PROCESSAMENTO DE CONSULTAS BASEADAS EM PALAVRAS-CHAVE SOBRE FLUXOS XML

EVANDRINO GOMES BARROS

PROCESSAMENTO DE CONSULTAS BASEADAS EM PALAVRAS-CHAVE SOBRE FLUXOS XML

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Orientador: Prof.Alberto Henrique Frade Laender
Coorientador: Prof. Mirella Moura Moro

Belo Horizonte

Novembro de 2012

EVANDRINO GOMES BARROS

KEYWORD-BASED QUERY PROCESSING OVER XML STREAMS

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Prof.Alberto Henrique Frade Laender
Co-Advisor: Prof. Mirella Moura Moro

Belo Horizonte
November 2012

Barros, Evandrino Gomes.

B277k Keyword-based Query Processing over XML Streams / Evandrino Gomes Barros — Belo Horizonte, 2012

xxvii, 119 f. : il. ; 29cm

Tese (doutorado) — Universidade Federal de Minas Gerais - Departamento de Ciência da Computação

Orientador: Alberto Henrique Frade Laender

1. Computação - Teses. 2. XML (Linguagem de marcação de documento) - Teses. I. Orientador. II. Título.

519.6*33(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Keyword-based query processing over XML streams
(Processamento de consultas baseadas em palavras-chave sobre fluxos XML)

EVANDRINO GOMES BARROS

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]
Prof. Alberto Henrique Frade Laender - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]
Profa. Mirella Moura Moro - Coorientadora
Departamento de Ciência da Computação - UFMG

[illegible]
Prof. Carlos Alberto Heuser
Departamento de Informática - UFRGS

[illegible]
Prof. Clodoveu Augusto Davis Júnior
Departamento de Ciência da Computação - UFMG

[illegible]
Prof. Marco Antônio Casanova
Departamento de Informática – PUC/RJ

[illegible]
Profa. Marta Lima de Queiros Mattoso
Programa de Engenharia de Sistemas - COPPE/UFRJ

Belo Horizonte, 22 de novembro de 2012.

I dedicate this PhD thesis to my wife, Fernanda, my son Gabriel, and my born-to-be son Mateus. You are the inspiration and reason of all this.

Acknowledgments

This PhD thesis is an old dream that has become possible with the support of many people. I would like to thank my advisor Prof. Alberto Laender for having trusted me and accepted my thesis proposal. His constant support and example of dedication inspired me to achieve my best. I would also like to thank my coadvisor Dr. Mirella Moro as well as Dr. Altigran Silva from UFAM for their valuable contributions and support during my PhD. I am very grateful to my wife, Fernanda, for her patience, companionship and love throughout my PhD. I would like to thank my son, Gabriel, whose *joie de vivre* brought me more enthusiasm and motivation, important feelings for concluding a PhD thesis. I also thank my born-to-be son, Mateus, who was conceived close to my PhD conclusion. He brought me more hope and energy in my final sprint. I also thank my parents, brothers and relatives for their support and encouragement in all my dreams, specially my PhD thesis. A special thanks goes to my friends for providing the support and friendship that I needed. I also thank my mother-in-law, Wania, for her prayers. I would like to thank the administrative staff of the UFMG Department of Computer Science, particularly Renata, Maristela, Sheila and Gilmara, for their warm and gentle assistance throughout my PhD. I have had the pleasure of interacting with great colleagues at the UFMG Database Laboratory, some of which became precious friends. Thank you all my colleagues. Finally, I would like to express my gratitude to CEFET-MG, specially to the Department of Computing for providing me the conditions for concluding my PhD thesis. Its faculty, students and administrative staff have also contributed for helping me to face my PhD challenges.

"I don't know anything with certainty,
but seeing the stars makes me dream."
(Vincent van Gogh)

Abstract

XML streams have become a relevant research topic due to the widespread use of applications such as online news, RSS feeds, and dissemination systems. Such streams must be processed rapidly and without retention. Retaining streams could cause data loss due to the large data traffic in continuous processing. This context becomes more complex when thousands of queries must be evaluated simultaneously. Different approaches explore simultaneous multiple query processing. However, they are based on structured languages such as XPath and XQuery, which require knowledge of their syntax and the data structure to formulate queries. Keyword-based language is a usual approach to submit queries informally, because they require minimal or no schema knowledge to formulate queries. Some approaches focus on improving search performance, but only in archived XML documents. More recent techniques have focused on keyword-based search algorithms for XML streams, but they only run one query at a time. Most of the keyword-based algorithms consider the lowest common ancestor (LCA) semantics. 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 since the same content can occur at different levels, such as keywords that correspond to XML labels occurring in different schema elements. Thus, ELCA is considered one of the most effective semantics because it returns a larger number of results. However, previous approaches do not support the major challenges in the new stream application scenarios. These challenges involve (i) the efficient processing of thousands of user queries over XML streams and (ii) the relief of users from knowing the source schemas when accessing ambiguous or heterogeneous data sources. To address these challenges, in this thesis, we propose new algorithms for processing multiple keyword queries over XML streams. The algorithms explore stream processing properties based on the LCA semantics and provide optimized methods to improve the overall performance. In addition, we propose strategies for ranking query results over XML streams. A comprehensive set of experiments thoroughly evaluates several aspects related to performance, scalability and accuracy of our algorithms, showing that our algorithms are efficient alternatives to search services over XML streams.

Resumo

Fluxos de dados XMLtornaram-se um relevante tema de pesquisa devido ao uso generalizado de aplicações Web em tempo real, tais como notícias on-line e RSS *feeds*. Esses fluxos devem ser processados rapidamente e sem retenção. Aplicações sobre fluxos XML tornam-se complexas quando milhares de consultas devem ser processadas simultaneamente. Diferentes abordagens exploram o processamento simultâneo de consultas sobre fluxos XML. No entanto, elas são baseadas em linguagens estruturadas, tais como XPath e XQuery. Essas linguagens exigem conhecimento de suas sintaxes e do esquema de dados envolvido para a formulação de consultas. Palavras-chave são uma alternativa informal para submeter consultas a aplicações sobre fluxos XML, pois requerem conhecimento mínimo do esquema de dados. Abordagens existentes, baseadas em palavras-chave, se concentram em melhorar o desempenho do processamento de consultas, mas geralmente envolvem documentos XML arquivados e estruturas auxiliares, tais como índices. Abordagens mais recentes concentram-se em algoritmos para palavras-chave sobre fluxos XML ou processam uma única consulta por vez. A maioria dos algoritmos para processamento de consultas baseadas em palavras-chave considera a semântica do menor ancestral comum (LCA - *Lowest Common Ancestor*). Especificamente, o nó LCA de dois nós em uma árvore XML é o ancestral desses nós mais distante da raiz. Os algoritmos LCA mais populares são baseados nas semânticas SLCA (*Smallest LCA*) e ELCA (*Exclusive LCA*). ELCA lida com a ambiguidade que pode existir em um documento XML pois uma palavra-chave pode ocorrer em diferentes níveis. As abordagens anteriores não suportam os grandes desafios para os novos cenários das aplicações sobre fluxos XML que são: (i) o processamento eficiente de milhares de consultas e (ii) desconhecimento os esquemas de dados envolvidos. Por isso, propomos novos algoritmos de processamento de múltiplas consultas baseadas em palavras-chave sobre fluxos XML. Os algoritmos exploram propriedades do processamento de fluxos e utilizam técnicas para melhorar o desempenho do processamento. Além disso, propomos estratégias para o *ranking* dos resultados. Experimentos abrangentes avaliam desempenho, escalabilidade e acurácia dos algoritmos e mostram que os mesmos são alternativas eficientes para serviços de consulta sobre fluxos XML.

List of Figures

List of Tables

List of Algorithms

Contents

Chapter 1

Introduction

XML has become the most successful and ubiquitous technology for data exchange on the Web [Wilde and Glushko, 2008]. This fact is corroborated by more than 7,000 scientific papers on XML available in DBLP[1] (early 2012), which address several aspects related to the XML technology. A recent trend on XML research has focused on XML streams [Wu and Theodoratos, 2012]. XML streams are characterized by data encapsulated in XML documents transmitted sequentially between origin and destination. XML streams have become a relevant research topic due to the widespread use of applications, such as online news, RSS (Really Simple Syndication) feeds and dissemination systems. These applications are increasing as Web users focus on them rather than on archived information. Such streams must be processed rapidly and without retention. Retaining streams could cause data loss or delay due to the continuous data traffic.

XML stream applications involve content search and filtering services based on queries. These applications become more complex when thousands of queries must be processed simultaneously. Moreover, XML stream applications process heterogeneous data sources with different schemas. Therefore, these applications must depend on simple query languages such as those based on keywords. Recent work has focused on efficient algorithms for processing keyword-based queries over XML streams [Barros et al., 2010, Hummel et al., 2011, Vagena and Moro, 2008, Wu and Theodoratos, 2012]. Once keyword-based queries over XML streams can be efficiently processed, a natural further step is to present the results to the user ordered by their relevance. Thus, another emerging research topic is related to dynamically ranking XML nodes returned by keyword-based queries.

In this thesis, we propose new algorithms for processing multiple keyword queries over XML streams. The algorithms explore stream processing properties based on tra-

[1]http://www.informatik.uni-trier.de/˜ley/db/

ditional keyword-based querying semantics and provide optimized methods to improve the overall performance. In addition, we propose strategies for ranking query results. A comprehensive set of experiments thoroughly evaluates several aspects related to performance, scalability and accuracy of our algorithms, showing that they are efficient alternatives to search services over XML streams.

Specifically, the first chapter of this thesis is organized as follows. Section 1.1 characterizes the data stream environment and introduces some typical applications. Section 1.2 presents the motivations of the thesis and Section 1.3 summarizes its major contributions. Finally, Section 1.4 describes the thesis organization.

1.1 Data Streams

Data streams are defined as encapsulated data flowing from source to destination without retention. They flow online and there is no control over the sequence of incoming data. Once an element of a data stream has been processed, it is discarded. It can not be recovered or archived, unless explicitly stored in memory, which is typically small relative to the size of the data stream [Babcock et al., 2002].

Several applications are based on data streams. Data streams are commonly generated when monitoring sensors in very complex equipments and installations such as airplanes and weather stations [Babcock et al., 2002]. Specifically, sensor monitoring applications use a large number of sensors distributed in the physical world which generate data streams that need to be grouped, monitored and analyzed [Carney et al., 2002, Madden and Franklin, 2002].

Another class of data stream applications is related to network traffic management systems. For example, the backbone network of an Internet Service Provider (ISP) monitors a variety of continuous data streams that may be characterized as unpredictable, arriving at a high rate and including both packet traces and network performance measurements [Duffield and Grossglauser, 2001]. Specifically, in ISP environments, providers connect millions of residential and business customers and involve several process tasks, such as filtering packets, traffic limitation and network performance analyses [Cortes et al., 2000].

Other applications that process large amounts of data streams, most of them XML streams, are Web social networks. In such an environment, millions of users share interests, opinions, likes and recent news. All this information provides an instantaneous snapshot of those networks that allows, for instance, the evaluation of social and market trends [Bigonha et al., 2012]. Facebook has just reached 1 billion[2] users and more than 70 language translations. Twitter reached 200 million tweets in multiple languages. Moreover, Facebook generates more daily traffic than any site in the world, except Google [McCafferty, 2011]. RSS feeds are another type of application that also involve large data stream usage. RSS is an XML feed format to publish frequently updated entities such as news headlines and blogs. RSS popularity has stimulated new XML stream applications, such as the RSS Watchdog system, which is capable of clustering news and monitoring instant events over multiple real and online news XML streams [Hu and Chou, 2009].

