The $XRank$ heuristic (Section 3) implies that a document fragment df_1 belongs to the query answer set if: (a) it contains at least one node that satisfies a query term; (b) this node does not belong to another document fragment df_2 that resides within df_1 and belongs

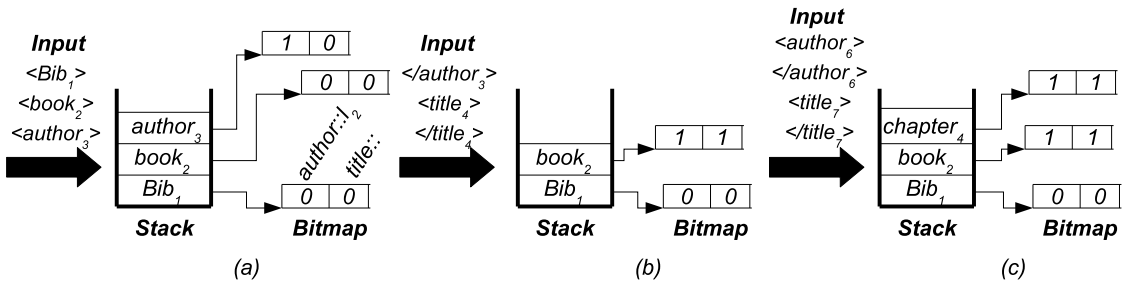

Figure 2: Semantic Searching Using the XRank Heuristic.

Algorithm 2 Evaluate XRANK Heuristic - Start Tag

{**Input**: Accessed Document Node n}
{ Set of Query Terms Q_t}
{ Node Stack s}

$s_n.label := n.label$ {create new stack node}
$s_n.term\_instances := \emptyset$
if $\exists\ t:: \in Q_t : t == n.label$ **then**
  $s_n.term\_instances(t::) := FOUND$
end if
if $\exists\ k \in Q_t : k == n.label$ **then**
  $s_n.term\_instances(k) := FOUND$
end if
s.push(s_n)

Algorithm 3 Evaluate XRank Heuristic - End Tag

{**Input**: Set of result XML nodes R_t}

$r_n := s.pop()$
if $r_n.term\_instances == COMPLETE$ **then**
  $R_t := R_t \cup \{r_n\}$
else
  for $tk \in r_n.term\_instances.keys$ **do**
    if $r_n.term\_instances(tk) == FOUND$ **then**
      $s.top().term\_instances(tk) := FOUND$
    end if
  end for
end if

Algorithm 4 Evaluate Search Heuristic - XML Text

{**Input**: Textual Content $text$}
{ Node Stack s}

for $word \in$ tokenize($text$) **do**
  if $\exists\ ::k \in Q_t : k == word$ **then**
    $s.top().term\_instances(::k) := FOUND$
  end if
  if $\exists\ k \in Q_t : k == word$ **then**
    $s.top().term\_instances(k) := FOUND$
  end if
  if $\exists\ t ::: k \in Q_t : (t == s.top().label\ AND\ k == word)$ **then**
    $s.top().term\_instances(t:::k) := FOUND$
  end if
end for

to the result set; and (c) there is at least one such node for each query term. Moreover, if a node belongs to a particular document fragment df then it is a descendant of the root node of df.

Taking those facts into consideration, the processing algorithm works as follows. First, note that, using a SAX parser, the algorithm scans the document in an in-order fashion and, as a result, a node is visited before all its descendants. In other words, when the SAX parser generates an open tag event (callback function *startElement*), it denotes the visit of a new node, while an *end tag* (*endElement*) denotes that the descendants of the corresponding node have all been visited. Each node that is visited is buffered within a stack and is popped from it when all its descendants have been visited. Each node that is pushed within the stack is associated with a bitmap, which maintains one bit for each query term in the query. Moreover, if the tag of this node satisfies a query term, the corresponding bit is set to 1. When the SAX parser parses XML text, this text is tokenized and for each token the algorithm checks whether: (a) the token satisfies a query term; and (b) combination of the the tag of the node currently at the top of the stack satisfies a query term. In each case, the corresponding bits at the bitmap of the top of the stack are set to 1. When all the descendants of a node n have been visited, the latter is popped from the stack. At this point, if all the bits of the corresponding bitmap bm_n have been set, the algorithm concludes that n is a root for a result to the search query. Otherwise, the bits in bm_n that are equal to 1 are copied to the corresponding places in the bitmap of the parent node (which now resides at the top of the stack). That way, the information about existence of nodes that satisfy query terms is propagated to ancestor nodes and the semantics of the language are supported.

