



LCA-based algorithms for efficiently processing multiple keyword queries over XML streams



Evandrino G. Barros^a, Alberto H.F. Laender^b, Mirella M. Moro^b, Altigran S. da Silva^c

^aCentro Federal de Educação Tecnológica de Minas Gerais, Belo Horizonte, Brazil

^bUniversidade Federal de Minas Gerais, Belo Horizonte, Brazil

^cUniversidade Federal do Amazonas, Manaus, Brazil

ARTICLE INFO

Article history:

Received 5 September 2014

Received in revised form 18 December 2015

Accepted 13 March 2016

Available online 31 March 2016

Keywords:

Multi-query processing

Keyword-based queries

XML streams

LCA semantics

ABSTRACT

In a stream environment, differently from traditional databases, data arrive continuously, unindexed and potentially unbounded, whereas queries must be evaluated for producing results on the fly. In this article, we propose two new algorithms (called SLCAStream and ELCAStream) for processing multiple keyword queries over XML streams. Both algorithms process keyword-based queries that require minimal or no schema knowledge to be formulated, follow the lowest common ancestor (LCA) semantics, and provide optimized methods to improve the overall performance. Moreover, SLCAStream, which implements the smallest LCA (SLCA) semantics, outperforms the state-of-the-art, with up to 49% reduction in response time and 36% in memory usage. In turn, ELCAStream is the first to explore the exclusive LCA (ELCA) semantics over XML streams.

A comprehensive set of experiments evaluates several aspects related to performance and scalability of both algorithms, which shows they are effective alternatives to search services over XML streams.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

A stream data processing environment presents challenges that go beyond the data and query processing on traditional database systems [41]. Specifically, data arrive continuously in the form of *streams*, i.e., data are unindexed and potentially unbounded. Also, query processing must evaluate data as they come in and produce results on the fly, in one sequential scan and with minimum space for handling temporary results [1, 13]. As XML has become the standard for data exchange, processing XML data streams has soon conquered its place in the research and industry communities [41].

As an example of real-world applications, data streams are commonly used with monitoring sensors such as those in airplanes and weather stations. Indeed, sensor monitoring uses many sensors distributed in the physical world that generate data streams to be grouped, monitored and analyzed [5, 29, 37]. Another example includes network traffic management systems, which monitor various data streams (characterized as unpredictable, arriving at a high rate, and including both packet traces

E-mail addresses: eb Barros@decom.cefetmg.br (E. Barros), laender@dcc.ufmg.br (A. Laender), mirella@dcc.ufmg.br (M. Moro), alti@icomp.ufam.edu.br (A. da Silva).

and network performance measurements) [10, 35]. Finally, there is also a huge set of general purpose streaming data environments in which data are not specified in XML [8, 42]. Our techniques may also work on those, provided that a proper mapping is employed, such as the one proposed by Mendell et al. [31].

More recent applications that present large amounts of data streams are online social networks, most of them based on XML. In these environments, millions of users share interests, opinions, likes and news, which provide instantaneous information to evaluate social and market trends [30]. Likewise, RSS feed-based applications make large use of data streams. RSS is an XML feed format to publish frequently updated entities such as news headlines and blog entries on the Web. RSS popularity has motivated XML stream-oriented applications, such as the RSS Watchdog system, which is capable of clustering news and monitoring instant events over multiple XML streams [20].

XML streams must be processed rapidly and without retention. Retaining streams may cause data loss due to the large data traffic in continuous processing. This scenario becomes more complex when thousands of queries must be processed simultaneously. Different approaches explore single and multiple query processing. However, they are based on structural languages such as XPath, XQuery and SQL-like ones [11, 17, 18, 22, 23, 26, 33, 36, 37, 41], which requires knowledge of their syntax and the underlying data structure to formulate queries. Keyword-based languages are a usual approach to submit queries informally, because they require minimal or no schema knowledge to formulate queries [4, 9, 19, 21, 25, 39, 43, 44, 48]. Most keyword-based methods for XML, however, address archived documents [25, 44, 48]. More recent techniques have focused on keyword-based search algorithms for XML streams, but they only run one query at a time [4, 19, 39]. To the best of our knowledge, only the algorithm proposed by Hummel et al. [21] address the problem of processing multiple keyword-based queries over XML streams and, therefore, it is the current state-of-the-art. However, it is specific to the SLCA search semantics.

In general, keyword-based algorithms consider the lowest common ancestor (LCA) semantic [4, 9, 19, 21, 39, 43, 44, 48]. Specifically, the LCA of two nodes u and v in an XML tree is the common ancestor of u and v that is located farthest from the root. The most popular LCA-based algorithms use the smallest LCA (SLCA) and the exclusive LCA (ELCA) semantics. Particularly, ELCA handles the ambiguity that might exist in an XML document because the same content can occur at different levels, such as keywords that correspond to XML labels occurring in different schema elements [2]. Thus, ELCA is considered one of the most effective semantics because it returns a larger number of results [48].

As this work shows later, current approaches are limited to one or more of the following: processing one-single query at a time, not supporting schema-free keyword queries, and returning the whole document or insignificant parts of it with actual results. With such challenges in mind, we propose two new, efficient LCA-based algorithms for processing multiple keyword-based queries over XML streams: SLCAStream and ELCAStream. Both algorithms exploit processing properties based on the LCA semantics and introduce optimization strategies that improve the overall performance. SLCAStream provides a new approach to SLCA evaluation that avoids the usual bitmap processing strategy [4, 21, 39]. Hence, it significantly improves existing SLCA algorithms' response time and memory consumption. Then, ELCAStream extends SLCAStream for the ELCA semantics, making it the *first* ELCA-based algorithm for processing multiple queries over XML streams. Our extensive experiments analyze the performance and scalability of the proposed algorithms against the state-of-the-art. The results show that both SLCAStream and ELCAStream are efficient alternatives for processing keyword-based queries over XML streams. In summary, the contributions of this article are:

- Two new algorithms for processing multiple keyword-based queries over XML, called SLCAStream and ELCAStream, which provide, respectively, new approaches to SLCA and ELCA evaluation that avoid the usual bitmap processing strategy, thus significantly improving response time and memory consumption of existing algorithms. Moreover, ELCAStream is the *first* ELCA-based algorithm for XML query processing in a stream environment.
- A thorough evaluation of the two proposed algorithms that includes a performance comparison against the state-of-the-art algorithm considering both time and memory consumption.

The rest of this article is organized as follows. Section 2 discusses related work. Section 3 addresses some fundamental concepts related to our work. Section 4 describes the two proposed algorithms, SLCAStream and ELCAStream, and Section 5 presents our experimental evaluation. Finally, Section 6 presents our conclusions and insights for future work.

2. Related work

XML query processing has solutions ranging from simple axis evaluation to full language specifications over *stored* data [14, 18, 22, 25, 34, 46, 47], and from simply filtering (find the documents that match a query [27, 40, 45]) to full querying (find the document parts that match a query [35, 37]) for *streaming* data. Considering streaming data, all XML processing techniques may be categorized according to three dimensions: language (XPath/XQuery or keyword), processing (single-query or multi-query) and output semantics (filtering or querying). Most of them are based on XPath and XQuery semantics, whose process depends heavily on the document schemas [24]. Instead of going through such an extensive state-of-the-art, this section focuses on XML stream environments with keyword-based semantics. Nonetheless, existing surveys on the literature cover specific topics on *stored* XML data, such as XPath and XQuery [14, 18], keyword search [28] and recent approaches on LCA, SLCA and ELCA-based semantics [46, 47, 49]. Finally, Fig. 1 puts everything in perspective: the three dimensions for *stream* processing, state-of-the-art examples and where our contributions fit in the big picture (i.e., keyword-based multi-querying process).

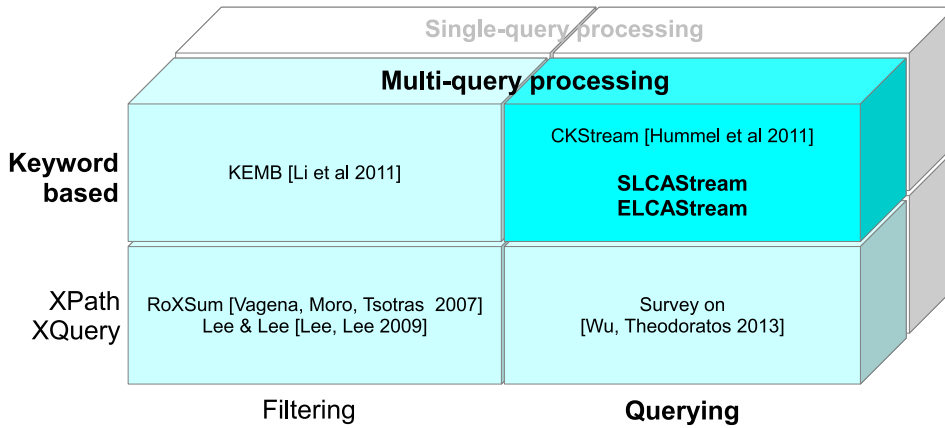


Fig. 1. Summary of XML data stream process in three dimensions, showing some state-of-the-art examples and where our contributions are.

Regarding single keyword-based queries over streams, Hristidis et al. [19] propose the SALowOne algorithm that works with minimum connected trees (MCTs) of an XML document containing terms of a single query. In its intermediary operations, it groups similar MCTs in reduced structures called GDMCTs (lowest grouped distance MCTs). These structures include only those nodes that meet the query terms and their LCA nodes. Vagena and Moro [39] propose two algorithms for semantic search over XML streams. Both algorithms are based on node relatedness heuristics for searching over XML documents (XRANK and SLCA). Barros et al. [4] propose an algorithm with a simple ranking strategy by combining the XRANK and SLCA algorithms proposed by Vagena and Moro [39]. Dimitriou et al. [9] address the problem of computing top-k-size LCAs as answers to keyword queries on tree-structured data. Finally, Zhou et al. [48] propose the Hash Count algorithm specific for the ELCA semantics, which presents performance improvements on the Indexed Stack algorithm [44]. These improvements involve specific archived data structures. Nonetheless, all aforementioned algorithms evaluate *one* query at time.

Hummel et al. [21] are the first to address the problem of multiple keyword-based queries over XML streams by proposing two algorithms called KStream and CKStream. However, both algorithms are limited to the SLCA semantics and their adaptation to ELCA is unfeasible. In summary, they are based on bitmaps and follow a similar approach, thus having similar time performance, whereas CKStream improves KStream memory consumption. In this article, we use CKStream as our main SCLA baseline algorithm. Particularly, CKStream uses a compact bitmap that represents the distinct terms for all queries. When an open node matches a query term, the corresponding bit is set to true in the bitmap entry. Upon closing a node, if all bits associated with a query have been set to 1, this means that the node matches the query being processed. Our SLCA algorithm, SCLASStream, is much simpler because it does not use a bitmap or any other additional data structure, since it is based on semantic properties, naturally derived from XML document traversal, as discussed in Section 3.