Recently, some distributed computing frameworks, such as System S [Gedik et al., 2008] and Watershed [Ramos et al., 2011], have been designed to process large data streams online and in real-time. System S, now called InfoSphere Streams, is a large-scale, high performance computing platform developed at IBM under the data stream paradigm. It can execute a large number of jobs in the form of data-flow graphs described in its special stream-application language SPADE (Stream Processing Application Declarative Engine) [Gedik et al., 2008]. Watershed is a distributed computing framework designed to support the analysis, online and real-time, of very large data streams. Data are obtained from streams by the its processing components and transformed, into other streams, thus creating large flows of information [Ramos et al., 2011].

All aforementioned applications depict the increasing and intense usage of data streams, emerging research topics, for instance, related to performance and accuracy of search services.

1.2 Motivation

Most query processing strategies on XML streams use structured query languages such as XPath[3] e XQuery[4] [Chan et al., 2002, Chen et al., 2006, Gou and Chirkova, 2007a, Green et al., 2004, Gupta and Suciu, 2003, Li et al., 2008, Olteanu, 2007, Onizuka, 2010, Ramanan, 2009, Schmidt et al., 2001]. However, interesting issues remain to be solved. One of those was identified by Vagena and Moro [2008] as *semantic search over XML streams*, i.e., keyword-based search over XML streams. This issue is motivated by three main problems raised when using structured XML query languages on stream environments: (i) these languages require knowledge

[2]http://www.businessweek.com/articles/2012-10-04/facebook-the-making-of-1-billion-users
[3]http://www.w3.org/TR/xpath/
[4]http://www.w3.org/TR/xquery/

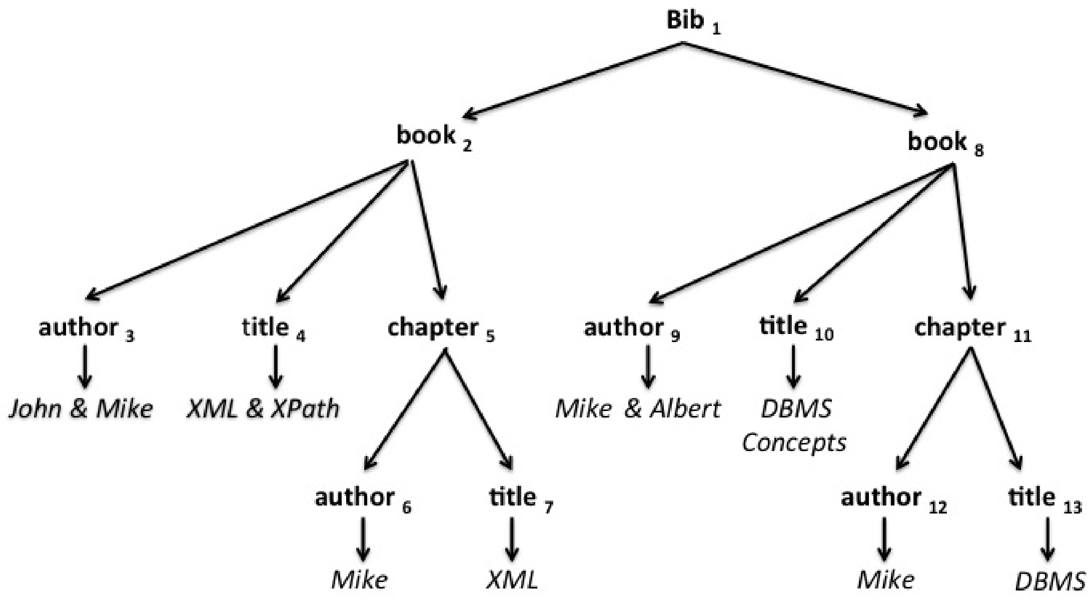


Figure 1.1. XML document example: books, theirs chapters e authors.

of the data structure to formulate meaningful queries; (ii) Web XML documents have large amounts of plain text, which makes it hard for processing value-based predicates with exact semantics; and (iii) these two problems are aggravated when we consider the Web scale, a distributed system with different data sources, since multiple data sources require structured query rewriting.

Keyword-based languages provide a traditional approach to submit queries informally, as they require minimal or no schema knowledge to formulate queries [Barros et al., 2010, Hummel et al., 2011, Vagena and Moro, 2008, Xu and Papakonstantinou, 2005, Xu and Papakonstantinou, 2008, Zhou et al., 2010]. Some proposed approaches focus on improving search performance, but only in archived XML documents [Xu and Papakonstantinou, 2008, Zhou et al., 2010]. More recent approaches have focused on keyword-based search algorithms for XML Streams [Hummel et al., 2011, Vagena and Moro, 2008]. However, these approaches only consider a specific search semantics or work with a single processing strategy based on bitmaps.

Most keyword-based algorithms consider the lowest common ancestor (LCA) semantics [Hummel et al., 2011, Xu and Papakonstantinou, 2005, Vagena and Moro, 2008, Xu and Papakonstantinou, 2008, Zhou et al., 2010]. Specifically, the LCA of two nodes u and v in an XML tree is the common ancestor of u and v located farthest from the root. For example, consider the keyword-based query q=$\{author, title\}$ and the XML document in Figure 1.1, which represents bibliographic data. In this figure, $book_2$ is an LCA result for query q, because it is the lowest ancestor for nodes $author_3$ and $title_4$ that match the keywords *author* and *title*. Similarly, $chapter_5$, $book_8$ and $chapter_{11}$ are also LCA nodes for q.

The most popular LCA-based algorithms use the smallest LCA (SLCA) and the exclusive LCA (ELCA) semantics. According to the SLCA semantics, an SLCA node has no LCA descendants. Thus, an SLCA node is the smallest lowest common ancestor node for a query. For example, consider again the XML document in Figure 1.1 and the query q=$\{author, title\}$. Only $chapter_5$ and $chapter_{11}$ are SLCA because they satisfy q and have no LCA descendants for q. The nodes $chapter_5$ and $chapter_{11}$ are also ELCA because they have exclusive occurrences for the q keywords. Likewise $book_2$ and $book_8$ are also ELCA results for query q. Specifically, the ELCA semantics addresses the ambiguity that might exist in XML data. The same content can occur at different levels, such as keywords corresponding to XML labels occurring in different schema elements [Bao et al., 2009]. Thus, ELCA is considered one of the most effective semantics because it returns a larger number of results [Zhou et al., 2010]. Chapter 3 formally defines the LCA and SLCA semantics while Chapter 5 defines the ELCA semantics.

XML streams must be processed rapidly and without retention. Retaining streams can cause data loss due to a large data traffic being continuously processed. This scenario becomes more complex when thousands of queries must be processed against a large number of documents flowing on the Web. Processing individually each query against each incoming document is an inefficient or unfeasible approach. If we evaluate individually each query per document, we need to locally store all incoming documents. This approach is inefficient since a same document must be repeatedly parsed, thus consuming a large storage space. Even creating an index to speed up the evaluation of queries, such an index would require local storage space. An ideal solution would be to simultaneously process multiple queries over one single transversal of each document on the stream. Different approaches explore simultaneous multiple query processing. However, they are based on structured languages such as XPath and SQL [Lee and Lee, 2009, Moro et al., 2007, Park et al., 2009], which require knowledge of their syntax as well as of the underlying data schemas to properly formulate queries. Recent approaches have focused on keyword-based algorithms for searching over XML Streams, but running one query at a time [Barros et al., 2010, Vagena and Moro, 2008]. To the best of our knowledge, Hummel et al. [2011] have been the first to address the problem of processing *multiple* keyword-based queries over XML streams. However, they only consider the specific LCA-based search semantics.

Once keyword-based queries over XML streams can be efficiently processed, a natural further step is to present the results to the user ordered by their relevance. Thus, another emerging research topic is related to dynamically ranking XML nodes returned by keyword-based queries. Currently, keyword-based search algorithms that provide ranked results use storage structures, such as inverted lists or indexes [Bao et al., 2009, Cohen et al., 2003, Guo et al., 2003, Li et al., 2010, Liu and Chen, 2008, Zhou et al., 2010]. Hence, these algorithms are unsuitable for the stream environment under consideration.

Considering the current XML stream applications scenario, we identify the following major challenges:

1. Efficient processing of thousands of user queries over XML streams. The goal is to addresses the problem of processing multiple queries over XML streams by developing news algorithms that reduce response time and save memory.

2. Freedom from knowing the source schemas to fully access ambiguous or heterogeneous data sources. The goal is to adopt semantics that relieve users from knowing structured query languages and the source schemas when accessing ambiguous or heterogeneous data sources.

3. Relevance ranking of query results over XML streams. The goal is to propose efficient and accurate strategies for ranking query results over XML streams.

1.3 Contributions

To address the previous challenges, this thesis presents several contributions which are summarized in the following subsections.

1.3.1 Query Evaluation Algorithms

To address the first two challenges, we have proposed five LCA-based algorithms for efficiently processing multiple keyword-based queries over XML streams. Three of them (BStream and MKStream for SLCA, and ELCABStream for ELCA) use traditional bitmaps for processing queries over data streams. The other two algorithms incorporate unexplored LCA stream processing properties, one based on the SLCA semantics (SLCAStream) and the other on the ELCA semantics (ELCAStream). In addition, our algorithms also rely on parsing stacks and query indexes specially designed to allow simultaneous matching of terms from different queries. Specifically, they can work with single or multiple parsing stacks.

Table 1.1 presents the characteristics of the five proposed algorithms considering the following aspects: (i) the adopted semantics, (ii) the strategy employed for query evaluation and (iii) the processing mode (single or multiple stack).

BStream is the first algorithm and extends the SLCA-based algorithm proposed by Vagena and Moro [2008], which evaluates a single query per execution. BStream evaluates all submitted queries simultaneously by representing their terms in a single bitmap. MKStream extends BStream since it represents all query terms in a single compact bitmap, without query term repetition. Moreover, MKStream simultaneously uses multiple parsing stacks, thereby improving performance compared to existing SLCA algorithms [Barros et al., 2012a]. SLCAStream extends BStream by relying on LCA stream processing properties as a new approach for query evaluation instead of the traditional bitmap strategy [Barros et al., 2010, Hummel et al., 2011, Vagena and Moro, 2008, Zhou et al., 2010]. Thus, SLCAStream incorporates optimization strategies that improves its overall performance with respect to MKStream.

Regarding the ELCA semantics, ELCABStream is a basic ELCA-based implementation obtained from MKStream. However, it uses a single bitmap and a single parsing stack for query evaluation. ELCAStream enhances ELCABStream by using LCA stream processing properties for query evaluation instead of the traditional bitmap strategy. It also uses a single parsing stack for query evaluation.

Figure 1.2 summarizes the characteristics of the proposed algorithm. We conducted extensive experiments to analyze their performance and scalability. The results show that each new algorithm improves its predecessor in both performance and memory usage. Specifically, SLCAStream and ELCAStream are our main contributions and the most efficient alternatives for processing multiple keyword-based queries over XML streams according to their respective semantics [Barros et al., 2012b].

Algorithm		Algorithm Semantics		Query Evaluation Strategy		Processing Mode	
		SLCA	ELCA	Bitmap	LCA Stream Processing Properties	Single Stack	Multiple Stack
1	BStream	X		X		X	
2	MKStream	X		X			X
3	SLCAStream	X			X	X	
4	ELCABStream		X	X		X	
5	ELCAStream		X		X	X	

Table 1.1. Proposed algorithms for efficient processing multiple keyword-based.