The pseudocode of the entire query processing task is presented in Algorithms 2, 4 and 3. Those algorithms describe what is executed when the SAX parser visit an *open tag, XML textual content* and a *close tag* respectively. For simplicity, we omitted the code that handles XML attributes, as the corresponding process is identical with that of visiting a leaf node, whose tag and textual values are equal to the attribute name and value respectively. The example that follows illustrates the functionality of the algorithm that we just described:

EXAMPLE 3. *We utilize the same setting (i.e. input datasets and query) as the one described in Example 2. As before, assume that the first document to enter the system is the one shown in Figure 1a. Moreover, let's assume that the system employs the $XRank$ heuristic to process the retrieval queries that are posed by the user. Upon receiving the input document, the search procedure invokes the SAX parser and starts scanning the XML document in a pre-order fashion. In Figure 2a we show the state of the algorithm (i.e. the contents of the stack and the bitmaps) after the start tags of the nodes* $\mathbf{Bib_1}$, $\mathbf{book_2}$ *and* $\mathbf{author_3}$ *have been visited (in that*

order) and the textual content of author has been tokenized and processed. As shown in the figure, all three nodes currently reside in the stack. Moreover the bitmap that corresponds to the author element has been updated so that it reflects the fact that the token l_2 satisfies the query term $\mathbf{author::l_2}$. *In the next step, the end tag of* $\mathbf{author_3}$ *is visited and that node is popped from the stack. At this point, and because it's bitmap is not complete with* $1's$*, the* $1's$ *that exist in that bitmap are copied to the corresponding places in the bitmap of the* $\mathbf{book_2}$ *node. A similar copying happens when the end tag of node* $\mathbf{title_4}$ *is visited. The state of the algorithm then is depicted in Figure 2b. At this point one can see that node* $\mathbf{book_2}$ *is a root of a query result as its bitmap is now complete with* $1's$*. The same situation, i.e. a complete bitmap with* $1's$*, will happen for node* $\mathbf{chapter_4}$*, after the end tag of* $\mathbf{title_7}$ *is visited and that node is popped from the stack. This is shown in Figure 2c. As a result, in the next step and after the node has been popped from the stack, the document fragment rooted at that node is returned as result to the user. The algorithm continues operating in a this fashion until all the tags within the document have been visited.*

4.2 Employing the SLCA Heuristic

The $SLCA$ heuristic (Section 3) suggests that a document fragment df_1 belongs to the query answer set if (a) it contains at least one node that satisfies a query term, (b) there exists no other document fragment df_2 that resides within df_1 and belongs to the result set. The astute reader may have realized that the main difference with the $XRank$ heuristic is that if a node n is identified as root of a query result, then none of $n's$ ancestors can be root of another query answer.

With this observation in mind, the processing algorithm operates in a very similar way as the one that employs the $XRank$ heuristics (and which we have just described). In particular, it also scans the document in an in-order fashion (through SAX parser) and keeps each visited node within a stack until all the descends of that node have been processed. Each node that resides within the stack is associated with a bitmap whose structure and functionality is the same as those of the bitmap that are used in the previous algorithm (i.e. to keep track of the query terms that are satisfied within the fragment that is rooted at the corresponding stack node). Moreover, each node in the stack is also associated with a flag that checks whether the node can be root of a query result. When a node n is inserted into the stack this flag is set to $TRUE$ denoting that n can be such a root. When n is popped from the stack, the corresponding flag is tested and if it is still $TRUE$ $n's$ bitmap, bm_n, is also checked. If its bits are all 1, the algorithm concludes that n is a root of a query result. At this point, it sets the flag of the parent (which now resides at the top of the stack) to $FALSE$, as the latter cannot be a root of a result anymore. If the bitmap test does not succeed, the bits in bm_n that are equal to 1 are copied to the corresponding places in the bitmap of the parent node, just like the previous algorithm. If, on the other hand, $n's$ flag is $FALSE$, the flag of the parent is set to $FALSE$. That way the fact that a descendant node has been identified as root of a query result is upwards propagated and the algorithm correctly implements the semantics of the $SLCA$ heuristic.