Approaches that process a single query count on a simple short data structure, such as a bitmap or MCTs, to record the occurrence of the query terms [4, 19, 39]. Their adaption to multiple queries demands controlling large bitmaps, complex MCTs or additional structures, such as specific indexes, as it is the case of CKStream. Our algorithms use additional structures, but not bitmaps. Our identification scheme is simple and allows us to derive semantic properties that provide efficient strategies for keyword-based query processing.

As we shall see, our algorithms process multiple keyword-based queries over XML streams according to *both* SLCA and ELCA semantics. Specifically, our SLCAStream significantly improves CKStream response time and memory consumption, whereas ELCASStream is the *first* ELCA-based algorithm for processing multiple queries over XML streams.

3. Fundamental concepts

This section overviews the fundamental concepts behind our work. We first summarize the keyword query language employed, and then describe new SLCA and ELCA processing properties, which are the basis of our proposed algorithms.

3.1. XML data model and query languages

An XML database (or document) is modeled as a tree structure in which the root represents the document root and each node represents a node label (e.g., *title*), a node value (e.g., “*Harry Potter and the Goblet of Fire*”) or both. This way, one single document carries information on both its data values and structure; hence, an XML query language must provide evaluation constraints for both as well [7,22]. The most common languages – XPath and XQuery – do so by providing an access path for navigating the document structure plus functions and predicates for inquiring its values. A common path constraint defines the axis of navigation. For example, the *ancestor/descendant* axis considers the whole tree under the ancestor node, whereas the

parent/child axis considers only the nodes at the next level under the parent node. Nonetheless, the user needs to know the structural organization (i.e., schema) of the documents in order to properly enjoy the access path methods.

3.2. Keyword-based query language

An XML database may be stored as a text document. Then, a pure keyword-based language is not enough for querying it, because it reduces the document structure and data values to plain text. Also, as aforementioned, an XML query language should be able to analyze both data and its structure, even using the keyword-based semantics.

Therefore, in order to provide better control while requiring minimal or no schema knowledge to formulate keyword-based queries, we adopt a simple query language inspired by [38] and [39]. According to this language, a keyword-based query q over an XML document stream d is a list of query terms (also denoted search terms) $\langle t_1, \dots, t_m \rangle$. Each query term is of the form $\ell :: k$, $\ell ::$, $:: k$, or k , where ℓ is an element label and k a keyword. Terms that involve element labels are called *structural terms*. A node n within a document d satisfies a query as follows:

- $\ell :: k$, if n 's label is equal to ℓ and its textual content contains the keyword k , being ℓ the label of the node in which k occurs directly;
- $\ell ::$, if n 's label is equal to ℓ ;
- $:: k$, if the textual content of n contains the keyword k ;
- k , if n 's label is equal to k or its textual content contains the keyword k .

Consider the XML document represented in Fig. 2 including books with their titles, authors and chapters. Table 1 presents queries that follow the adopted keyword-based query language and illustrates different scenarios regarding the user knowledge about the XML labels. It also presents the query specifications and their results. In queries q_1 , q_2 and q_3 , the user knows the labels and employs query terms of the form $\ell :: k$ and $\ell ::$ k respectively. In queries q_4 and q_5 , the user knows a single label (*title* and *author*, respectively), whereas in query q_6 the user has no knowledge of labels.

3.3. SLCA and ELCA semantics

Next, we first overview the lowest common ancestor (LCA) semantics. Then, we discuss the SLCA and ELCA semantics according to Xu and Papakonstantinou [43] and Zhou et al. [48], respectively.

Definition 1. Given an XML document d and a subset of nodes v_1, v_2, \dots, v_m from d matching a set of query terms t_1, t_2, \dots, t_n , the LCA of those nodes is a node e that is their common ancestor located farthest from the root of d .

As an example, consider the subtree rooted at node $book_5$ from the XML document in Fig. 2 and the query $q = \{title, author\}$. Then, the returned LCA nodes are $chapter_7$ and $book_5$. However, $book_5$ is an ambiguous answer because there are three pairs of descendant nodes that match the query, which are: i) $title_6$ and $author_{10}$, ii) $title_6$ and $author_9$, and iii) $title_8$ and $author_{10}$.

In order to solve this kind of ambiguity, the SLCA and ELCA semantics have been introduced [16, 43]. Definitions 2 and 3 formalize the two semantics.

Definition 2. Given a set of LCA nodes returned as the result of a query q on an XML document d , the corresponding SLCA nodes are the LCA nodes that contain no other LCA node as descendant.

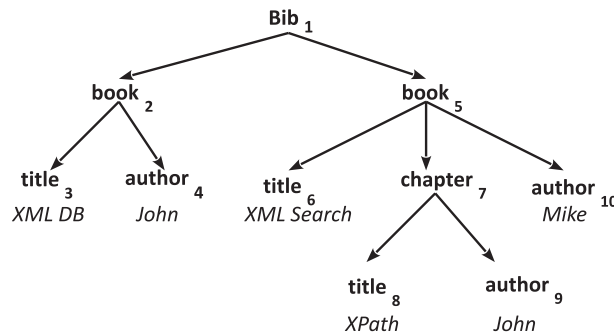


Fig. 2. Tree representation of an XML document.

Table 1
Examples of keyword-based queries.

Query	Specification	Result
q_1) title::author::	Books or chapters having title and author	<i>book₂</i> , <i>book₅</i> , <i>chapter₇</i>
q_2) title author	Books or chapters having title and author	<i>book₂</i> , <i>book₅</i> , <i>chapter₇</i>
q_3) title::XML author::John	Books or chapters having “XML” in their titles and written by “John”	<i>book₂</i>
q_4) title::XML John	Books or chapters having “XML” in their titles and written by “John”	<i>book₂</i>
q_5) XML author::John	Books or chapters having “XML” in their titles and written by “John”	<i>book₂</i>
q_6) XML John	Books having the terms “XML” and “John” anywhere	<i>Bib₁</i> , <i>book₂</i> , <i>book₅</i> , <i>chapter₇</i>

For the previous LCA example, the SLCA node is *chapter₇*, which includes only nodes *title₈* and *author₉*, thus removing the ambiguity of the query terms. Notice that the SLCA semantics defines a subset of LCA nodes that includes no LCA descendants.

Definition 3. Given a set of LCA nodes returned as the result of a query q on an XML document d , the corresponding ELCA nodes are those LCA nodes that contain at least one occurrence of each query term, excluding all their descendent LCA subtrees that also match any of those terms.

Following the previous example, ELCA nodes are *chapter₇*, which satisfies both query terms with no descendants, and *book₅*, which (excluding its LCA/SLCA descendant) also satisfies the query terms. Notice that node *Bib₁* is not an ELCA node because it does not contain any occurrence of the query terms as its own nodes.

In practice, the ELCA semantics subsumes SLCA, i.e., its result includes all SLCA nodes that match the query terms. Moreover, only ELCA handles the ambiguity that might exist among keywords [2]. Hence, ELCA is more comprehensive than SLCA, thus producing more query results. This makes ELCA a more appropriate semantics than SLCA when querying XML documents that contain the same content at different levels. For example, in Fig. 2, *title* occurs as a child of *book* as well as of *chapter*.

3.4. SLCA/ELCA processing properties

In a stream environment, XML documents are usually processed by a SAX parser, which generates sequential events for each XML tree node visited. Specifically: *startElement()* opens a new node, *characters()* is triggered when the current node has a text content and *endElement()* closes the current node (after all its descendants have already been visited). Upon opening a node, *startElement()* defines an *id* to the current node as a sequential number that corresponds to its processing order when traversing the tree in preorder (see the number assigned to each node in Fig. 2).

The *id* values encapsulate important properties for the SLCA and ELCA semantics, as discussed and exemplified next. The examples are based on the query $Q = \{\text{title}, \text{author}\}$ and the XML document in Fig. 2. Although we discuss these properties considering a single query, they equally apply to multiple queries processed at once.

Property 1. Given an XML document d , the *id* of any node v in d is smaller than the *id* of any of its descendants and greater than the *id* of its all previously closed nodes.

This property defines the sequential order of the document nodes when traversing the document tree in preorder (see the numbers in Fig. 2). The following properties are based on Property 1.

Property 2. Given a document d and a query q , let id_u and id_v be respectively the *ids* of two nodes u and v in d that match q . If u is the previously SLCA node and has been processed before v , then v can only be an SLCA node if id_v is greater than id_u .

In Fig. 2, node *chapter₇* satisfies Property 2 because its descendants match the query terms and its *id* is greater than the *id* of the last SLCA node processed, which is *book₂*. Notice, however, that even though *book₅* matches all query terms, its *id* is smaller than the *id* of *chapter₇*, which is the previous SLCA node when *book₅* is closed. Thus, *book₅* is not an SLCA node. Because *chapter₇* is closed before *book₅*, *chapter₇* becomes the last SLCA node, preventing *book₅* from being an SLCA node as well.

Property 3. Let v be an SLCA node returned as a result for query $q = \{t_1, \dots, t_n\}$. Then, id_v is less than or equal to the *id* of its descendant nodes that match the term t_i ($1 \leq i \leq n$) and greater than the id_u of the previous SLCA node u .

Property 3 guarantees that if node v satisfies query q , this node or some of its descendants satisfies all query terms, meaning that its *id* is greater than or equal to the *id* of each node that satisfies a query term. Notice that Property 4 also guarantees Property 2. Therefore, if node v satisfies Property 4, it also satisfies the SLCA semantics, according to Property 2. In Fig. 2, *chapter₇* satisfies Property 4 because its *id* is less than or equal to the *ids* of its descendants that match the query q terms. In addition, the *id* of node *chapter₇* is greater than the *id* of the previous SLCA node that satisfies q , which is *book₂*.

Property 4. Let q be a query comprising the set of terms $\{t_1, t_2, \dots, t_n\}$ and $IdList[t_i]$ ($t_i \in q$) be the id list of nodes where t_i occurs in the LCA descendants of node v . For each term t_i , if there is at least one node u that belongs to the subtree rooted at v and satisfies t_i , and id_u does not occur in $IdList[t_i]$, then v is an ELCA node.

According to Property 4, v is an ELCA node if it has at least one occurrence of each query term, excluding all query term occurrences of its LCA descendants. As an ELCA node, v has LCA descendants that are SLCA nodes. Notice that id_v is less than or equal to id_u and less than id_p , where $id_p \in IdList[t_i]$ ($t_i \in q$). As an example of Property 4, consider $v = book_5$. When closing this node, its descendants that match the query terms are $title_6$ and $author_{10}$. Notice that these two nodes do not occur under node $chapter_7$, which is an LCA descendant of $book_5$. Thus, $book_5$ is an ELCA node.