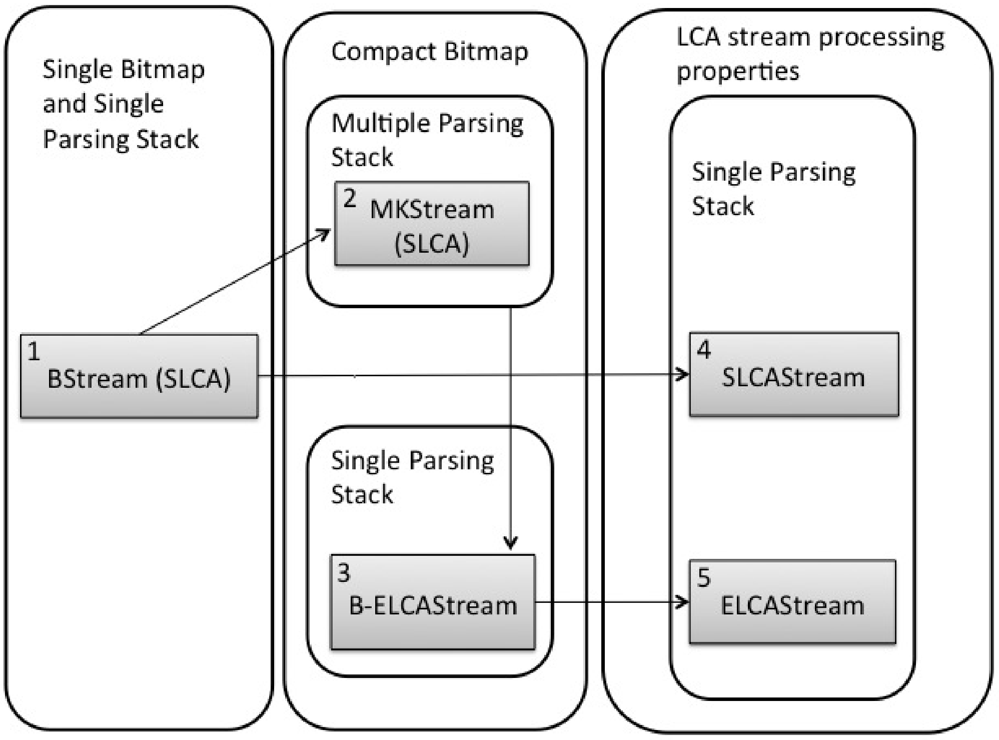


Figure 1.2. Proposed algorithms and their characteristics.

1.3.2 Stream Ranking

To address the third challenge presented in Section 1.3, we proposed the LCARank algorithm [Barros et al., 2010] and the SLCARank and StreamRank heuristics. LCARank provides a simple, efficient and effective strategy for ranking XML nodes. It combines the XRANK and SLCA stream search algorithms proposed by Vagena and Moro [2008] and prioritizes SLCA nodes as results, since XRANK includes SLCA and other LCA results. XRANK stream algorithm is a stream version for the keyword-based query algorithm proposed by Guo et al. [2003], which adheres to the LCA semantics and includes SLCA nodes. The original XRANK only addresses stored XML documents.

Considering that LCARank is a simple ranking algorithm, it is important to propose a fine-grained ranking heuristic for improving its results. Thus, we also propose the SLCARank heuristic and discuss its functioning for future implementation. The LCARank algorithm and the SLCARank heuristic focus on ranking XML nodes returned by a keyword query against a single XML document. They can be used on large XML streams, such as scientific data stored on large XML repositories [Green et al., 2004]. However, both algorithms would be more useful when applied to multiple query results obtained from a set of streams defined by a time slot or specific number of documents [Li et al., 2007, Singh et al., 2008, Sourlas et al., 2009]. Thus, for this specific scenario, we also proposed the StreamRank heuristic.

1.3.3 Accuracy Evaluation

To address the third challenge presented in Section 1.3, we evaluated the accuracy of the algorithms in terms of recall and precision.

First, we evaluated the accuracy of the SLCA and ELCA semantics, which are both adopted in our proposed algorithms. This evaluation showed that ELCA semantics significantly improves SLCA semantics [Barros et al., 2012b].

The second accuracy evaluation considered our LCARank algorithm, a simple, but effective ranking strategy for keyword-based query results over XML streams. As LCARank delivers SLCA results first and subsequently non-SLCA ones, we were interested in showing how this simple ranking strategy improves ranking when compared to the original XRANK algorithm. Therefore, we conducted a second evaluation which experimentally demonstrated that LCARank is a simple, effictive ranking strategy.

Both accuracy evaluations used keyword-based queries adapted from XPath queries specified for the XPathMark benchmark [Franceschet, 2005]. We considered XPath queries in order to provide a baseline for a consistent accuracy evaluation concerning recall and precision.

1.4 Thesis Organization

The remaining chapters of this thesis are organized as follows:

- **Chapter 2 – Querying XML Documents Using Keywords**. In this chapter, we review related work, emphasizing how this thesis contributes to the state-of-the-art. It is organized in three parts. First, we present algorithms for processing single keyword-based queries over XML documents, then algorithms for processing multiple keyword-based queries over XML documents and, finally, algorithms for ranking query results from XML documents.

- **Chapter 3 – SLCA Algorithms Based on Bitmaps.** In this chapter, we present the two SLCA-based algorithms we proposed for processing multiple keyword-based queries over XML streams. First, we introduce the SLCA semantics and the keyword-based query language we adopted. Then, we present the two algorithms we proposed that are based on traditional bitmaps, BStream and MKStream. Finally, we describe our experimental evaluation in which we compare the two algorithms with the current state-of-the-art algorithms proposed by Hummel et al. [2011]. Our experimental results show that MKStream reaches better memory and response time balance when compared to Hummel et al.'s algorithms [Barros et al., 2012a].

- **Chapter 4 – SLCA Algorithms Based on LCA Stream Processing Properties.** In this chapter, we present a third SLCA-based algorithm, SLCAStream, for processing multiple keyword-based queries over XML streams. Unlike the previous SLCA algorithms, SLCAStream is based on unexplored LCA stream processing properties. Based on throughly experimental evaluation, we compared SLCAStream with MKStream in terms of response time and memory usage. Our evaluation results show that SLCAStream is faster and provides a better performance memory than MKStream [Barros et al., 2012b].

- **Chapter 5 – ELCA Algorithms based on LCA Stream Processing Properties.** In this chapter, we present our two ELCA-based algorithms, ELCABStream and ELCAStream, for processing multiple keyword-based queries over XML streams. These algorithms are also based on further LCA stream processing properties for query evaluation. Based on throughly experimental evaluation, we compared ELCABStream with ELCAStream in terms or response time and memory usage. Our evaluation results show that ELCAStream is slightly faster and provides a better performance memory than ELCABStream [Barros et al., 2012b].

- **Chapter 6 – Ranking Algorithms and Accuracy Evaluation.** In this chapter, we present an accuracy evaluation of the SLCA and ELCA semantics, by comparing the results provided by SLCAStream and ELCAStream algorithms. For this comparison, we used as a baseline, results provided by an XPath benchmark. In addition, we also present the algorithm LCARank [Barros et al., 2010] and the heuristics SLCARank and StreamRank as ranking strategies. Finally, we present a performance and accuracy evaluation of the LCARank algorithm, which provides an initial ranking strategy for keyword-based query results over XML streams.

- **Chapter 7 – Conclusions and Future Work.** In this concluding chapter, we review our contributions by summarizing our proposed algorithms and ranking strategies, then this chapter concludes the thesis, presenting its challenges and contributions. Finally, we discuss future work, focusing on open or new issues related to improving keyword-based query processing and ranking strategies over XML streams.

Chapter 2

Querying XML Documents Using Keywords

This chapter reviews the main related work on querying XML documents using keywords. The addressed work was grouped into three main topics which guide this thesis. Specifically, Section 2.1 addresses algorithms for XML query processing over stored XML documents and streams, Section 2.2 algorithms for processing multiple queries over XML and Section 2.3 algorithms for ranking results of keyword-based queries processed over XML documents. In each section, we emphasize how this thesis contributes to the state-of-the-art.

2.1 Algorithms for Keyword-based Query Processing

Algorithms for keyword-based queries are divided into two groups. The first refers to algorithms for processing queries on *stored* XML documents and the second to algorithms for processing queries on XML *streams*. The first group considers stored XML documents and auxiliary structures to process them, such as inverted lists, indexes and hash tables. The second group considers XML document streams and avoids using auxiliary structures since XML streams flow continuously without retention. In the stream environment, retentions can cause data loss due to the large data volume involved, thus the use of any kind of auxiliary structure impacts query processing.

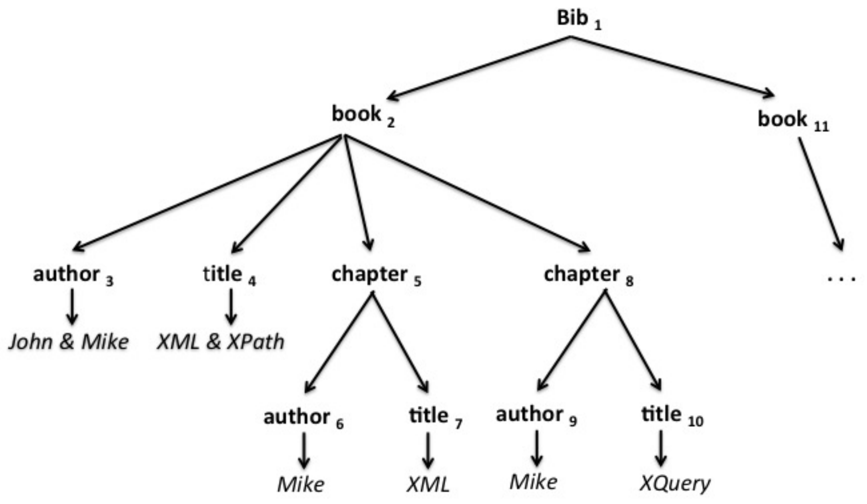


Figure 2.1. Example of XML document: a book with two chapters.

2.1.1 Querying Stored XML Documents

Algorithms for processing queries based on keywords employ heuristics that decide whether a node[1] satisfies a user query. Most consider stored XML documents, i.e., documents are permanently stored and indexed. Usually, the best answers to XML queries are sets of nodes containing the user query terms [Termehchy and Winslett, 2009]. After applying some heuristic, search algorithms return the XML document fragments that satisfy the queries. In this context, most heuristics employ the LCA (Lowest Common Ancestor) semantics [Liu and Chen, 2008, Liu and Chen, 2011, Sun et al., 2007, Tian et al., 2011, Vagena et al., 2007a, Xu and Papakonstantinou, 2008, Zhou et al., 2010]. Current work based on the LCA semantic focuses on accuracy and performance improvements. However, they are based on stored XML documents and their auxiliary structures, such as indexes and inverted lists. Some of them are presented next.

Xu and Papakonstantinou [2008] propose the algorithm Indexed Stack, which improves the XRANK algorithm [Guo et al., 2003] performance. Indexed Stack uses a keyword index that points to the nodes containing the keywords. According to Zhou et al. [2010], the original XRANK algorithm adopts the ELCA semantics.

Liu and Chen [2008] propose MaxMatch, an algorithm that reduces the size of original results based on the SLCA semantics. Specifically, MaxMatch prunes the sibling subtrees, considered less relevant for SLCA results, because they match fewer query terms. For example, consider the XML document in Figure 2.1. Unlike the

[1]Nodes in XML documents are elements or attributes.

example in Figure 1.1, the document in Figure 2.1 has two chapters in node *book*$_2$. For the query q=$\{Mike, XML\}$, original SLCA-based algorithms return the entire subtree rooted by node *book*$_2$, while MaxMatch prunes the ancestor node *chapter*$_8$ from the *book*$_2$ subtree. MaxMatch prunes the node *chapter*$_8$ because it only satisfies the query keyword $Mike$, thus being *chapter*$_8$ less relevant than its sibling node *chapter*$_5$, which satisfies all q keywords. To prune less relevant sibling descendants, MaxMatch uses inverted lists and B-tree indexes.

Zhou et al. [2010] present the algorithm HashCount that adopts the ELCA (Exclusive Lowest Common Ancestor) semantics. ELCA is based on the LCA semantics. According to the ELCA semantics, the result of a keyword query q is the set of nodes containing at least one occurrence of all its query keywords either in their child or descendant nodes, after excluding the keyword occurrences in the subtrees containing at least one occurrence of all query keywords. For example, consider the keyword-based query q=$\{Mike, XML\}$ and the XML document in Figure 1.1. Nodes *book*$_2$ and *chapter*$_5$ are ELCA since q keywords occur in *book*$_2$ and *chapter*$_5$ descendants. Although, *book*$_2$ also contains q keyword occurrences under *chapter*$_5$, it is an ELCA result because it contains exclusive occurrences for the keywords $Mike$ and XML in its descendants *author*$_3$ and *title*$_4$ respectively.