The relevant pseudocode is presented in Algorithms 5, 4 and 6. Similarly with the previous section, those algorithms describe what is executed when the SAX parser visit an *open tag*, *XML textual content* and a *close tag* respectively. We further explain the functionality of the those algorithms with a concrete example:

Algorithm 5 Evaluate SLCA Heuristic - Start Tag

```
{Input: Accessed Document Node n}
{       Set of Query Terms Q_t}
{       Node Stack s}

s_n.label := n.label {create new stack node}
s_n.CAN_BE_SLCA := TRUE
s_n.term_instances := ∅
if ∃ t:: ∈ Q_t : t == n.label then
   s_n.term_instances(t::) := FOUND
end if
if ∃ k ∈ Q_t : k == n.label then
   s_n.term_instances(k) := FOUND
end if
s.push(s_n)
```

Algorithm 6 Evaluate SLCA Heuristic - End Tag

```
{Input: Set of result XML nodes R_t}
{       Node Stack s}

r_n := s.pop()
if r_n.CAN_BE_SLCA == FALSE then
   s.top().CAN_BE_SLCA := FALSE
else
   if r_n.term_instances == COMPLETE then
      R_t := R_t ∪ {r_n}
      s.top().CAN_BE_SLCA := FALSE
   else
      for tk ∈ r_n.term_instances.keys do
         if r_n.term_instances(tk) == FOUND then
            s.top().term_instances(tk) := FOUND
         end if
      end for
   end if
end if
```

EXAMPLE 4. *We change Example 3 with the assumption that the $SLCA$ heuristic is employed in order to process the retrieval queries. The algorithm operates in the same fashion as in the case when the $XRank$ heuristic is used up to the point that the node* $\mathbf{chapter_4}$ *is about to be popped from the stack. The state of the algorithm then is illustrated in Figure 3a. After node* $\mathbf{chapter_4}$ *is popped and identified as root of a query result, the algorithm decides that its ancestor (i.e. node* $\mathbf{book_2}$*) cannot be root of a query result anymore. In Figure 3a, the node* $\mathbf{book_2}$ *is shadowed with gray to denote the fact that its flag is going to be marked as $FALSE$. When that node is popped from the stack, the value of its flag is propagated its parent node (e.g. node* $\mathbf{Bib_1}$*) (as shown in Figure 3b) and the node is discarded. From then on, the computation of the algorithm continues in a similar fashion, until all the tags of the input document have been processed.*

5. RELATED WORK

Our work is related to the extensive research on XML search within persistent environments as well as the processing of structured queries over streaming data. In this section, we review those works and compare/contrast them with our contributions.

XML search within persistent environments. There are many specialized XML search engines that implement keyword search over XML data (e.g [2, 5, 7, 15]). Those engines identify sets of

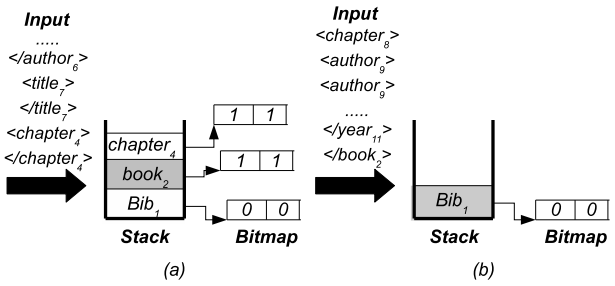

Figure 3: Semantic Searching Using the SLCA Heuristic.

nodes that are relevant to the user query (a node is relevant if its content and/or label satisfy a query keyword) and have close associations with each other. Subsequently, they return either the identified sets or appropriate document fragments that contain the nodes in these sets. Some of them employ *node relatedness heuristics* in order to choose combinations of relevant nodes that have close associations, while the majority produces all possible combinations of relevant nodes and utilizes a score function to choose the most relevant node combinations. Node relatedness heuristics have also been utilized by techniques that enhance existing structured XML retrieval languages (i.e. XPath and XQuery) with new operators [8, 11, 18] that enable enable the user to set the context of a particular query, even if she has only partial knowledge of the schema. Our work is orthogonal and complementary to all those works, as it focuses on supporting semantic search over streaming of XML data, as opposed to stored ones.