4. Proposed algorithms

Having presented the SLCA/ELCA processing properties in the previous section, we now describe our two new algorithms. We first introduce SLCAStream, which is a new algorithm for SLCA evaluation over an XML data stream. It avoids the usual bitmap processing and uses specific optimization strategies to improve the evaluation performance of multiple queries. SLCAStream is based on Property 4, which allows the identification of SLCA nodes based only on their id values. In practice, SLCAStream is the starting point for ELCAStream, as the latter starts the evaluation of ELCA nodes from the SLCA nodes identified by the SLCAStream strategy. Then, we describe the ELCAStream algorithm, which is based on Property 4. In other words, ELCAStream considers a node v as an ELCA result if v has at least one occurrence for each query term, excluding those term occurrences that already contribute to its LCA descendant nodes.

Before describing our algorithms, we first introduce the data structures used by our algorithms (Section 4.1) and then present the general multi-query algorithm that triggers our query evaluation process (Section 4.2). Afterwards, in Sections 4.3 and 4.4, we go over our two algorithms, respectively SLCAStream and ELCAStream. We finalize by analyzing in Section 4.5 the complexity of both algorithm.

4.1. Data structures

Parsing stack. Our algorithms use a parsing stack S for keeping the XML nodes open during the SAX parser. Its top entry corresponds to a node v being processed and the previous entries correspond to ancestor nodes of v in the path from v to the document root. Each entry is popped from the stack when its corresponding node and all its descendants have been visited. Each entry in the parsing stack stores: (i) the XML node label and (ii) a set (called *used_queries*) containing the queries whose terms match keywords in that node or any of its descendants, either as labels or values.

Query index. Upon the traversal of a document, it is necessary to look for queries matching text elements or labels. Thus, our algorithms rely on a specific index, called *query_index*, to avoid looking up each query individually. In this index, each entry represents a query term and refers to queries in which this term occurs. This query index is built directly from the set of queries submitted by the user. Handling new submitted queries requires rebuilding the query index. This is a rather simple task and requires no effort from users or developers. It is reasonable to expect that the set of submitted queries is stable as long as streams continually arrive.

Inverted lists. As shown in Subsection 3.4, by only knowing the nodes where the query terms occur, it is possible to establish if a candidate node is ELCA. For this, ELCAStream requires inverted lists for indexing query term occurrences while processing a document.

Specifically, it uses two types of inverted list. The first one is the inverted list G , whose entries represent the query terms that match already processed nodes and its values correspond to the ids of such nodes. For instance, consider the XML document in Fig. 2 and the query $q = \{title, author\}$. When processing node $title_8$, the *title* inverted list is $G[title] = \{3, 6, 8\}$ and the *author* inverted list is $G[author] = \{4\}$.

The second type of inverted list, called g , is built for all distinct query terms $T = \{t_1, t_2, \dots, t_n\}$. Its entries represent the terms t_1, t_2, \dots, t_n and its values correspond to node ids that match these terms up to the node being processed, thus contributing to an LCA result. In other words, the inverted list g contains id values of all contributing nodes for LCA results up to the node being processed.

We consider as a *contributing node* one that satisfies a query term of q . In our example, considering the query $q = \{title, author\}$, after closing node $chapter_7$, the inverted list g for the term *title* is $g[title] = \{3, 8\}$, since such values correspond respectively to nodes that contribute to $book_2$ and $chapter_7$ SLCA results, which are also LCA results for q . When closing node $chapter_7$, the inverted list g for the term *author* is $g[author] = \{4, 9\}$. Likewise, considering the term *author*, these values correspond to nodes that contribute to $book_2$ and $chapter_7$ SLCA results, which are also LCA results for q .

Both inverted lists are essential for ELCAStream. However, SLCAStream uses a simplified version of G . According to Property 4, SLCAStream requires only the last node id value for the query term occurrences. This simplifies the inverted list G that can be represented by a vector, whose keys are the query term occurrences and the corresponding values are the last node id values that correspond to the query terms. This means that, instead of saving a list of values for each entry as a usual inverted list, G is simplified and only saves the last id occurrence of the node as a vector. This simplification reduces SLCAStream memory consumption.

Approaches that process a single query usually rely on a simple data structure, such as a bitmap, which maintains occurrences for the terms of a single query [4, 39]. Adapting them for processing multiple queries demands controlling large bitmaps and additional structures such as a query index. For example, Hummel et al. [21] have made some effort to adapt a single query approach to process multiple queries, which included the use of a query index. Our algorithms use additional structures, but not bitmaps. Our node identification strategy is very simple and easily supports the SLCA/ELCA processing properties, thus facilitating the evaluation of keyword-based queries.

4.2. General multi-query procedure

Given a set of queries Q from distinct users' profiles being processed against a stream D of documents, Fig. 3 presents the general multi-query procedure that implements our algorithms. Each document d in D is individually processed (Lines 3 to 6) to check whether it satisfies any query in Q . The results found are then collected and returned (Lines 5 and 6). This is accomplished at the same time for all queries $q \in Q$, and results are individually collected in each r_i , which contains *id* values of the corresponding result nodes.

Each document d_j in D is processed by a SAX parser, which triggers three types of event for a document: *startElement(tag)*, *characters(text)* and *endElement(tag)*. Our algorithms then work by means of SAX Callback Functions for these events. The parser is called to action in Line 5.

4.3. The SLCAStream algorithm

SLCAStream is based on the SLCA Property 3 presented in Subsection 3.4. In addition, it uses two optimization strategies to improve time performance and memory consumption (as explained below). SLCAStream consists of three callback functions, SLCAStream.Start, SLCAStream.Text and SLCAStream.End, which handle respectively the SAX events *startElement(tag)*, *character(text)* and *endElement(tag)*. These functions are presented respectively in Figs. 4, 5 and 6, and discussed next.

SLCAStream.Start (Fig. 4) updates the parsing stack pushing the current node being processed if its label match any query term of the forms ℓ or $\ell ::$. SLCAStream pushes entries to the stack only if necessary, thus reducing stack operations and, therefore, improving response time and memory consumption. This is our first optimization strategy. Initially, SLCAStream.Start generates each *id* node (Line 2) and records on the stack top entry the high of the node being processed (Line 3). Then, it stores all queries with terms of the form ℓ or $\ell ::$ on the stack top entry, and updates the inverted list G and the *matching_terms* field (Lines 9 to 14). This field stores the number of query terms that have matched the node being processed or any of its descendants, and it is used by SLCAStream.End to select which queries will be evaluated. This constitutes our second optimization strategy and is the one responsible for considerably improving response time. Note that our optimization strategies are simple and specific to the stream processing context and, most importantly, have not yet been exploited by any similar work (such as [19] and [21]).

As an example, consider the query q_1 in Table 1 and the XML document in Fig. 2. When closing node *title*₈, SLCAStream skips the evaluation of query q_1 because this node matches only one q_1 term. However, node *chapter*₇ has two descendants that partially or totally match q_1 terms. Thus, when closing node *chapter*₇, SLCAStream evaluates each q_1 term individually, since node *chapter*₇ descendants may match q_1 terms totally or partially. A partial match would occur if *chapter*₇ had more than one occurrence for a single query term, but none for another term. Note that this optimization has no effect on the SLCA semantics because SLCAStream guarantees its compliance.

SLCAStream.Text (Fig. 5) is similar to SLCAStream.Start in the sense that it updates the parsing stack if the queries include text tokens as a term (terms such as k or $:: k$), eventually combined with its label (terms of the form $\ell :: k$). SLCAStream.Text pushes only the necessary entries to the stack, i.e., only those nodes being processed whose entries are not on the top of the stack (Lines 3 to 8). Specifically, SLCAStream.Start processes each text token of the node being currently evaluated, in any of its possible variations (k , $:: k$ or $\ell :: k$) (Lines 9 to 21), by storing all queries containing these variations on the stack top entry and updating the inverted list G and the *matching_terms* field (Lines 13 to 19).


Finally, SLCAStream.End (Fig. 6) checks which nodes or their descendants match the stored queries according to the SLCA semantics. For this, it only pops the stack top entry if this entry corresponds to the node being currently processed (Lines 1 to 31). Then, it processes each query stored in the stack top entry (Lines 5 to 21) whose number of terms is less than or equal to the value of the *matched_terms* field on the stack top entry. Then, according to Property 2, SLCAStream.End evaluates if the

Algorithm Multiple_Queries

Input: A stream of XML documents D

1. **let** $Q = q_1, \dots, q_n$ be a set of queries from users' profiles
2. **while** D is not empty **do**
3. get a new document d_j from D
4. $S.clear()$ {initialize the stack}
5. $\langle r_1, \dots, r_n \rangle := \text{SAX_Parser}(d_j, \langle q_1, \dots, q_n \rangle, S)$
6. **return** r_1, \dots, r_n
7. **end while**

Fig. 3. General multi-query procedure.



```

Callback Function SLCAStream.Start
Input: The parsing stack S
Input: The XML node  $e$  being processed
Input: The node identification  $id$  for node  $e$ 
1.  $\ell := \text{label}(e)$ 
2.  $id := id + 1$  {next  $id$  for the new node}
3.  $\text{node\_path}[e.\text{height}].id = id$ 
4. if  $\text{query\_index}[\ell] \neq \emptyset$  or  $\text{query\_index}[\ell::] \neq \emptyset$  then
5.    $sn.id := id$ 
6.    $sn.height := e.height$ 
7.    $\text{terms} := \{\ell, \ell::\}$ 
8.   for all  $t \in \text{terms}$  do
9.      $Q := \text{query\_index}[t]$  {set queries with term  $t$  to  $Q$ }
10.    if  $Q \neq \emptyset$  then
11.       $sn.\text{used\_queries} := sn.\text{used\_queries} \cup Q$ 
12.      add  $id$  to  $G[t]$ 
13.       $sn.\text{matched\_terms} = sn.\text{matched\_terms} + 1$ 
14.    end if
15.  end for
16.   $S.\text{push}(sn)$  {create new stack entry}
17. end if

```

Fig. 4. SLCAStream.Start callback function.


current node or any of its descendants match all current query terms (Lines 8 to 12). If the node being processed satisfies Q completely, SLCAStream.End checks if it is an SLCA node according to Property 2 (Lines 13 to 19). Lastly, it updates the parsing stack (Lines 22 to 30) by either storing on its top entry the queries being currently processed and updating its $sn.\text{matched_terms}$ field (Lines 23 to 25), or updating its height and id fields (Lines 26 to 30).

4.4. The ELCAStream algorithm

ELCAStream is a direct implementation of Property 4, presented in Subsection 3.4, which states that a node v is an ELCA result if v has at least one occurrence for each query term, except term occurrences that contribute to its LCA node descendants. Thus, for each query term, the algorithm stores in the inverted list G the id values of the processed nodes that include such a term and in the inverted list g the id values of all LCA nodes that contribute to query results. Therefore, both lists are used for processing the ELCA semantics as we shall see next.

For most ELCA node evaluations, Property 4 can be directly assessed by using the inverted lists G and g . However, we can evaluate ELCA nodes by using a faster and more practical strategy by starting with SLCA nodes, which are also ELCA nodes. Therefore, ELCAStream uses the same SLCAStream strategy for finding the lowest ELCA nodes in order to enforce a tighter semantics as defined by Property 4.