Specifically, HashCount records the number of keyword occurrences in an XML node and its descendants to determine whether a node is ELCA. Specifically, it counts the number of keyword occurrences in an ELCA candidate node and its descendants, and compares this number with the number of child node keyword occurrences. If the candidate node has its own query keyword occurrences, it is an ELCA node. HashCount considers the query q=$\{k_1, k_2, \cdots, k_i\}$ and defines the function $C_n(k_1, k_2, \cdots, k_i)$=$(c_1, c_2, \cdots, c_i)$ which returns c_j as the number of k_j keyword occurrences in node n or its descendants, being $1 \leq j \leq i$. For example, considering the XML document in Figure 1.1 and the query q=$\{Mike, XML\}$, then $C_{book_2}(Mike, XML)$=$(2, 2)$, because the keywords $Mike$ and XML occur twice in the subtree rooted by *book*$_2$. For node *chapter*$_5$, $C_{chapter_5}(Mike, XML)$=$(1, 1)$. According to HashCount, *book*$_2$ is an ELCA node as C_{book_2} is greater than $C_{chapter_5}$ for all q keywords. To evaluate ELCA candidates, HashCount uses a hash structure, which stores the number of keyword occurrences for each node in an XML document. This hash structure involves only stored XML documents, thus it does not consider incoming XML documents such as in stream environments.

All aforementioned algorithms only work on *stored* XML documents and rely on auxiliary structures, such as indexes and inverted lists, to perform appropriately. The algorithms presented by Liu and Chen [2011] confirm recent research efforts to improve the performance of LCA-based algorithms and corroborate LCA as the usual semantics for keyword-based queries on XML applications [Liu and Chen, 2008, Liu and Chen, 2011, Sun et al., 2007, Tian et al., 2011, Vagena et al., 2007a, Xu and Papakonstantinou, 2008, Zhou et al., 2010]. Our proposed algorithms differ from those because they work on the XML stream environment, which has been little explored. In this environment, building auxiliary structures for XML document streams is unfeasible. Finally, for more information on querying XML stored documents, we refer the reader to [Gou and Chirkova, 2007b] for XPath and XQuery processing and [Liu and Chen, 2011] for keyword-based search.

2.1.2 Querying XML Streams

Several works propose algorithms for query processing over XML streams and, most of them, are based on the XPath language[2] [Wu and Theodoratos, 2012]. These algorithms are usually based on automata or stack structures and are designed to improve performance and memory usage [Chen et al., 2006, Onizuka, 2010, Peng and Chawathe, 2005, Ramanan, 2009]. However, there are few works that address keyword-based queries [Barros et al., 2010, Hummel et al., 2011, Vagena and Moro, 2008]. As this thesis is related to keyword-based queries over XML streams, we only describe works related to this subject.

Vagena and Moro [2008] have proposed two algorithms for keyword-based query processing over XML streams. Both are based on original versions of the algorithms XRANK [Guo et al., 2003] and SLCA [Xu and Papakonstantinou, 2005]. However, that work presents no performance or accuracy evaluation of the proposed algorithms. Barros et al. [2010] provide this evaluation as well as introduce an algorithm, LCARank, that combines the two algorithms proposed by Vagena and Moro [2008] to establish a simple ranking strategy.

More recently, Hummel et al. [2011] have proposed two algorithms, KStream and CKStream, which are multiple query versions of the single query SCLA algorithm proposed by Vagena and Moro [2008]. KStream and CKStream are discussed in Section 2.2 since they are the first algorithms proposed in literature for processing multiple keyword-based queries.

Except these two, all aforementioned algorithms only process one query per XML stream. However, KStream and CKStream only consider the SLCA semantics. Their adaption to other semantics, such as ELCA, is unfeasible. The algorithms proposed in this thesis, on the other hand, besides processing multiple queries, significantly improv-

[2]http://www.w3.org/TR/xpath/

ing KStream and CKStream response time and memory consumption, also consider the SLCA and ELCA semantic. We adopted ELCA in our algorithms because it improves SLCA accuracy as shown in Chapter 6.

2.2 Processing Multiple Queries over XML Streams

Distinct approaches have been proposed to process queries over XML streams. Among them, some evaluate multiple XPath queries over XML streams [Altinel and Franklin, 2000, Chan et al., 2002, Diao et al., 2002, Min et al., 2007, Moro et al., 2007, Onizuka, 2010, Vagena et al., 2007b]. These approaches use non-deterministic finite automata (NFA) or compressed data structures of candidate XML nodes to evaluate multiple queries. They focus on improving performance and memory usage.

Other algorithms approach the problem of processing multiple queries processing over XML streams from a different view point [Lee and Lee, 2009, Park et al., 2009]. Lee and Lee [2009] propose XP-table, a system that transforms multiple XPath queries into a specific data structure used for matching incoming XML streams. The XP-table structure is designed to minimize the run-time workload of continuous query processing. Part et al. [2009] present M-COPE (Multiple Continuous Query Processing Engine), a scalable query processing engine that efficiently evaluates multiple continuous SQL-like queries. Thus, in both cases, the authors disregard keyword-based queries.

To the best of our knowledge, Hummel et al. [2011] have been the first to address the problem of multiple keyword-based queries over XML streams. However, their KStream and CKStream algorithms only consider the SLCA semantic and their adaptation to ELCA is unfeasible. CKStream implements a parsing stack whose entries are associated with visited nodes during document transversing. Each stack entry contains a single and compact bitmap that represents the distinct terms for all queries. When an opened node matches a query term, the corresponding bit is set to true in the entry bitmap. Upon closing a node, if the bits associated with a query are complete, it means the node matches the query being processed. KStream and CKStream follow a similar approach, thus they have similar performance. However, CKStream improves KStream memory consumption. Therefore, in this thesis, we use CKStream as our main SCLA baseline algorithm. As we shall see, our algorithms process multiple keyword-based queries over XML streams according to both SLCA and ELCA semantic and significantly improve CKStream response time and memory consumption.

When compared to the aforementioned algorithms, the contributions of our work to the state-of-the-art are two-fold. First, our new algorithms explore both SLCA and ELCA semantics when processing keyword queries over XML streams, thus, relieving users from knowing the source schemas. Second, they allow simultaneous processing of thousands of keyword queries over XML streams while saving both memory and processing time.

2.3 Ranking Query Results

Ranking results when processing keyword-based queris over XML stream is another important issue addressed by this thesis. Current work addressing this issue proposes strategies for returning the most relevant XML nodes for keyword-based queries considering a single XML document [Bao et al., 2009, Bao et al., 2010, Cohen et al., 2003, Guo et al., 2003, Li et al., 2010, Tian et al., 2011]. Such strategies, however, are totally based on stored XML documents and auxiliary structures, such as indexes and inverted lists. They are briefly described as follows.

Guo et al. [2003] have included a ranking strategy in their XRANK algorithm for LCA results obtained when processing keyword-based queries. They adapt the well-known PageRank algorithm [Brin and Page, 1998] for XML documents. PageRank establishes the ranking of Web documents by means of their hyperlinks. This ranking reflects the probability of a document being visited randomly or from hyperlinks. In an XML document context, these probabilities are adjusted to relationships between XML document nodes[3] and consider XML data structures. However, establishing relationships between nodes in an XML document is not a usual practice according to Laender et al. [2009]. In addition, the XRANK strategy for ranking XML node results relies on stored indexes, which are unfeasible for XML streams.

The XSearch algorithm [Cohen et al., 2006] adopts an LCA semantics variation and establishes a simple ranking strategy for result nodes. This strategy combines the TF-IDF similarity scheme, the size of the XML document tree and the relationship among the XML document nodes. However, this algorithm only processes a single document and relies on stored indexes for establishing relationships between nodes. Moreover, this strategy requires a previous knowledge of the XML document structure, thereby limiting its applicability.

Bao et al. [2009] describe the search engine XReal which extends the TF-IDF similarity scheme to rank XML fragments returned by a keyword-based query over a single stored XML document. XReal tries to identify if a keyword-based query term

[3]Relationships between XML nodes are established by XML attributes *ID* and *IDREF*.

is an XML label or XML content. Furthermore, this algorithm attempts to resolve possible ambiguities since keywords may occur in the document schema or in element contents. Combining guidelines for node type identification, ambiguity resolution and statistics of underlying XML data, the authors present a novel XML TF*IDF ranking strategy to rank individual matches of all possible search intentions. Bao et al. [2010] provide an interactive search strategy based on XReal by allowing users to select their desired search targets from a list of XReal suggestions. This strategy provides more precise results and learn how users perceive involved XML schemas.

Li et al. [2010] have proposed a heuristic for suggesting XML keyword query results. The method employs a ranking function using the correlation between the possible results and query keywords. This correlation is based on different types of statistical information, such as XML content and XML structure distributions, and uses a new data structure called XSketch. The heuristic implements the proposed method in a keyword search engine prototype called XBridge. Again, XBridge uses stored structures such as inverted lists and indexes to access statistical information. As mentioned before, these structures are unfeasible in XML stream environments.

Ranking strategies surveyed by Tian et al. [2011] and all aforementioned algorithms reinforce the great interest in improving result accuracy obtained by algorithms for keyword-based query processing. However, these algorithms only process stored XML documents and use stored auxiliary data structures. Such structures are unfeasible in XML stream environments. In contrast, this thesis presents the first strategies to rank results obtained by processing keyword-based queries over XML streams. Specifically, in Chapter 6, we present an algorithm, LCARank [Barros et al., 2010], that combines the SLCA and XRANK algorithms proposed by Vagena and Moro [2008] to establish a simple ranking strategy. XRANK includes SCLA nodes between other result nodes. However, SLCA nodes are the lowest result nodes, which means their descendant nodes are closer and, therefore, more meaningfully related. Thus, LCARank returns SLCA nodes first followed by other results. Chapter 6 presents LCARank in detail. It also presents the ranking strategies SLCARank and StreamRank. SLCARank is a fine-grained ranking strategy for a single document in an XML stream, thus improving LCARank, while StreamRank ranks results obtained from a set of XML documents.

Chapter 3

SLCA Algorithms Based on Bitmaps

This chapter presents our first two proposed SLCA-based algorithms, which are based on traditional bitmap structures. It is organized as follows. Section 3.1 formalizes the SLCA semantics. Section 3.2 presents a simple keyword-based query language adopted by the proposed algorithms. Section 3.3 provides an overview of our multi-query procedure as well as the data structures. Section 3.4 presents the algorithms BStream and MKStream, both based on traditional bitmap structures. Finally, Section 3.5 presents an experimental evaluation of our two algorithms in which we compare them with state-of-the-art algorithms in terms of response time and memory usage.

3.1 SLCA semantics

In this section, we formalize the SLCA semantics. First, however, we overview the lowest common ancestor (LCA) semantics.

Definition 1. *Given an XML document d and a subset $v_1, v_2, \ldots, v_m$ of nodes from d that satisfy a set of query terms $t_1, t_2, \ldots t_n$, the Lowest Common Ancestor (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 XML document d in Figure 1.1 and the query $q = \{Mike,\ \ XML\}$. The returned LCA nodes for query q are *book*$_2$ and *chapter*$_5$ because they match the keywords $Mike$ and XML in their leaf nodes. However, *book*$_2$ contains *chapter*$_5$, which means that they represent a same match for query q. Thus, to avoid this kind of problem, the SLCA semantics has been introduced [Xu and Papakonstantinou, 2005]. Definition 2 below formalizes this 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.*

Thus, for the previous LCA example, the SLCA node is $chapter_5$, whose leaf nodes match both keyword $Mike$ and XML, and there is no other LCA node as its descendant. On the other hand, $book_2$ is a non SLCA node. Despite being an LCA node for q, it does not qualify as an SLCA node because it includes $chapter_5$ as its LCA descendant. Notice that the SLCA semantics defines a subset of LCA nodes that include no LCA descendants. The intuition behind the SLCA semantics is that smaller result subtrees contain closer related nodes.