Structured Query Processing over Streams. There is currently a lot of interest on the issues that are involved when evaluating structured queries over relational streams (e.g. [1, 4]). Such streams are composed of relational tuples from diverse sources and are processed by a DSMS where the users have previously registered continuous queries. As new tuples arrive in the stream, all registered queries are evaluated, and the matched tuples are sent to the respective users.

In the realm of XML, a plethora of algorithms have been tailored for processing structured queries over XML streams. One group of relevant research focuses on XML filtering, where a set of queries is evaluated against the stream of documents to verify which queries are satisfied by each document. Among the earlier works on XML filtering, YFilter is probably the major representative [3]. It evaluates a set of queries by defining a finite state machine for all of them, such that common prefixes are processed only once. More recently, [10] introduced an profile-pruning technique that quickly identifies queries that are bound not to match the documents.

All those works impose the common requirement that a user needs to have deep and detailed knowledge of the structure (or schema) of the queried data sources in order to able to formulate meaningful queries. Our work attempts to lift this restriction by providing semantic keyword search over streaming XML.

6. CONCLUSIONS

We have introduced, motivated and tackled the problem of evaluating semantic search queries over streaming XML document. Building on relevant proposals for XML semantic keyword search within persistent environments, we described a retrieval language that enables semantic search over XML documents. Our language requires little or no knowledge of the schema of queried XML sources. Moreover, we devised novel query processing algorithms that efficiently process queries posed in that language in a streaming fashion. We are currently in the process of evaluating the effectiveness of our techniques within realistic streaming environments.

7. REFERENCES

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proc. of PODS*, 2002.

[2] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. In *Proc. of VLDB*, 2003.

[3] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Ficher. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM TODS*, 28(4):467–516, Dec 2003.

[4] L. Golab and M. T. Özsu. Issues in Data Stream Management. *SIGMOD Record*, 32(2):5–14, 2003.

[5] J. Graupmann, R. Schenkel, and G. Weikum. The SphereSearch Engine for Unified Ranked Retrieval of Heterogeneous XML and Web Documents. In *Proc. of VLDB*, 2005.

[6] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *Proc. of SIGMOD Conference*, 2003.

[7] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword Proximity Search on XML Graphs. In *Proc. of ICDE*, 2003.

[8] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *Proc. of VLDB*, 2004.

[9] Z. Liu and Y. Chen. Identifying Meaningful Return Information for XML Keyword Search. In *Proc. of SIGMOD Conference*, 2007.

[10] M. M. Moro, P. Bakalov, and V. J. Tsotras. Early Profile Pruning on XML-aware Publish/Subscribe Systems. In *VLDB*, 2007.

[11] T. L. Saito and S. Morishita. Amoeba Join: Overcoming Structural Fluctuations in XML Data. In *Proc. of WebDB*, 2006.

[12] R. Schenkel, A. Theobald, and G. Weikum. Semantic Similarity Search on Semistructured Data with the XXL Search Engine. *Inf. Retr.*, 8(4):521–545, 2005.

[13] A. Schmidt, M. Kersten, and M. Windhouwer. Querying XML Documents Made Easy: Nearest Concept Queries. In *Proc. of ICDE*, 2001.

[14] C. Sun, C.-Y. Chan, and A. K. Goenka. Multiway SLCA-based Keyword Search in XML Data. In *Proc. of WWW*, 2007.

[15] A. Theobald and G. Weikum. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. In *Proc. of EDBT*, 2002.

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

[17] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *Proc. of SIGMOD Conference*, 2005.

[18] S. Zhang and C. Dyreson. Symmetrically Exploiting XML. In *Proc. of WWW*, 2006.