Next, Subsection 4.4.1 describes how the inverted lists are used for identifying ELCA nodes and Subsection 4.4.2 presents the callback functions that implement ELCAStream.



```

Callback Function SLCAStream.Text
Input: The parsing stack S
Input: The XML node  $e$  being processed
Input: The node identification  $id$  for node  $e$ 
1.  $\ell := \text{label}(e)$ 
2.  $\text{new\_stack\_entry} := \text{false}$ 
3. if  $S.\text{height} = e.\text{height}$  then
4.    $sn := \text{top}(S)$  { $sn$  pops the top entry in the stack to  $sn$ }
5. else
6.    $sn.id := id$ 
7.    $sn.height := e.height$ 
8. end if
9.  $K := \text{set of tokens in node } e$ 
10. for all  $k \in K$  do
11.    $\text{terms} := \{k, ::k, \ell::k\}$  {possible terms containing  $k$ }
12.   for all  $t \in \text{terms}$  do
13.      $Q := \text{query\_index}[t]$  {set queries with term  $t$  to  $Q$ }
14.     if  $Q \neq \emptyset$  then
15.        $\text{new\_stack\_entry} := \text{true}$ 
16.       add  $id$  to  $G[t]$ 
17.        $sn.\text{used\_queries} := sn.\text{used\_queries} \cup Q$ 
18.        $sn.\text{matched\_terms} = sn.\text{matched\_terms} + 1$ 
19.     end if
20.   end for
21. end for
22. if  $\text{new\_stack\_entry}$  then
23.    $S.\text{push}(sn)$  {create new stack entry}
24. end if

```

Fig. 5. SLCAStream.Text callback function.


```

Callback Function SLCAStream.End
Input: The parsing stack S
Input: The XML node  $e$  that is being closed
1. if S.height =  $e.height$  then
2.    $sn := \text{pop}(S)$  {pops the top entry in the stack to sn}
3.    $id := sn.id$ 
4.    $\ell := sn.label$ 
5.   for all  $q \in sn.used\_queries$  do
6.     if  $sn.matched\_terms \geq q.terms.size()$  then
7.       COMPLETE := true
8.       for all  $t \in q.terms$  and COMPLETE do
9.         if  $id < \text{last}(G[t])$  then
10.          COMPLETE := false
11.        end if
12.      end for
13.      if COMPLETE then
14.        {checks if  $id$  is an SLCA result}
15.        if  $id > q.last\_result.id$  then
16.           $q.results := q.results \cup \{sn\}$ 
17.           $q.last\_result.id := id$ 
18.        end if
19.      end if
20.    end if
21.  end for
22.   $tn := \text{top}(S)$  {tn points to the top entry in the stack}
23.  if  $(sn.height - tn.height) = 1$  then
24.     $tn.used\_queries = \text{top}.used\_queries \cup sn.used\_queries$ 
25.     $tn.matched\_terms = tn.matched\_terms + sn.matched\_terms$ 
26.  else
27.     $sn.height = sn.height - 1$ 
28.     $sn.id = \text{node\_path}[e.height-1].id$ 
29.    S.push(sn)
30.  end if
31. end if

```

Fig. 6. SLCAStream.End callback function.

4.4.1. Using inverted lists to identify ELCA nodes

The inverted lists keep the node id values in ascending order as they are added to them. Such a feature is essential for evaluating ELCA nodes with ELCAStream.

Using the inverted list G (and applying [Property 1](#)), it is possible to identify the node occurrences for a term t that are descendant nodes of the node v being closed (at this point all descendant nodes have already been closed). Thus, a search in the inverted list $G[t]$ for id values greater than or equal to v 's id results in id values of nodes containing the term t . For instance, consider the document in [Fig. 2](#) and the query $q = \{title, author\}$. When closing node $book_5$ ($id = 5$), a search for $title$ occurrences in $G[title]$ results in the set of nodes $\{6, 8\}$. Likewise, a search for $author$ occurrences in $G[author]$ results in the set of nodes $\{9, 10\}$.

Next, using the inverted list g , it is possible to identify the LCA contributing nodes for the term t that are descendant nodes of the v node being closed. For instance, when closing node $book_5$, a search for $title$ occurrences in the inverted list g results in the set of nodes $\{8\}$ and a search for $author$ occurrences results in the set of nodes $\{9\}$.

Finally, by comparing the results, we conclude that node $book_5$ is an ELCA result because the sets of nodes $\{6\}$ and $\{10\}$ match the query q and do not occur as LCA contributing nodes of $book_5$ descendants.

4.4.2. Callback functions

Similarly to SLCAStream, ELCAStream consists of three callback functions, each one corresponding to a distinct event. The functions ELCAStream.Start and ELCAStream.Text are equivalent to their counterparts SLCAStream.Start and SLCAStream.Text respectively, and, therefore, are not described here. Function ELCAStream.End, although based on SLCAStream.End, requires handling the inverted lists G and g for ELCA evaluation and is discussed next.

ELCAStream.End ([Fig. 7](#)) evaluates which nodes or their descendants match the stored queries, following the ELCA semantics. First, it pops the stack top entry if this entry corresponds to the node being processed (Line 2). Then, it evaluates each stored query in this entry whose number of terms is less than or equal to the value of the $sn.matched_terms$ field (Lines 6 to 38). ELCAStream.End also tests if the node being processed or any of its descendants has occurrences for all terms of the current query (Lines 8 to 12). If so, the node satisfies the current query completely. Next, ELCAStream.End tests if it is an SLCA node, according to [Property 4](#) (Lines 15 to 20). If so, it is also an ELCA node. Thus, ELCAStream.End stores this node as a query result, updates the inverted list g and records it as the last SLCA result. If the node is a non-SLCA node, it can be an ELCA one. Then, according to [Property 4](#), ELCAStream.End tries to find at least one node that satisfies each term of the current query, excluding its descendants that contribute to LCA results (Lines 21 to 36). While verifying this, ELCAStream.End updates the inverted list g (Line 30). If the node is an ELCA node for the current query, ELCAStream.End adds it to the result list (Line 34). Similarly to SLCAStream.End, ELCAStream.End updates the parsing stack S accordingly (Lines 40 to 48).

4.5. Complexity analysis

For analyzing the time complexity of our algorithms, we consider each callback function individually. For comparison purposes, operations such as index and inverted list lookups, set assignments and initializations are considered to be $\mathcal{O}(1)$.

```

Callback Function ELCAStream.End
Input: The parsing stack S
Input: The XML node  $e$  that is ending
1. if S.height =  $e.height$  then
2.    $sn := \text{pop}(S)$  {pops the top entry in the stack to sn}
3.    $id := sn.id$ 
4.    $\ell := sn.label$ 
5.   for all  $q \in sn.used\_queries$  do
6.     if  $sn.matched\_terms \geq q.terms.size()$  then
7.       COMPLETE := true
8.       for all  $t \in q.terms$  and COMPLETE do
9.         if  $id < last(G[t])$  then
10.          COMPLETE := false
11.        end if
12.      end for
13.      if COMPLETE then
14.        {checks if  $id$  is an SLCA result}
15.        if  $id > q.last\_resultId$  then
16.           $q.results := q.results \cup \{sn\}$ 
17.           $q.last\_resultId := id$ 
18.          for all  $t \in q.terms$  do
19.            add  $\{id_1, \dots, id_k\}$  to  $g[t]$ , where  $id_i (1 \leq i \leq k) \in G[t]$  and  $id_i \geq sn.id$ 
20.          end for
21.        else
22.          CAN_BE_ELCA := true
23.           $IdList[q.terms[1], \dots, q.terms[q.terms.size()]] := \emptyset$ 
24.          for all  $t \in q.terms$  do
25.            add  $\{id_1, \dots, id_k\}$  to  $IdList[t]$ , where  $id_i (1 \leq i \leq k) \in G[t]$  and  $id_i \geq sn.id$ 
26.             $IdList[t] := IdList[t] - \{id_1, \dots, id_k\}$ , where  $id_i (1 \leq i \leq k) \in g[t]$  and  $id_i \geq sn.id$ 
27.            if  $IdList[t] = \emptyset$  then
28.              CAN_BE_ELCA := false
29.            else
30.              add  $IdList[t]$  to  $g[t]$ 
31.            end if
32.          end for
33.          if CAN_BE_ELCA then
34.             $q.results := q.results \cup \{sn\}$ 
35.          end if
36.        end if
37.      end if
38.    end for
39.     $tn := \text{top}(S)$  {tn points to the top entry in the stack}
40.    if  $(sn.height - tn.height) = 1$  then
41.       $tn.matched\_terms = tn.matched\_terms + sn.matched\_terms$ 
42.       $tn.used\_queries = tn.used\_queries \cup sn.used\_queries$ 
43.    else
44.       $sn.height = sn.height - 1$ 
45.       $sn.id = \text{node\_path}[e.height-1].id$ 
46.      S.push(sn)
47.    end if
48.  end if
49. end if

```

Fig. 7. ELCAStream.End callback function.

Initially, SLCAStream depends on the number Q of queries to be processed. Thus, the time complexity of SLCAStream.Start is $\mathcal{O}(Q)$ in the worst case, i.e., when all queries are evaluated. Similarly, SLCAStream.Text depends on Q . However, it also depends on the number K of keywords that occur in a node, which means that, in the worst case, its time complexity is $\mathcal{O}(K \times Q)$. As we consider K a small constant because we are dealing with data-centric XML documents, SLCAStream.Text time complexity is actually $\mathcal{O}(Q)$. Finally, once SLCAStream.End performs similar operations, its time complexity is also $\mathcal{O}(Q)$. Thus, the time complexity of each SLCAStream function is $\mathcal{O}(Q)$. Considering that these functions are executed for N nodes, the SLCAStream final time complexity is $\mathcal{O}(N \times Q)$, being N and Q linear factors.

Regarding ELCAStream, its callback functions have the same time complexity, except ELCAStream.End. This function, in the worst case, evaluates all Q queries and their respective terms. Since the number of terms per query is small, we consider it as a constant. Thus, the ELCAStream.End time complexity is $\mathcal{O}(Q)$. Since, this function is executed for N nodes, in the worst case its time complexity is $\mathcal{O}(N \times Q)$. Considering that the other two functions have this same complexity, the final ELCAStream complexity is $\mathcal{O}(N \times Q)$, being N and Q linear factors. Notice that we consider invert list operations, which are essential operations for this algorithm, as $\mathcal{O}(1)$, since they are implemented by set operations in many programming languages with this complexity.