3.2 A Simple Keyword-based Query Language

Several algorithms for processing keyword-based queries over XML data allow to search for keyword in a label, in a node text or in both. Following Cohen et al. [2003] and Vagena and Moro [2008], we opted for providing better control while requiring minimal (or no) schema knowledge to formulate a query. Thus, we assume a simple keyword-based query language, with a syntax borrowed from them and defined as follows.

A keyword-based query q over an XML document stream is a list of query terms (also denoted search terms) $\langle t_1, \ldots, t_m \rangle$. Each query term is of the form: ℓ*::k*, ℓ*::*, *::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 term of the form:

- ℓ*::k*, if n's label is equal to ℓ and its textual content contains the keyword k;
- ℓ*::*, 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.

As an example, consider the following query specifications:
s_a: *New comedies starring Lewis*
s_b: *New comedies having "Lewis" in their title*
s_c: *New movies in color format by director named "Color"*

Query	Formulation	Specification
q_1	*Actor*::*Lewis* *Genre*::*Comedy*	s_a
q_2	*Actor*::*Lewis* *Comedy*	s_a
q_3	*Lewis* *Genre*::*Comedy*	s_a, s_b
q_4	*Lewis* *Comedy*	s_a, s_b
q_5	*Title*::*Lewis* *Genre*::*Comedy*	s_b
q_6	*Title*::*Lewis* *Comedy*	s_b
q_7	*Color*:: *Director*::*Color*	s_c

Table 3.1. Examples of keyword-based queries.

The queries in Table 3.1 illustrate different scenarios regarding the user's knowledge about the XML labels. In queries q_1 and q_5 the user knows the labels and employs query terms of the form *ℓ::k*, where each keyword k is qualified with a node label ℓ. In queries q_2, q_3 and q_6 the user knows only some labels whereas in query q_4 the user has no knowledge at all (no labels are used) about the XML labels. Notice that q_3 and q_4 are ambiguous, i.e., they may represent both requirements s_a and s_b. This is an inherent collateral effect of using keyword-based queries. Also notice that in query q_7, the term *Color*:: is an example of a pure structural condition where XML subtrees must contain an element with label *Color*.

As an example, consider the XML document represented in Figure 3.1, which contains movie data. Consider also the query q_1={*Actor*::*Lewis*, *Genre*::*Comedy*}. The SCLA result for q_1 is the node $Movie_2$ since it is the smallest subtree that contains all query keywords.

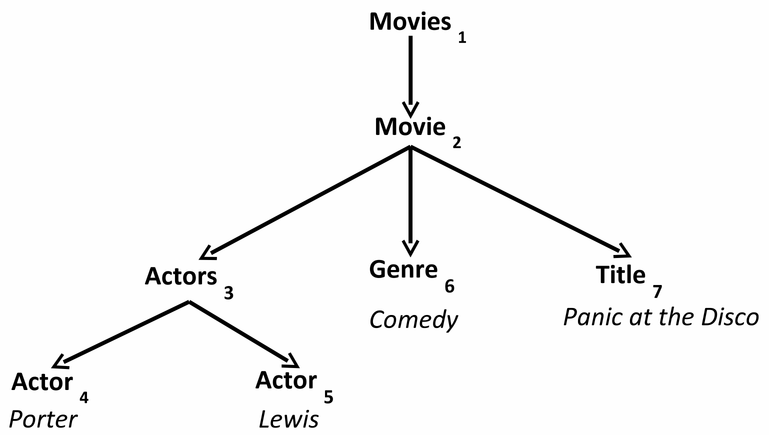


Figure 3.1. Tree representation of an XML document.

Algorithm 1 General Multi-Query Procedure.

```
Procedure MultiQuery
Input: A stream $D$ of XML documents
1. let $Q = q_1, \ldots, q_n$ 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.     $\langle r_1, \ldots, r_n \rangle$ := SAX_Parsed($d_j$,$\langle q_1, \ldots, q_n, \rangle$,S)
6.     return $r_1, \ldots, r_n$
7. end while
```

3.3 Processing Multiple Queries

Querying over XML streams follows a paradigm that is different from traditional database systems: the query engine matches queries against streams of documents, instead of stored data. Therefore, approaches that rely on indexed data and use traditional query optimization techniques do not apply.

In this section, we present an overview of our multi-query procedure for keyword-based queries as well as of the data structures used. Note that even though it is not possible to index the documents (which arrive as a stream), the queries are known beforehand (i.e., from users' profiles), thus allowing their indexing.

3.3.1 General Multi-Query Procedure

Algorithm 1 describes our general multi-query procedure. We consider that a set of queries Q are processed against a stream D of documents. Upon its arrival, each document d_j in D is individually processed (Lines 3 to 6). The results found within this document are collected and returned (Lines 5 and 6). This is accomplished at the same time for all queries q_i and results are individually collected in each r_i. A result includes the resulting node of d_j, if any.

Each document d_j in D is processed by a SAX parser, which generates five types of event for a document: *startDocument()*, *startElement(tag)*, *characters(text)*, *endElement(tag)* and *endDocument()*. Our algorithms then work by means of SAX *Callback Functions*[1] for those events. The parser is called to action in Line 5 of the procedure described by Algorithm 1.

[1]http://www.saxproject.org

3.3.2 Data Structures

3.3.2.1 Parsing Stack

As the SAX parser traverses the document in an in-order fashion, each visited node is associated with an entry in a stack S, called the *Parsing Stack*. Each entry is popped from the stack when its corresponding node and all its descendants have been visited.

To support the recursive processing of a document, each entry in the parsing stack handles the following information:

- The label of the element corresponding to the entry;
- A bitmap called CAN_BE_SLCA, which contains one bit for each query q_i being evaluated (this bitmap keeps track of which queries can still match the corresponding node as an SLCA result);
- A set used_queries containing the IDs of the queries whose terms include keywords present in the element (or its descendants), either as labels or within text values;
- Which keywords from these queries have occurred in the corresponding document node and its descendants (as explained later, the algorithms handle this information in different ways with specific time and space trade-offs).

Notice that, without loss of generality, we assume that labels have a unique meaning within a same element type in each document. For instance, in one single document, the label *Actor* is always used to represent a movie actor.

3.3.2.2 Query Index

During the traversal of a document, it is necessary to look for keywords that occur in text elements or labels. As we expect to process a large number of queries, our algorithms rely on *query indexes* in order to avoid looking up each query individually. Such a multi-query processing idea is commonly used in document filtering methods [Chen et. al, 2008, Diao et al., 2004, Vagena et al., 2007b]. In our case, the indexing structures are adaptations of inverted lists in which each index entry represents a query term and refers to queries in which this term occurs. Specifically, the query indexes store keywords occurring in query terms referring to both values and labels. Moreover, for each term, the indexes hold references to queries that use it, making a distinction between structural (label) and non-structural (value) query terms.

Notice that, as it happens with any typical inverted list, query indexes are generated beforehand from the set of queries posed by users specifying their needs. Tackling new submitted queries requires the query indexes to be rebuilt. This is a rather simple task that does not require any effort from users or developers. Moreover, it is reasonable to expect that the set of queries posed by users is stable, while documents keep arriving through streams.

3.4 Proposed Algorithms

The two algorithms proposed in this section run as SAX callback functions on the general procedure described in Section 3.3.1. Both algorithms are based on bitmaps, whose bits correspond to query terms. During the XML node transversal, query term occurrences are recorded by setting their corresponding bits. When bits corresponding to terms of a query are complete, the algorithms evaluate if the node being processed satisfies the SLCA definition. The algorithms represent the query terms by distinct bitmap configuration, which makes difference in terms of response time and memory usage as described next.

BStream is the first algorithm, which is a multi-query version of the SLCA algorithm for searching over XML document streams proposed by Vagena and Moro [2008]. BStream is a straightforward implementation of Vagena and Moro's SLCA algorithm and its bitmap configuration individually represents each query term. Moreover, BStream uses a single parsing stack and is the basis for the second algorithm, called MKStream. MKStream also uses bitmaps for query evaluation. However, it compacts the bitmap representation within the parsing stack, optimizing space consumption. In addition, MKStream significantly improves BStream response time by separating all queries in different stacks whose queries are evaluated as necessary. At node closing, MKStream pops specific stacks, whose query lists are smaller than that evaluated by BStream, which holds the entire query list. Moreover, MKStream outperforms the two state-of-the-art algorithms for processing multiple keyword-based queries over XML streams, KStream and CKStream [Hummel et al., 2011], as we shall see in our experimental evaluation.

3.4.1 BStream

In the following subsections, we describe our first two SLCA-based algorithms. As mentioned, BStream is a simple alternative for processing multiple keyword queries since it is a straightforward implementation of Vagena and Moro SLCA algorithm. Its data structures and callback functions serve as a basis for MKStream and other algorithms. Therefore, they are described next.

3.4.1.1 Data Structures

BStream uses a bitmap to store information about which terms (labels or keywords) from each query occur in an XML node. This bitmap, query_bitmap, is kept in the stack entry of each XML node. Each one of its bits is associated with one search term from each query q_i, where $q_i \in Q$, the set of submitted queries. When a node matches any query term, its corresponding bit receives true. If the bitmap corresponding to a query is complete, the bitmap node or its descendants match all query terms. Figure 3.2 presents the query_bitmap for all queries in Table 3.1. In this figure, each cell $q_{i,j}$ refers to the j-th term of query q_i. Notice that the size of the query_bitmap is equal to the total number of terms in all queries.

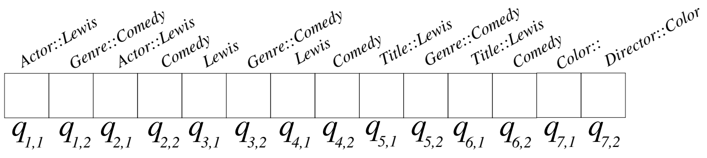


Figure 3.2. Configuration example of BStream query_bitmap.

Additionally, BStream uses a query index to identify which queries match the node terms (label or keyword) being processed. It adds these queries to the set used_queries, included in each stack entry. Thus, when BStream finishes a node, it pops the respective stack entry and evaluates only the queries in the set used_queries instead of the entire set of submitted queries. This index is implemented using an inverted list in which each index entry represents a distinct keyword. Each of these index entries is associated with two lists: one corresponding to occurrences in labels and the other to occurrences in values. Figure 3.3 presents the query_index configuration for queries in Table 3.1.

3.4.1.2 Callback Functions

Initially, BStream.Start (Algorithm 2) creates a new stack entry for the node being processed (Line 2). For this entry, the CAN_BE_SLCA bitmap and all query bitmaps are initialized. While CAN_BE_SLCA bits are set to true (Line 4), bits in query bitmaps are set to false (Line 7). Then, BStream.Start stores in the set QL the queries that contain the label in j (Line 10). After that, for each query in set QL (Line 11 to 20), BStream.Start sets to true the corresponding bits whose query terms are of the form ℓ or ℓ:: (Line 16) and adds all queries in QL to the set used_queries in the current stack entry (Line 21). Finally, BStream.Start pushes entry sn into the stack S

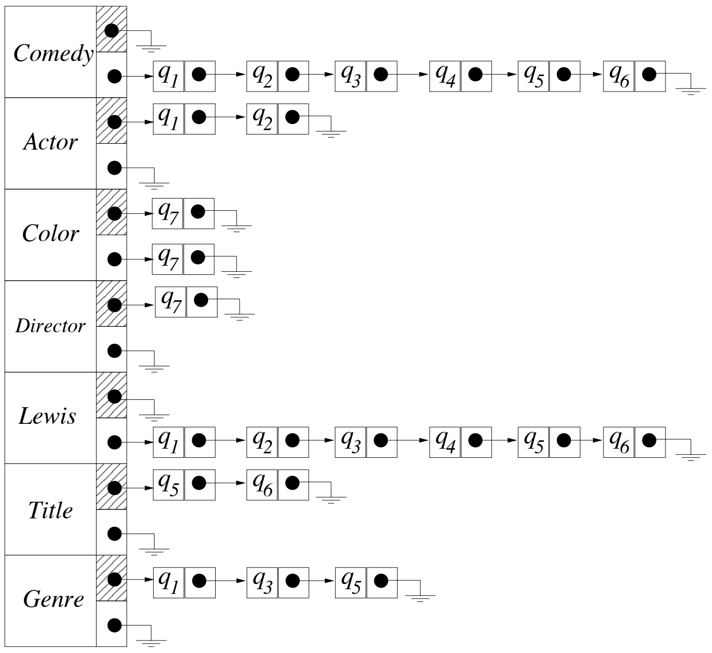


Figure 3.3. Configuration example of BStream query_index.

(Line 22). As mentioned before, BStream.End will only evaluate the set of queries in QL, instead of the whole set of submitted queries.

BStream.Text identifies and sets to true the bits in query_bitmap corresponding to terms of the form ℓ::k, ::k and k, which represent the combination of labels and text tokens. Initially, it tokenizes the node content and stores the resulting tokens in K (Line 2). BStream.Text also gets the reference to the stack top entry (Line 3), which corresponds to the XML node whose content is being processed. For each token, BStream.Text retrieves the queries whose terms contain the token (Line 5) and stores them in the set used_queries (Line 6). For these queries, the function sets to true the bits that correspond to terms of the form ℓ::k, ::k and k (Lines 10 to 15).

Finally, BStream.End evaluates which nodes or its descendants match the submitted queries. It first pops the stack top entry corresponding to the node being closed (Line 1) and retrieves the queries from the set used_queries for being evaluated (Line 3). For each query, this function verifies if its corresponding bit in CAN_BE_SLCA is true (Line 4). If so, the current node is eligible to be a smallest lowest common ancestor (SLCA) for the current query. In addition, the function verifies if the corresponding entry in query_bitmap is complete (Lines 7 and 8). If so, the query terms occur in the current node or in its descendants. Thus, this node is an SLCA result for the query (Line 9). However, its parent node tn is no longer eligible to be an SLCA result for the query (Line 10), since the current node is the lowest result. The parent node becomes then the new stack top entry after the current node is popped out (Line 2). As the

Algorithm 2 BStream.Start Callback Function

```
Callback Function BStream.Start
Input: The parsing stack $S$
Input: set of queries $Q$
Input: The XML node $e$ being processed
1.  $j$ := label($e$)
2.  sn.label := $j$ {create and initialize a new stack entry}
3.  for all $q \in Q$ do
4.     sn.CAN_BE_SLCA[$q$]:= $true$
5.     $i$ := $q.first\_bit\_position$ {first bit position in sn.query_bitmap for $q$}
6.     $n$ := $q.\#terms$ {number of query terms in $q$}
7.     sn.query_bitmap:= [$i$, $i$+1, ..., $i$+$n$-1] := $false$
8.  end for
9.  {get queries that contain the label in $j$}
10. $QL$ := query_index[$j$].labelOcurrences
11. for all $q \in QL$ do
12.    {for queries in $QL$, set bits corresponding to terms of the form $\ell$ and $\ell$::}
13.    $i$ := $q.first\_bit\_position$
14.    for all $term$ in $q.terms$ do
15.       if $term = j$ or $term = j$||"::" then
16.          sn.query_bitmap[$i$] := $true$
17.       end if
18.       $i$:=$i$+1{corresponding bit position for the next query term in $q$}
19.    end for
20. end for
21. sn.used_queries.add($QL$)
22. $S$.push(sn) {push the stack entry sn into stack $S$}
```

current node bitmap records which query terms occur in the node or in its descendants, BStream.End copies the true bits to the parent node bitmap_map (Line 14). Moreover, BStream.End propagates to the parent node those queries that can no longer be SLCA results (Line 15), since the current CAN_BE_SLCA bitmap records which queries became SLCA results in the current node or in its descendants. Finally, the function adds all queries from the set sn.used_queries sn to the set tn.used_queries tn (Line 16) since these queries can be satisfied when closing the parent node.

3.4.1.3 Example

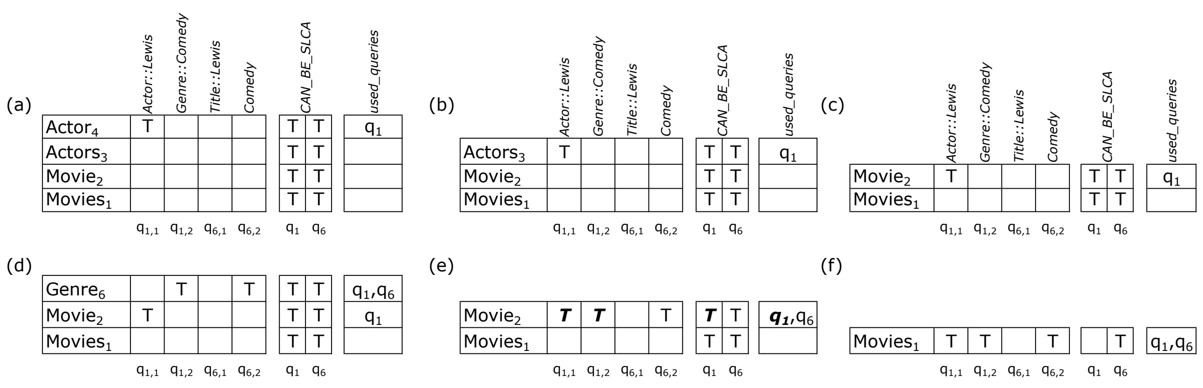
We now discuss a very simple example to illustrate how BStream works. Consider the XML document represented in Figure 3.1. For simplicity, we show how BStream processes this document against only two queries from Table 3.1: q_1 (*Actor*::*Lewis Genre*::*Comedy*) and q_6 (*Title*::*Lewis Comedy*). Also consider the

Algorithm 3 BStream.Text Callback Function

```
Callback Function BStream.Text
Input: The parsing stacks $S_1, \ldots, S_{|Q|}$
Input: The XML node $e$ being processed
1.  $j$ := label($e$)
2.  $K$ := set of tokens in node $e$
3.  sn := *S.top {sn points to the top entry in the stack S }
4.  for all $k \in K$ do
5.    $QL$ := query_index[$k$].keywordOcurrences
6.    sn.used_queries.add($QL$)
7.    for $q \in QL$ do
8.      $i := q.first\_bit\_position$
9.      {for queries in $QL$, set bits corresponding to terms of the form $k$, $\ell$::$k$, ::$k$}
10.     for all $term$ in $q.terms$ do
11.       if $term = k$ or $term = j$||"::"||$k$ or "::"||$k$ then
12.         sn.query_bitmap[$i$] := $true$
13.       end if
14.       $i$:=$i$+1{bit position corresponding to next query term}
15.     end for
16.   end for
17. end for
```

Algorithm 4 BStream.End Callback Function

```
Callback Function BStream.End
Input: The set of queries $Q$
Input: The parsing stacks $S_1, \ldots, S_{|Q|}$
Input: The XML node $e$ that is ending
1.  sn := pop(S) {pops the top entry in the stack S to sn}
2.  tn := *S.top {tn points to the top entry in the stack S}
3.  for $q \in$ sn.used_queries do
4.    if sn.CAN_BE_SLCA[$q$] then
5.      $i := q.first\_bit\_position$ {first bit position in sn.query_bitmap for $q$}
6.      $n := q.\#terms$ {number of query terms in $q$}
7.      COMPLETE := sn.query_bitmap[$i$] and ... and sn.query_bitmap[$i$+$n$-1]
8.      if COMPLETE then
9.        $q$.results := $q$.results ∪ sn
10.       tn.CAN_BE_SLCA[$q$]:= $false$
11.     end if
12.   end if
13. end for
14. tn.query_bitmap:=tn.query_bitmap or sn.query_bitmap
15. tn.CAN_BE_SLCA := tn.CAN_BE_SLCA and sn.CAN_BE_SLCA
16. tn.used_queries.add(sn.used_queries)
```


(a)

	Actor::Lewis	Genre::Comedy	Title::Lewis	Comedy	CAN_BE_SLCA		used_queries
Actor$_4$	T				T	T	q_1
Actors$_3$					T	T	
Movie$_2$					T	T	
Movies$_1$					T	T	
	$q_{1,1}$	$q_{1,2}$	$q_{6,1}$	$q_{6,2}$	q_1	q_6	

(b)

	Actor::Lewis	Genre::Comedy	Title::Lewis	Comedy	CAN_BE_SLCA		used_queries
Actors$_3$	T				T	T	q_1
Movie$_2$					T	T	
Movies$_1$					T	T	
	$q_{1,1}$	$q_{1,2}$	$q_{6,1}$	$q_{6,2}$	q_1	q_6	

(c)

	Actor::Lewis	Genre::Comedy	Title::Lewis	Comedy	CAN_BE_SLCA		used_queries
Movie$_2$	T				T	T	q_1
Movies$_1$					T	T	
	$q_{1,1}$	$q_{1,2}$	$q_{6,1}$	$q_{6,2}$	q_1	q_6	

(d)

Genre$_6$		T		T	T	T	q_1,q_6
Movie$_2$	T				T	T	q_1
Movies$_1$					T	T	
	$q_{1,1}$	$q_{1,2}$	$q_{6,1}$	$q_{6,2}$	q_1	q_6	

(e)

Movie$_2$	***T***	***T***		T	***T***	T	***q_1***,q_6
Movies$_1$					T	T	
	$q_{1,1}$	$q_{1,2}$	$q_{6,1}$	$q_{6,2}$	q_1	q_6	

(f)

Movies$_1$	T	T		T		T	q_1,q_6
	$q_{1,1}$	$q_{1,2}$	$q_{6,1}$	$q_{6,2}$	q_1	q_6	

Figure 3.4. Parsing stack states when processing queries q_1 and q_6 against the document in Figure 3.1.

query index presented in Figure 3.3, but without the other queries. Figure 3.4 shows distinct states of the parsing stack when processing queries q_1 and q_6 against the document in Figure 3.1. When BStream.Text opens the node $Actor_4$, it pushes a new stack entry. By using the query index, it also identifies that q_1 contains the term $Actor{::}Lewis$. Thus, it sets to true the corresponding bit in the top query_bitmap and adds q_1 to the set used_queries. Figure 3.4(a) presents the stack state after opening the node $Actor_4$. This figure also includes the entries corresponding to the nodes $Actors_3$, $Movie_2$ and $Movies_1$, which were previously pushed into the stack. As these nodes do not satisfy any of the query terms, their corresponding bitmaps remain as initialized. Then, BStream.End closes node $Actor_4$ and pops out its corresponding stack entry. The query_bitmap entry is incomplete for query q_1. So this query remains unsolved even though the corresponding node satisfies the SLCA semantics. Additionally, BStream.End copies the current true bits to the new top query_bitmap, which corresponds to node $Actors_3$. It also adds q_1 to the set used_queries in the new top entry, since this entry may satisfy q_1 when closing node $Actors_3$. Moreover, BStream.End applies an **and** operation over the bitmaps CAN_BE_SLCA from the popped and the new top entries. Notice that this operation causes no impact on the new top entry. Figure 3.4(b) presents the stack state after closing node $Actor_4$. Notice that Figure 3.4 ignores the stack states of node $Actor_5$, because it does not satisfy any of the query terms and causes no effect on the current stack states.