To estimate the space complexity of SCLASStream, we consider that its parsing stack controls Q queries. Specifically, the parsing stack stores the queries that must be evaluated during the document traversal. Moreover, each parsing stack has up to H entries, where H is the XML document height. However, we assume that H is a small constant h . Thus, the space complexity for the parsing stacks is $h \times \mathcal{O}(Q) = \mathcal{O}(Q)$. In the worst case, in which all queries must be evaluated for all N visited nodes, the space complexity is $N \times \mathcal{O}(Q) = \mathcal{O}(N \times Q)$. In addition, SCLASStream saves the last node id occurrence for each term in T , being T the set of all distinct query terms, which yields a space complexity of $\mathcal{O}(N \times Q) + \mathcal{O}(T)$. Regarding the *query_index* space complexity, each *query_index* entry stores a term and the ids of the queries in which it occurs. In the worst case, all Q queries include all terms in T . Thus, the space complexity of *query_index* is $\mathcal{O}(Q \times T)$. Summing up, the space complexity of SCLASStream is $\mathcal{O}(N \times Q) + \mathcal{O}(T) + \mathcal{O}(Q \times T) = \mathcal{O}(N \times Q) + \mathcal{O}(Q \times T)$, being N , Q and T linear factors.

Table 2

Details of the datasets used in the experiments: dataset name, number of documents in the dataset, average height from root to leaf node, number of elements per schema, average number of nodes in each document, average number of objects (each distinct path from root to leaf node) and average size.

Dataset	#Docs	Avg. height	#Elem.	Avg.# of nodes	Avg.# of objects	Avg. size
SIGMODR	18	7	13	1697	167	58 KB
XMARK	5	5	74	10312	5515	754 KB
ISFDB	10	2	11	41637	4110	1.3 MB

Regarding ELCAStream space complexity, we have to consider two additional data structures as compared to SLCAStream, which are the inverted lists g and G (differently from G , g only saves LCA nodes). These two lists store the ids of those nodes that have matched terms in T during the document traversal. Thus, in the worst case, when each term occurs in all N nodes, the space complexity of both inverted lists is $\mathcal{O}(T \times N)$. This means that the final ELCAStream space complexity is $\mathcal{O}(N \times Q) + \mathcal{O}(Q \times T) + \mathcal{O}(T \times N)$, being N , Q and T linear factors.

5. Experimental evaluation

In this section, we empirically evaluate the performance of the two proposed algorithms in terms of processing time and memory space. Specifically, the performance experiments consist of processing XML document streams against a set of queries simultaneously posed and measuring both the time spent and the memory consumed. The datasets and queries employed in this experimental evaluation are described next.

5.1. Experimental setup

All algorithms were implemented using Java and the SAX API from Xerces Java Parser. The query indexes and other data structures were kept entirely in memory. All experiments were performed in an Intel 1.8 GHz Core i7 computer with 4 GB of memory.

Datasets. The performance evaluation employed three distinct datasets. The first, SIGMODR, contains data from the table of contents of past SIGMOD Record issues¹. The second, XMARK, is a well-known synthetic XML auction dataset² [12]. The last, ISFDB, consists of bibliographic data from fiction books available on the ISFDB website³. We divided each original dataset into several documents in order to simulate an actual stream environment. The average sizes of SIGMODR, XMARK and ISFDB documents are 58 KB, 754 KB and 1.3 MB, respectively. Notice that the three datasets have different orders of magnitude. Table 2 details these three datasets. We used several documents per stream to obtain significant measures. The size of the datasets is not important for our experiments because they are processed document by document. Despite that, except for SIGMODR, the other two datasets, XMARK and ISFDB, are much larger than the average size of XML files found on the Web, which is 223 KB [15]. SIGMODR includes the deepest documents. ISFDB includes flat, but large documents. XMARK includes deep documents with repetitive, recursive elements. It also contains more objects than the other two datasets. For this regard, we considered each distinct XML node path as an object.

Queries. For our performance evaluation, we used the query language from Section 3.2 and randomly generated sets of queries using data available from each dataset. For each experiment, we generated different sets of queries by varying the number and type of terms in each one. We used up to 50,000 queries in each experiment for each dataset.

5.2. Baseline algorithms

We compared SLCAStream with CKStream, the SLCA stream-based algorithm [21] that, to the best of our knowledge, is the current state-of-the-art. CKStream implements a parsing stack whose entries are associated with visited nodes during document traversing. Each stack entry contains a single and compact query bitmap (called query bitmap), representing the distinct terms of all queries. When an opened node matches some query terms, their corresponding bits are set to true in the entry bitmap. Upon closing a node, if the bits associated with a query are complete, the node satisfies the query being processed. As an example, Fig. 8 (left) shows the query bitmap for those queries in Table 1. Notice that some terms occur in more than one query. However, each one is associated with a single bit. For instance, the query term *title:: XML* occurs in queries q_1 and q_2 , and is only associated with the fifth (t_5) bit of the query bitmap. Fig. 8 (right) presents the configuration of this bitmap when the node *chapter₇*, which occurs in the XML document of our running example (Fig. 2), is closed. This bitmap configuration summarizes which query terms occur in the descendants of node *chapter₇*. Thus, when closing this node, the algorithms find out that the bits corresponding to q_1 and q_2 terms are complete and, therefore, node *chapter₇* is a result for both queries.

¹ <http://www.sigmod.org/publications/sigmod-record/>

² <http://www.xml-benchmark.org>

³ <http://www.isfdb.org>

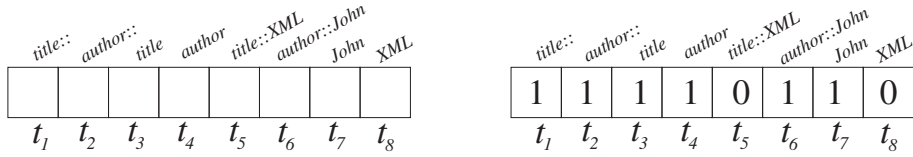


Fig. 8. Query bitmap for queries in Table 1.

We compare ELCAStream with a baseline that implements a simple ELCA strategy. Like ELCAStream, this baseline uses SLCA nodes as a starting point for evaluating ELCA nodes. However, its SLCA evaluation is based on the CKStream implementation, i.e., likewise it uses a bitmap for query evaluation and, therefore, incorporates none of the ELCAStream optimizations.

5.3. Response time performance

We performed three experiments with streams containing documents from our XML datasets, each experiment focusing on a different aspect. The first experiment analyzes the impact of increasing the number of queries. These queries include only

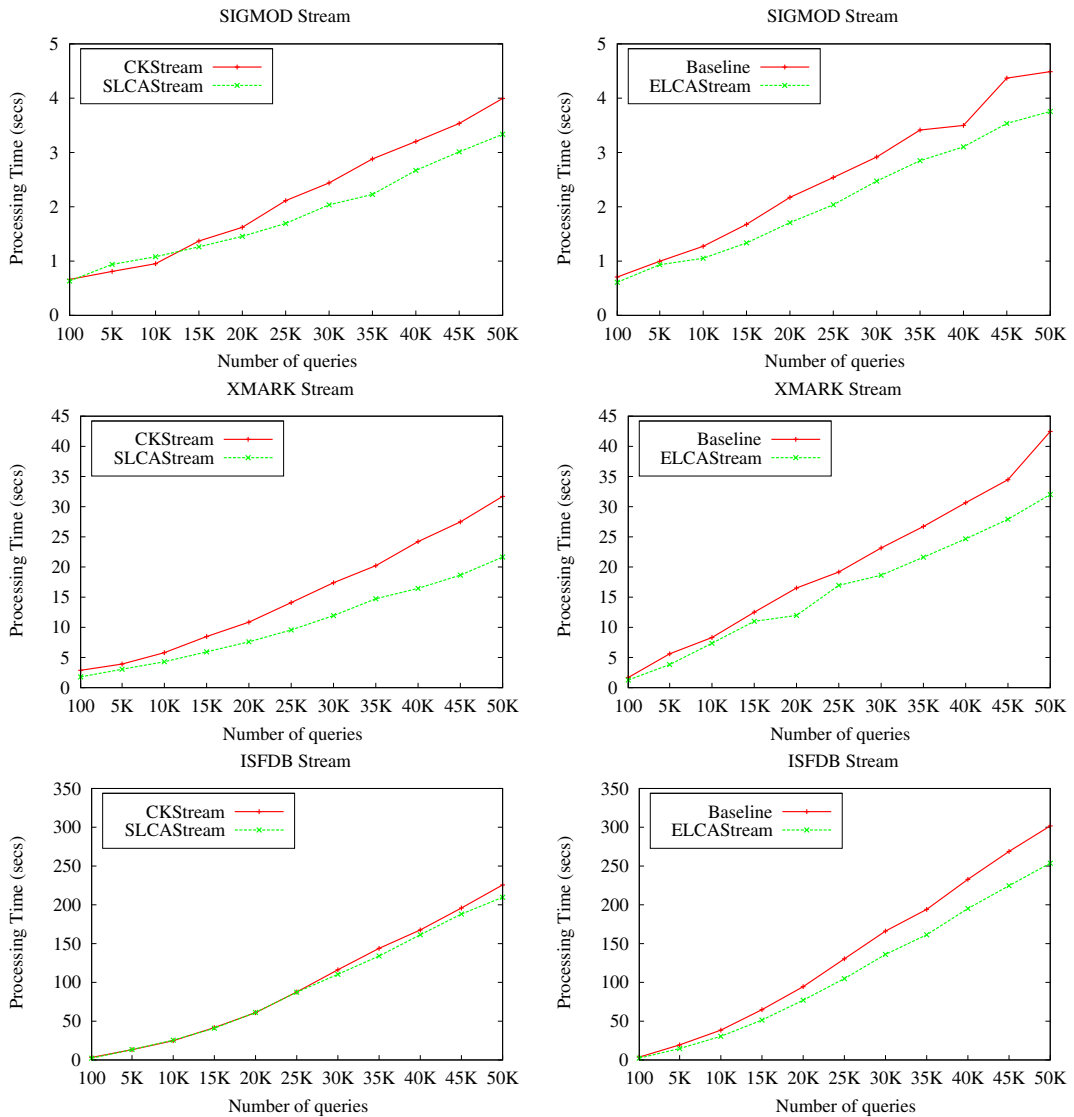


Fig. 9. First performance experiment: response times for the SLCAStream (left) and the ELCAStream (right) algorithms in the three datasets.

pure keywords, i.e., those that refer to the contents of the XML nodes. The second experiment analyzes how labels (*structural terms*) impact the results of keyword-based queries. The third experiment evaluates how our algorithms scale with the number of distinct terms in the queries.

The experiments consider only the actual time spent by the callback functions, thus excluding the time spent for creating the query indexes, which occurs before query processing starts. For simplicity, we show memory usage results only for the third experiment because they reflect the same behavior of the other two experiments. When measuring memory usage, we considered the average memory used while processing each XML document, including memory used by all index structures.

5.3.1. Varying the number of queries

The first experiment aims at analyzing the response time as the number of keyword-based queries increases. It considers random queries, each one with up to four terms. Each query term has the form $::k$. We discuss the results separately for each algorithm.