Similarly to $Actor_4$, when BStream.End closes node $Actors_3$, it updates the new top entry, which corresponds to node $Movie_2$. This update operation involves the following data structures: query_bitmap, used_queries and CAN_BE_SLCA. Figure 3.4(c) illustrates this operation. When BStream.Start function opens node $Genre_6$, it pushes a new stack entry and identifies that both q_1 and q_6 contain the terms *Genre*::*Comedy* and *Comedy* respectively. Thus, it sets the corresponding bits to true in the query_bitmap entry and adds q_1 and q_6 to set used_queries. Figure 3.4(d) presents the stack state after opening node $Genre_6$. At $Genre_6$ closing step, BStream.End copies the popped out entry true bits to the new top entry query_bitmap, which corresponds to node $Movie_2$. It also adds q_6 to the new top entry used_queries set for evaluation when closing node $Movie_2$. Figure 3.4(e) presents the next stack state after closing node $Genre_6$. Similarly to node $Actor_5$, in Figure 3.4(e), the algorithm ignores the $Title_7$ node processing since it does not satisfy any of the query terms. Finally, Figure 3.4(f) presents the stack state after closing node $Movie_2$. Notice that this node is a result for query q_1 because it satisfies the SLCA semantics and the q_1 bitmap is complete. However, node $Movie_2$ does not satisfy q_6 because its corresponding bitmap is incomplete. BStream.End also copies the popped out entry true bits to the new top entry, which corresponds to node $Movies_1$. In addition, it adds q_1 and q_6 to the new top entry used_queries set. However, as expressed by the CAN_BE_SLCA bitmap, node $Movies_1$ is no longer eligible to be an SLCA result for q_1 since its descendant is already an SLCA result.

3.4.2 MKStream

MKStream reduces BStream space usage by compacting the query index and the bitmap representation within the parsing stack. MKStream also improves BStream performance by evaluating fewer queries than BStream. Specifically, it separates all queries into different stacks which are evaluated as necessary. Thus, each stack has fewer queries for evaluation, improving the overall response time significantly. Its data structures, callback functions and examples are presented next.

3.4.2.1 Data Structures

Unlike BStream, MKStream avoids redundant information on search terms occurring in more than one query. In BStream, we use N bits to represent the same search term in N different queries. On the other hand, MKStream uses only *one* bit for each distinct search term in all queries. Thus, each stack entry includes a different configuration for query_bitmap, in which a bit is associated with each *distinct* query term. Figure 3.5 illustrates the query_bitmap version used by MKStream when processing the queries in Table 3.1. As a consequence, in the query_index the entry corresponding to a term t needs only to store the position of the bit corresponding to this term in the query_bitmap, as illustrated in Figure 3.6.

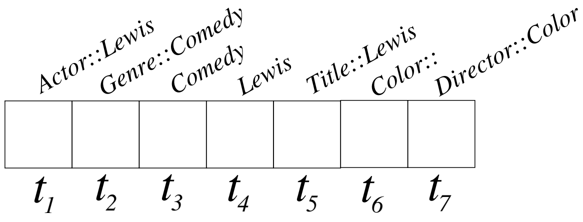


Figure 3.5. Configuration example of MKStream query_bitmap.

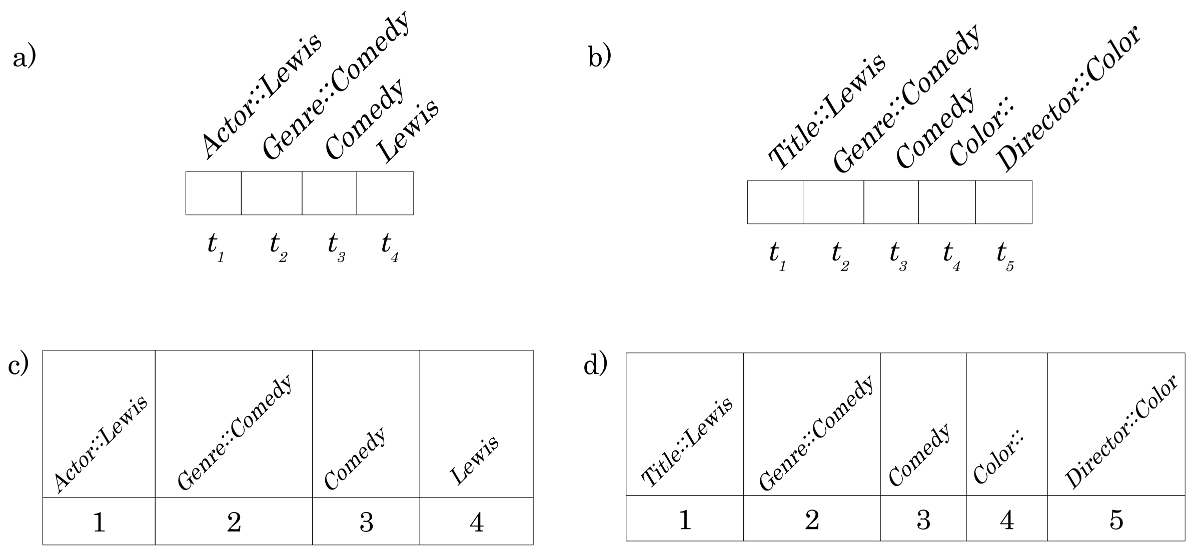
Color::	Lewis	Comedy	Director::Color	Genre::Comedy	Title::Lewis	Actor::Lewis
6	4	3	7	2	5	1

Figure 3.6. Configuration example of MKStream query_index.

MKStream design aims to reduce the number of pushing and popping stack operations and the number of query evaluations, thus improving response time and space consumption. To reduce pushing operations, a node is pushed into a parsing stack only if its label or keywords occur in some query. To reduce popping operations, entries in the stacks are popped only if they correspond to the node being closing. Furthermore, MKStream can process multiple parsing stacks simultaneously. Therefore all queries can be distributed into different stacks, which means controlling a smaller number of queries if compared to a single parsing stack. By working with multiples stacks, MKStream can process only the stacks whose top entries correspond to a node being processed. Since not all parsing stacks are processed at each node, MKStream reduces the number or query evaluations because each parsing stack controls a subset of queries. Thus, MKStream improves performance compared with BStream, which uses only one parsing stack and pushes and pops all processed XML nodes.

MKStream separates the initial set of queries into G query groups, G being a user defined parameter. Each query group is controlled by a parsing stack. Thus, for G query groups, MKStream handles G parsing stacks. The number of queries for each query group is up to $\lceil |Q|/G \rceil$, being Q the set of queries.

As each MKStream parsing stack controls its own query group, the corresponding query_bitmap represents a specific set of query terms. For example, considering the queries in Table 3.1 and G=2, i.e., two query groups, the first parsing stack controls queries q_1, q_2, q_3 and q_4, and the second one queries q_5, q_6 and q_7. Figures 3.7.a and 3.7.b illustrate the corresponding query_bitmap structures. As each query group uses a specific query_bitmap, it requires a specific query_index. Considering the query_bitmap instances in Figures 3.7.a and 3.7.b, the corresponding query_index



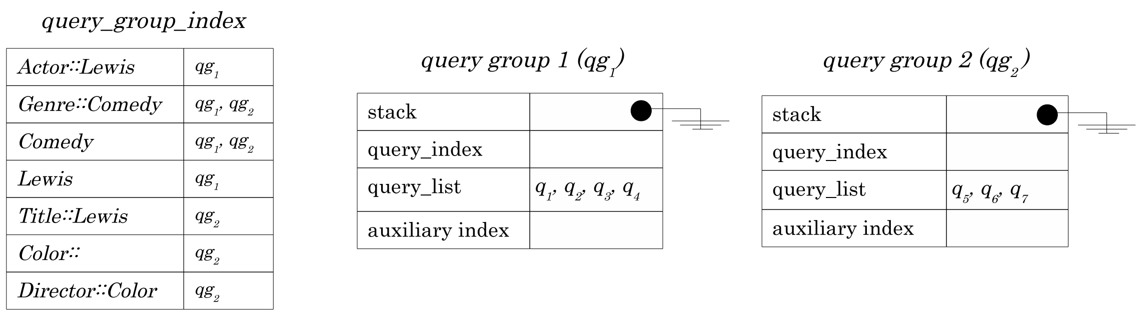
c)

Actor::Lewis	Genre::Comedy	Comedy	Lewis
1	2	3	4

d)

Title::Lewis	Genre::Comedy	Comedy	Color::	Director::Color
1	2	3	4	5

Figure 3.7. Examples of MKStream query_bitmap and query_index instances


query_group_index

Actor::Lewis	qg_1
Genre::Comedy	qg_1, qg_2
Comedy	qg_1, qg_2
Lewis	qg_1
Title::Lewis	qg_2
Color::	qg_2
Director::Color	qg_2

query group 1 (qg_1)

stack	●
query_index	
query_list	q_1, q_2, q_3, q_4
auxiliary index	

query group 2 (qg_2)

stack	●
query_index	
query_list	q_5, q_6, q_7
auxiliary index	

Figure 3.8. Examples of MKStream query_group_index and query_group

instances are presented in Figures 3.7.c and 3.7.d.

MKStream identifies the correct parsing stack for each query by using the global index query_group_index, whose keys are query terms and entries are query_group instances. For example, considering G=2 and queries in Table 3.1, Figure 3.8 illustrates the query_group_index configuration, whose keys are the corresponding query terms and whose entries are the query_group instances qg_1 and qg_2. Each query_group instance includes the fields stack, query_index, query_list and auxiliary_index.

Specifically, the field stack keeps a parsing stack, whose query_bitmat configuration is specific to query_group. For example, considering the query_group instance qg_1, Figure 3.7.a illustrates its query_bitmap configuration. MKStream also uses a query index structure, but different from that used by BStream. MKStream query_index stores the bit position corresponding to a query term in query_bitmap. For the query_group instances $qg1$ and $qg2$, the query_index fields contain the instances shown in Figures 3.7.c and 3.7.d, respectively. Each query_group instance also includes the field query_list, which lists the respective queries. For the query_group instances $qg1$ and $qg2$, for example, their query_list fields in Figure 3.8 list their respective queries.

MKStream also uses an auxiliary indexing structure to speed up the search for queries that contain a certain query term. This index, called auxiliary_index, is implemented using an inverted list where each index entry represents a query term and refers to the queries in which this term occurs. Considering the queries in Table 3.1 and a single parsing stack, Figure 3.9 presents the auxiliary_index example. Specifically, each MKStream query_group contains its own auxiliary_index, which is kept in the auxiliary_index field, as depicted in Figure 3.8. Each auxiliary_index contains its own keys and entries, according to the queries in the corresponding query_group. For simplicity, we omit the auxiliary_index instances in Figure 3.8.

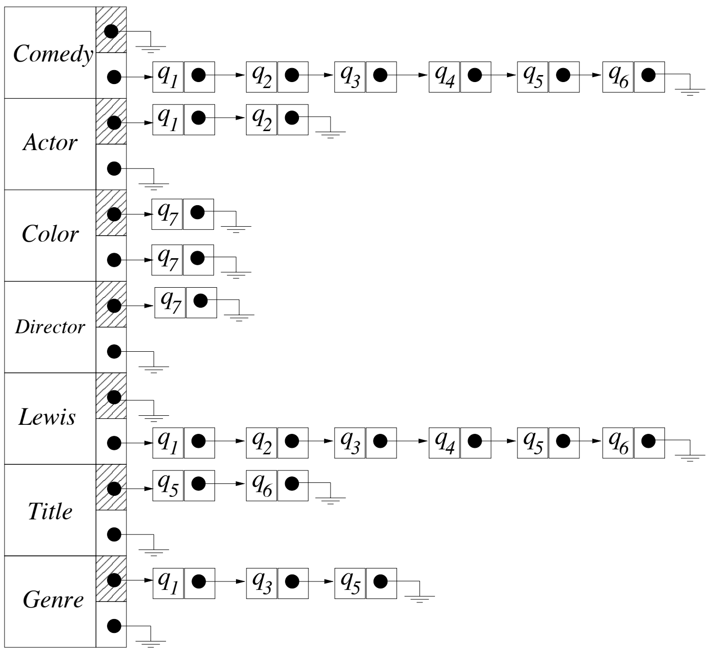


Figure 3.9. Example of MKStream auxiliary_index.