Fig. 9 (left) compares the time spent by SLCAStream and CKStream, our SLCA baseline, over each dataset. The time curves show a better performance of SLCAStream in the three datasets due to a more efficient SLCA semantics implementation. For 50,000 queries, SLCAStream reduced CKStream time up to 32% on the XMARK dataset. Fig. 9 (right) compares time spent by ELCAStream and its baseline over each dataset. The time curves show a clear advantage of ELCAStream over the baseline in the

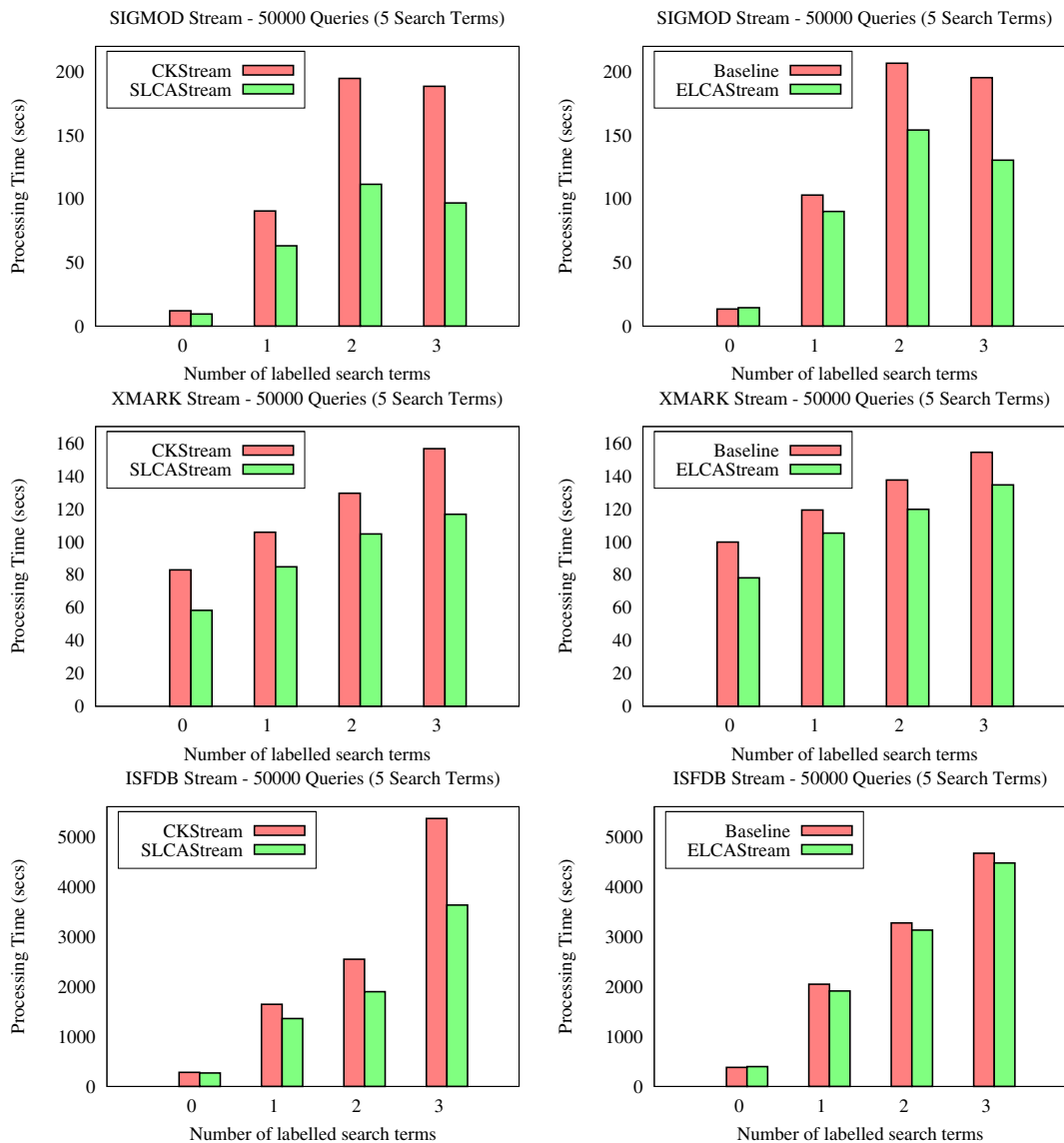


Fig. 10. Second performance experiment: response times for the SLCA (left) and the ELCA (right) algorithms in the three datasets.

three datasets, since it requires no bitmap and stack operations, and evaluates only plausible queries. For the same queries, ELCAStream reduced the time spent by the baseline up to 25% on the XMARK dataset.

5.3.2. Searching with structural constraints

The second experiment analyzes the impact of using structural terms, such as $\ell::k$, in the queries. The experiment considers 50,000 queries with five query terms each and varies the number of structural query terms from 0 to 3. Similarly to the first experiment, we discuss the results separately for each algorithm.

Fig. 10 (left) presents the time comparison of SLCAStream and CKStream over each dataset. The results show a significant advantage of SLCAStream in all datasets due to our optimization techniques. For 50,000 queries with three labels and two keywords, SLCAStream reduced CKStream time up to 49% on the SIGMOD dataset. Fig. 10 (right) compares the time spent by ELCAStream and its baseline over each dataset. Likewise, these results also show a clear advantage of ELCAStream for the same reasons pointed out in the previous experiment. For the same queries, ELCAStream reduced the time spent by our baseline up to 33% on the SIGMOD dataset.

For both algorithms, Fig. 10 also shows a performance degradation when the number of structural terms increases. In this experiment, structural terms are present in each of the 50,000 queries to stress the algorithms' performance, thus representing the worst case scenario. However, in real world, we expect little use of such structural terms because users generally have a minimum or no knowledge of the document structure. Thus, in this scenario, the performance is quite acceptable.

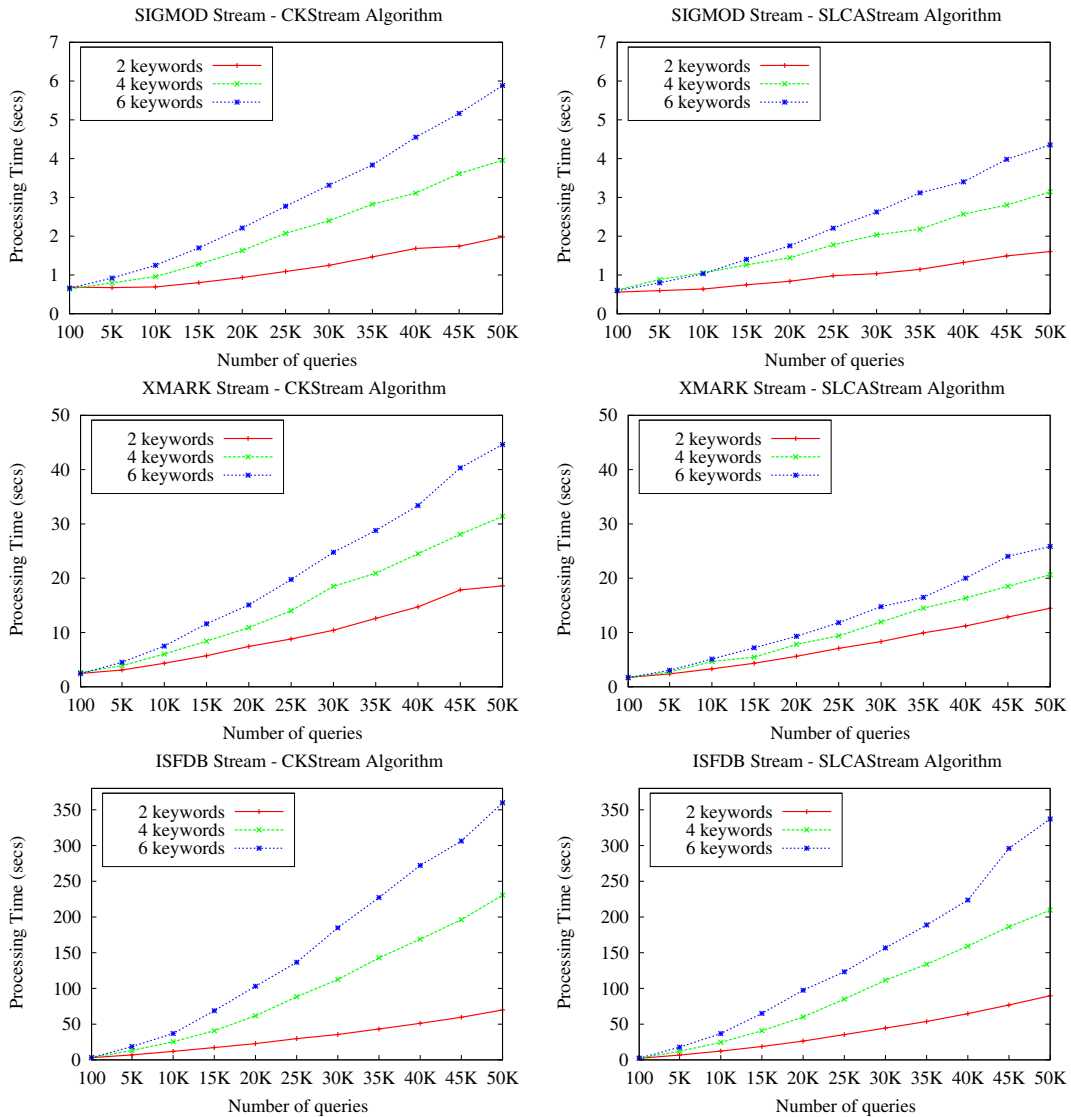


Fig. 11. Third performance experiment: response times for the CKStream(left) and SLCAStream (right) algorithms in the three datasets.

5.3.3. Varying the number of terms

This experiment evaluates the scalability of our algorithms by increasing the number of distinct terms in the queries. It analyzes the impact of using 2, 4 and 6 terms in queries of the form $::k$. Notice that in general the number of terms in these queries surpass 2.91, which is the average number of terms found in keyword queries submitted to Web databases according to Coffman and Weaver [6].

Fig. 11 presents SLCAStream and CKStream results on the left and right graphs respectively. As expected, increasing the number of terms affects the performance of the algorithms. However, SLCAStream performs faster in all datasets. For 50,000 queries with six keywords, SLCAStream reduced CKStream time up to 42% on the XMARK dataset.

Likewise, Fig. 12 presents the results for the baseline and ELCAStream on the left and right graphs respectively. Again, as expected, increasing the number of terms affects the performance. However, ELCAStream performs better than the baseline due to its optimization techniques and because it requires no bitmap processing. For the same queries, ELCAStream reduced the time spent by the baseline up to 24% on the XMARK dataset.

5.4. Memory usage performance

Regarding memory usage performance, we present results only for the third experiment, considering 50,000 queries with 4 and 6 query terms, since they follow the same pattern of the other two. Fig. 13 (left) shows that SLCAStream spends much

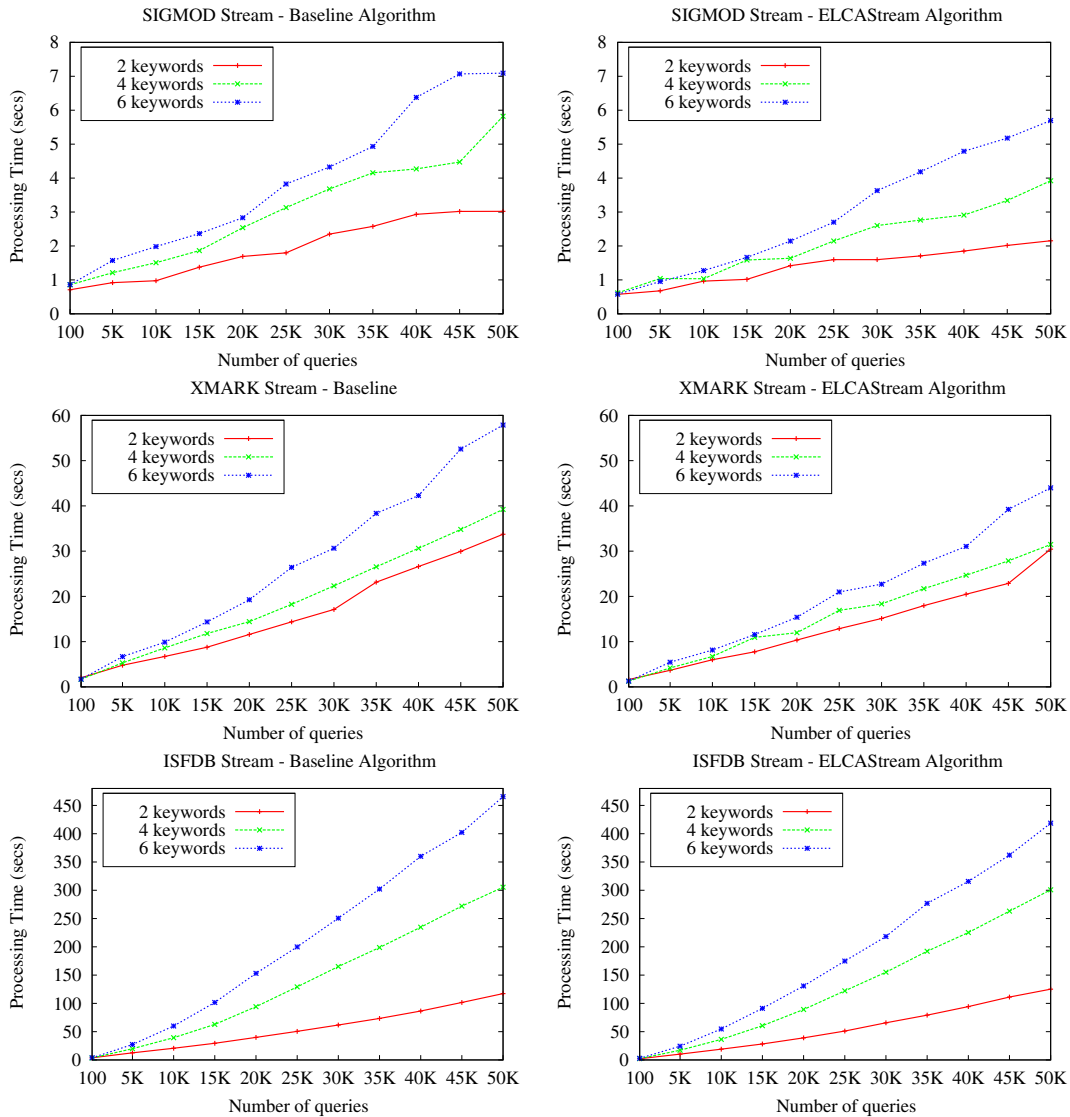


Fig. 12. Third performance experiment: response times for the baseline (left) and ELCAStream (right) algorithms in the three datasets.

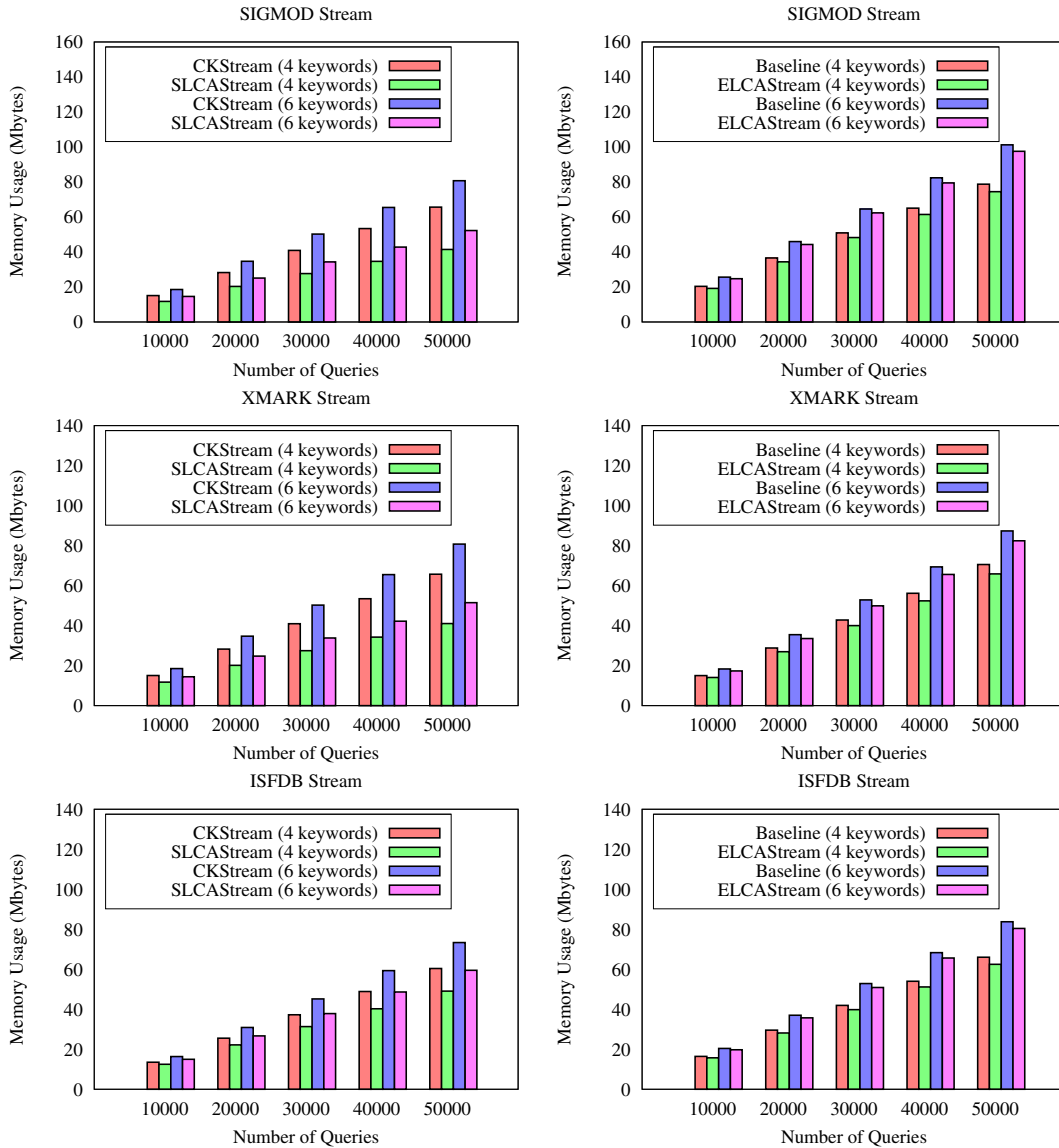


Fig. 13. Memory usage experiment: memory consumption for the SLCA (left) and ELCA (right) algorithms in the three datasets.

less memory than CKStream. As previously mentioned, CKStream uses a very complex query bitmap for SLCA evaluation, which becomes quite large as the number of queries increases. Our SLCAStream implementation adopts similar data structures, but does not use such a bitmap. Thus, for queries with six labels, when compared with CKStream, SLCAStream reduced memory usage up to 36% on the XMARK dataset. Fig. 13 (right) shows that ELCAStream has a performance similar to its baseline, but with less memory consumption for not using a query bitmap. For the same 50,000 queries, when compared with its baseline, ELCAStream reduced memory usage spent by up to 6% on the XMARK dataset.

We notice that, although ELCAStream uses inverted lists for processing the queries, which increases memory consumption when compared with SLCAStream, this consumption is still acceptable, showing that its implementation is feasible and causes no relevant impact in terms of time performance, as discussed in Section 5.3.

5.5. Performance evaluation summary

Each one of our experiments used different types of query on three datasets. On average, the number of evaluated queries achieved almost 30% of the total submitted. Particularly, on the ISFDB dataset, this number reached 80% because it is the largest one and, therefore, the number of matched terms was much larger. In the first experiment, SLCAStream, compared with its baseline, reached up to 32% reduction in response time on the XMARK dataset. In the second, this reduction reached up to 43%,

but on the SIGMOD dataset. In the third experiment, the reduction reached up to 42% on the XMARK dataset. Regarding the ELCA algorithms, we performed the same three experiments comparing ELCAStream with its baseline, which incorporates none of the ELCAStream optimizations. In this comparison, the reduction in response time were up to 25% in first experiment on XMARK, 33% in the second experiment on SIGMOD and 24% in the third experiment on XMARK, all in favor of ELCAStream.

Regarding memory usage performance, compared with CKStream, SLCAStream reduced its memory usage by up to 36% on the XMARK dataset. On the other hand, when compared with its baseline, ELCAStream reduced memory usage by up to only 6% on the same dataset, since both algorithms are based on data structures that use more memory.

Thus, compared with CKStream, SLCAStream showed the best performance on small XML documents (SIGMODR and XMARK datasets) due to the small number of nodes processed. For larger documents, such as the ones in the ISFDB datasets, both algorithms presented high response times due to the larger number of nodes and queries processed. For the same reasons, compared with its baseline, ELCAStream presented the best gains on small XML documents and high response times on the ISFDB dataset. However, the performance gains were smaller since our ELCA baseline and ELCAStream work with inverted lists, which present high memory consumption.

It is worth mentioning that, according to Barbosa et al. [3], the average size of an XML document in real Web scenarios is only 4 KB, while the maximum size reaches 500 KB. These figures are also typical for XML data dissemination settings [32]. Thus, our response time experiments show that SLCAStream and ELCAStream are efficient alternatives for processing keyword-based queries over XML streams.

6. Conclusions

In this article we proposed two new, efficient LCA-based algorithms for processing multiple keyword-based queries over XML streams: SLCAStream and ELCAStream. SLCAStream provides a new approach to SLCA evaluation that avoids the usual bitmap processing strategy, thus significantly improving response time and memory consumption when compared to CKStream [21], the state-of-the-art SLCA algorithm. ELCAStream builds over SLCAStream for the ELCA semantics and, therefore, is the first ELCA-based algorithm for processing multiple queries over XML streams. Both algorithms exploit stream processing properties based on the LCA semantics and introduce optimization strategies that improve their overall performance. To evaluate the proposed algorithms, we conducted extensive experiments that analyze their performance and scalability, thus showing that both are efficient alternatives for processing keyword-based queries over XML streams. In fact, both algorithms have linear time and space complexity, as shown by our complexity analysis.

As future work, we plan to extend our algorithms with ranking strategies [9] and then evaluate their accuracy. Additionally, we plan to develop a complete parallel framework for processing multiple queries over XML streams based on our algorithms.

Acknowledgment

This work was partially funded by projects InWeb (grant MCT/CNPq 573871/2008-6) and MASWeb (grant FAPEMIG/PRONEX APQ-01400-14), and by the authors' individual grants from CNPq and FAPEMIG.

References

- [1] J. Widom, B. Babcock, Models and issues in data stream systems, *Proceedings of the 21st ACM Symposium on Principles of Database Systems*, ACM, Madison, USA, 2002, pp. 1–16.
- [2] Z. Bao, T.W. Ling, B. Chen, J. Lu, Effective XML keyword search with relevance oriented ranking, *Proceedings of the 25th International Conference on Data Engineering*, Shanghai, China, 2009, pp. 517–528.
- [3] D. Barbosa, L. Mignet, P. Veltri, Studying the XML web gathering statistics from an XML sample, *World Wide Web* 9 (2) (2006) 187–212.
- [4] E.G. Barros, M.M. Moro, A.H.F. Laender, An evaluation study of search algorithms for XML streams, *J. Inf. Data Manag.* 1 (3) (2010) 487–502.
- [5] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, S. Zdonik, Monitoring streams: a new class of data management applications, *Proceedings of the 28th International Conference on Very Large Data Bases*, 2002, pp. 215–226. Hong Kong, China.
- [6] J. Coffman, A.C. Weaver, A Framework for Evaluating Database Keyword Search Strategies, *ACM*, New York, NY, USA, 2010, 729–738.
- [7] G. Costa, G. Manco, R. Ortale, E. Ritacco, Hierarchical clustering of XML documents focused on structural components, *Data Knowl. Eng.* 84 (March 2013) 26–46.
- [8] M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in ad hoc time windows, *Data Knowl. Eng.* 87 (2013) 66–90. September.
- [9] A. Dimitriou, D. Theodoratos, T. Sellis, Top-k-size keyword search on tree structured data, *Inf. Syst.* 47 (2015) 178–193.
- [10] N.G. Duffield, M. Grossglauser, Trajectory sampling for direct traffic observation, *ACM Trans. Networking* 9 (3) (2001) 280–292.
- [11] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M.J. Carey, A. Sundararajan, G. Agrawal, The BEA/XQRL streaming XQuery processor, *Proceedings of the 29th International Conference on Very Large Data Bases—Volume 29*, Pages 997–1008, Berlin, Germany, 2003.
- [12] M. Franceschet, XPathMark: an XPath benchmark for the XMark generated data, *Proceedings of the Third International Conference on Database and XML Technologies*, Springer-Verlag, Berlin Heidelberg, 2005, pp. 129–143.
- [13] L. Golab, M.T. Özsu, Issues in data stream management, *SIGMOD Rec.* 32 (2) (2003) 5–14.
- [14] G. Gou, R. Chirkova, Efficiently querying large XML data repositories: a survey, *IEEE Trans. Knowl. Data Eng.* 19 (10) (2007) 1381–1403.
- [15] S. Grijzenhout, M. Marx, The quality of the XML web, *Web semantics: science, services and agents on the world wide web* 19 (2013) 59–68. Mar.
- [16] L. Guo, F. Shao, C. Botev, J. Shanmugasundaram, XRANK: ranked keyword search over XML documents, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, Pages 16–27, San Diego, USA, 2003.
- [17] A.K. Gupta, D. Suciu, Stream processing of XPath queries with predicates, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2003, pp. 419–430. San Diego, USA.
- [18] S.C. Haw, C.S. Lee, Data storage practices and query processing in XML databases: a survey, *Knowl.-Based Syst.* 24 (8) (2011) 1317–1340.
- [19] V. Hristidis, N. Koudas, Y. Papakonstantinou, D. Srivastava, Keyword proximity search in XML trees, *IEEE Trans. Knowl. Data Eng.* 18 (4) (2006) 525–539.

- [20] C.-L. Hu, C.-K. Chou, RSS Watchdog: an instant event monitor on real online news streams, *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, 2009, pp. 2097–2098. Hong Kong, China.
- [21] F. Hummel, A.S. da Silva, M.M. Moro, A.H.F. Laender, Multiple keyword-based queries over XML streams, *Proceedings of the 20th ACM Conference on Information and Knowledge Management*, 2011, pp. 1577–1582. Glasgow, UK.
- [22] S.K. Izadi, M.S. Haghighi, T. Härder, S³ Processing tree-pattern XML queries with all logical operators, *Data Knowl. Eng.* 72 (2012) 31–62. February.
- [23] C. Koch, S. Scherzinger, N. Schweikardt, B. Stegmaier, FluXQuery: an optimizing XQuery processor for streaming XML data, *Proceedings of the 30th International Conference on Very Large Data Bases*, Toronto, Canada, 2004.
- [24] A.H.F. Laender, M.M. Moro, C. Nascimento, P. Martins, An X-ray on web-available XML schemas, *ACM SIGMOD Rec.* 38 (1) (2009) 37–42.
- [25] T.N. Le, H. Wu, T.W. Ling, L. Li, J. Lu, From structure-based to semantics-based: towards effective XML keyword search, 2013, pp. 356–371. Hong Kong, China.
- [26] H.H. Lee, W.S. Lee, Selectivity-sensitive shared evaluation of multiple continuous XPath queries over XML streams, *Inform. Sci.* 179 (12) (2009) 1984–2001.
- [27] G. Li, J. Feng, J. Wang, L.Z.h.o.u. KEMB, A keyword-based XML message broker, *IEEE Knowl. Data Eng.* 23 (7) (2011) 1035–1049.
- [28] Z. Liu, Y. Chen, Processing keyword search on XML: a survey, *World Wide Web* 14 (5–6) (2011) 671–707.
- [29] S. Madden, M.J. Franklin, Fjording the stream: an architecture for queries over streaming sensor data, *Proceedings of the 18th International Conference on Data Engineering*, 2002, pp. 555–566. San Jose, CA, USA.
- [30] D. McCafferty, Brave, new social world, *Commun. ACM* 54 (7) (2011) 19–21. July.
- [31] M.P. Mendell, H. Nasgaard, E. Bouillet, M. Hirzel, B. Gedik, Extending a general-purpose streaming system for XML, *Proceedings of the 15th International Conference on Extending Database Technology*, 2012, pp. 534–539. Berlin, Germany.
- [32] I. Miliaraki, Z. Kaoudi, M. Koubarakis, XML data dissemination using automata on top of structured overlay networks, *Proceedings of the 17th International Conference on World Wide Web*, 2008, pp. 865–874. Beijing China.
- [33] Early profile pruning on XML-aware publish–subscribe systems, M.M. Moro, P. Bakalov, V.J. Tsotras, *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007, pp. 866–877. Vienna, Austria.
- [34] M.M. Moro, V. Braganholo, C.F. Dorneles, D. Duarte, R. Galante, R.S.Mello, XML, Some papers in a haystack, *SIGMOD Record* 38 (2009) 29–34.
- [35] M. Onizuka, Processing XPath queries with forward and downward axes over XML streams, *Proceedings of the 13th International Conference on Extending Database Technology*, 2010, pp. 27–38. Lausanne, Switzerland.
- [36] H.K. Park, S.J. Shin, S.H. Na, W.S. Lee, M-COPE: a multiple continuous query processing engine, *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, 2009, pp. 2065–2066. Hong Kong, China.
- [37] J.P. Park, C.S. Park, Y.D.C.h.u.n.g. Encoding, Lineage an efficient wireless XML streaming supporting twig pattern queries, *IEEE Trans. Knowl. Data Eng.* 25 (7) (2013) 1559–1573.
- [38] Z. Vagena, L.S. Colby, F. Özcan, A. Balmin, Q. Li, On the effectiveness of flexible querying heuristics for XML data, In *Proceedings of the 3rd International Workshop on Database Technologies for Handling XML Information on the Web*, 2007, pp. 77–91. Vienna, Austria.
- [39] Z. Vagena, M.M. Moro, Semantic search over XML document streams, *Proceedings of the 3rd International Workshop on Database Technologies for Handling XML Information on the Web*, Nantes, France, 2008.
- [40] Z. Vagena, M.M. Moro, V.J. Tsotras, RoXSum: Leveraging data aggregation and batch processing for XML routing, *Proceedings of the 23rd International Conference on Data Engineering*, 2007, pp. 1466–1470. Istanbul, Turkey.
- [41] X. Wu, D. Theodoratos, A survey on XML streaming evaluation techniques, *VLDB J.* 22 (2) (2013) 177–202.
- [42] Y. Xu, J. Guan, F. Li, S. Zhou, Scalable continual top-k keyword search in relational databases, *Data Knowl. Eng.* 86 (2013) 206–223.
- [43] Y. Xu, Y. Papakonstantinou, Efficient keyword search for smallest LCAs in XML databases, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2005, pp. 537–538. Goa, India.
- [44] Y. Xu, Y. Papakonstantinou, Efficient LCA-based keyword search in XML data, *Proceedings of the 11th International Conference on Extending Database Technology*, 2008, pp. 535–546. Nantes, France.
- [45] W. Yang, F. Fang, N. Li, J.L.u. XKFitler, A keyword filter on XML stream, *Int. J. Inf. Retr. Res.* 1 (1) (2011) 1–18.
- [46] J. Zhou, Z. Bao, W. Wang, T.W. Ling, Z. Chen, X. Lin, J. Guo, Fast SLCA and ELCA computation for XML keyword queries based on set intersection, *Proceedings of the IEEE 28th International Conference on Data Engineering*, 2012, pp. 905–916. Washington, DC, USA.
- [47] J. Zhou, X. Zhao, W. Wang, Z. Chen, J.X. Yu, Top–down keyword query processing on XML data, *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management*, 2013, pp. 2225–2230. San Francisco, USA.
- [48] R. Zhou, C. Liu, J. Li, Fast ELCA computation for keyword queries on XML data, *Proceedings of the 13th International Conference on Extending Database Technology*, 2010, pp. 549–560. Lausanne, Switzerland.
- [49] R. Zhou, C. Liu, J. Li, J.X. Yu, ELCA evaluation for keyword search on probabilistic XML data, *World Wide Web* 16 (2) (2013) 171–193.