An important setup step in MKStream is the query grouping. As a basic heuristics, it tries to separate queries in groups whose queries involve common terms. Thus, when MKStream searches for queries with terms matching the node being processed, all returned queries are controlled by a single parsing stack belonging to a single query_group. Therefore, MKStream considers that all queries have been correctly grouped into G groups with a maximum of $\lceil |Q|/G \rceil$ queries each.

This grouping process is performed by a simple procedure. It starts by adding the first query to the first group. For each of the remaining queries, this procedure determines the first query group that contains any term of the current query and adds the query to this group. Otherwise, it inserts the query in a new group, or in the smallest query group when the number of groups G has already been reached. This procedure also builds the query_group_index by using the previously obtained query group. For each query group, it initializes the parsing stack stored in the stack field and its query_list. Furthermore, the algorithm creates the query_index and the auxiliary_index.

3.4.2.2 Callback Functions

MKStream.Start, MKStream.Text and MKStream.End functions are described by Algorithms 5, 6 and 7 respectively. They play similar roles than their counterparts in the previous algorithm, but they push and pop fewer nodes for being processed. The previous algorithm pushes and pops all nodes and uses a single stack for all queries.

MKStream.Start searches the query_group_index for the current node label in j and obtains the query groups associated with that label (Line 3). It processes each query_group by filling its stack up to the node being processed (Lines 6 to 11). For each node, MKStream.Start sets the query_bitmap (Line 8) and the respective CAN_BE_SLCA bitmaps (Line 9), and then pushes the node into its stack (Line 10). It also gets the stack top entry reference and stores it in sn (Line 12). Finally, it adds the group query list Q to the set used_queries (Line 15) and sets to *true* the bit position associated with the label occurrence in the query_bitmap of entry sn (Line 16) .

MKStream.Text process each keyword found in the text query group (Line 2), filling the stacks up to the node being processed (Lines 7 to 12). These stacks control queries that include the keyword k (Line 4). For each stack top entry (Line 15), MKStream.Text adds the corresponding group query list Q to the set used_queries (Line 16) and sets to *true* the bit position associated with the current keyword occurrence in query_bitmap (Line 17). Similarly to BStream, terms of the form l::k are handled by MKStream.Text like k terms.

Finally, MKStream.End retrieves all query groups (Line 1) whose stacks will be processed. To be processed, the stack top entry must correspond to a node being finished, meaning that the stack height is equal to the height of the node being processed (Line 3). For each one of these stacks, MKStream.End evaluates if the stack top entry satisfies any query controlled by the stack (Lines 9 to 17). For this evaluation, MKStream.End identifies the positions of the bits corresponding to q terms in the

Algorithm 5 MKStream.Start Callback Function

```
Callback Function MKStream.Start
Input: query group index query_group_index
Input: The XML node $e$ being processed
1.  $j$ := label($e$)
2.  node_path[$e.height$] := $j$
3.  groups := query_group_index[$j$]
4.  for all gr $\in$ groups do
5.      $N$ := number of distinct terms for all queries of the group gr
6.      for $i$ := gr.stack.height+1 to $e.height$ do
7.          sn$_i$.label := node_path[$i$]
8.          sn$_i$.query_bitmap[0,. . .,$N$-1] := $false$
9.          sn$_i$.CAN_BE_SLCA[gr.query_list[1], . . . ,gr.query_list[gr.query_list.size()]:=
            $true$
10.         gr.stack.push(sn$_i$)
11.     end for
12.     sn := *gr.stack.top()
13.     $q$ := gr.query_index[$j$].asLabel
14.     $Q$ := gr.auxiliary_index[$j$].labelOccurrences
15.     sn.used_queries.add($Q$)
16.     sn.query_bitmap[$q$] := $true$
17. end for
```

current *query_bitmap* (Line 10) and checks in *query_bitmap* if the corresponding bits are complete (Line 11).

3.4.2.3 Example

We now present an example that illustrates how MKStream works. This example is quite similar to the one from Section 3.4.1.3, as it involves the same queries q_1 (*Actor*::*Lewis Genre*::*Comedy*) and q_6 (*Title*::*Lewis Comedy*) and the same XML document presented in Figure 3.1. However, it considers two query groups, i.e., $G = 2$. Each group uses its own parsing stack, i.e., the first handles query q_1 and the second handles query q_6.

By using query_group_index, MKStream.Start opens the $Actor_4$ node and identifies that $Actor_4$ and its keyword *Lewis* are associated with the first query group, i.e., query group g_1. Then, differently from BStream.Text, MKStream.Text pushes the nodes $Actor_4$, $Actors_3$, $Movie_2$ and $Movies_1$ into the first group stack at once. It also sets to true the bit corresponding to the $Actor_4$::*Lewis* term on the top query_bitmap and adds q_1 to the used_queries set. Figure 3.10(a) presents the stack instance after opening node $Actor_4$. Notice that nothing happens to the second query group struc-

Algorithm 6 MKStream.Text Callback Function

```
Callback Function MKStream.Text
Input: query group index query_group_index
Input: The XML node $e$ being processed
1.  $j$ := label($e$)
2.  $K$ := set of tokens in node $e$
3.  for all $k \in$ K do
4.    groups := query_group_index[$k$]
5.    for all gr $\in$ groups do
6.      $N$ := number of distinct terms for all queries of the group gr
7.      for $i$ := gr.stack.height+1 to $e.height$ do
8.        sn$_i$.label := node_path[$i$] {create stack entry for node $i$}
9.        sn$_i$.query_bitmap[0,...,$N$-1] := $false$
10.       sn.CAN_BE_SLCA[gr.query_list[1],...,gr.query_list[gr.query_list.size()]:=
          $true$
11.       gr.stack.push(sn$_i$)
12.     end for
13.     $q$ := gr.query_index[$k$].asKeyword {$q$ gets the position of term $k$}
14.     $Q$ := gr.auxiliary_index[$k$].KeywordOccurrences
15.     sn := *gr.stack.top() {sn points to the top entry in the group stack}
16.     sn.used_queries.add($Q$)
17.     sn.query_bitmap[$q$] := $true$
18.   end for
19. end for
```

tures since node $Actor_4$ does not satisfy any of its terms. Next, MKStream.End closes the $Actor_4$ node and pops out its corresponding stack entry. Notice that it evaluates only the first query group stack since the stack top entry corresponds to the node being closed (the stack and the closed node have the same height). However, MKStream.End discards $Actor_4$ as a q_1 result since the current query_bitmap is incomplete for q_1. It also copies the current true bits to the new top query_bitmap, which corresponds to node $Actors_3$, and updates the set used_queries and the new top bitmap CAN_BE_SLCA. Figure 3.10(b) presents the stack state after closing node $Actor_4$. Both query groups skip node $Actor_5$ processing since it does not match any of the q_1 or q_6 terms. Notice that the second query group structure remains unchanged.

MKStream.End closes the node $Actors_3$ and updates the new top structures for both query groups. Figure 3.10(c) illustrates the stack states after closing node $Actor_4$. MKStream.Text opens the node $Genre_6$ and identifies that $q1$ and $q6$ contain the terms $Genre$::$Comedy$ and $Comedy$, respectively. Thus, MKStream.Text sets the corresponding bits to true in both query_bitmap structures and adds q_1 and q_6 to the corresponding used_queries sets. Figure 3.10(d) presents the stack states after opening node $Genre_6$.

Algorithm 7 MKStream.End Callback Function

```
Callback Function MKStream.End
Input: query group index query_group_index
Input: The XML node $e$ that is ending
1.  $query\_groups$ := query_group_index.values {gets all query groups}
2.  for all group $in\ query\_groups$ do
3.    if group.stack.height = $e.height$ then
4.      sn := group.stack.pop() {pops the top entry in the stack to sn}
5.      tn := *group.stack.top() {tn points to the top entry in the stack}
6.      tn.CAN_BE_SLCA:=tn.CAN_BE_SLCA and sn.CAN_BE_SLCA
7.      tn.used_queries.add(sn.used_queries)
8.      sn.query_bitmap := sn.query_bitmap or tn.query_bitmap
9.      for all $q \in$ sn.used_queries do
10.       let $\{j_1, \ldots, j_N\}$) be the positions of the bits corresponding to terms from
          query $q$ in $query\_bitmap$ of the current query group
11.       COMPLETE := sn.query_bitmap[$j_1$] and ...
          and sn.query_bitmap[$j_N$]
12.       if sn.CAN_BE_SLCA[$q$] and COMPLETE then
13.         $q$.results := $q$.results ∪ { sn }
14.         tn.CAN_BE_SLCA[$q$]:= $false$
15.         tn.used_queries.remove($q$)
16.       end if
17.     end for
18.   end if
19. end for
```

Notice that MKStream pushes the entries $Genre_6$, $Movie_2$ and $Movies_1$ into the second group stack at once. At $Genre_6$ closing, for both query groups, MKStream.End copies the popped out entry true bits to the new top MKStream.query_bitmap, corresponding to node $Movie_2$. It also updates the used_queries sets and the CAN_BE_SLCA bitmaps for both query groups. Notice that MKStream.End evaluates both stacks since their top entries correspond to node $Genre_6$ (the stack top entry and $Genre_6$ have the same height). Figure 3.10(e) presents the stack states after closing $Genre_6$. Similarly to node $Actor_5$, MKStream.End ignores the query matching for $Title_7$ because it does not satisfy any of the query terms. Finally, MKStream.End closes node $Movie_2$, which is a result for q_1 since the corresponding query_bitmap in the first query group is complete. However, q_6 is not satisfied because its bitmap is incomplete. For both query groups, MKStream.End also copies the popped out entry true bits to the new top entries, which correspond to node $Movies_1$. In addition, it updates the used_queries sets and the CAN_BE_SLCA bitmaps. As can be seen from the second CAN_BE_SLCA bitmap, node $Movies_1$ is no longer eligible to be an SLCA result for

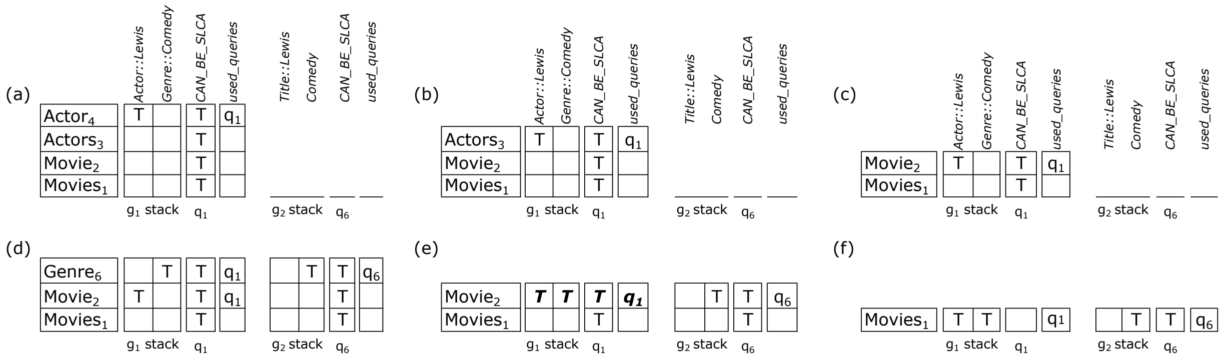


Figure 3.10. Stack instances when MKStream processes queries q_1 and q_6 against the XML document in Figure 3.1.

q_1 since its descendant node ($Movie_2$) is also an SLCA result.

3.5 Experimental Evaluation

In this section, we empirically study the efficiency of the BStream and MKStream algorithms in terms of processing time and memory space. We also compare them with the KStream and CKStream algorithms [Hummel et al., 2011], both considered the state-of-art algorithms for multiple keyword-based query processing under the SLCA semantics.

The experiments consist in processing streams of XML documents against all posed queries simultaneously and measuring the time spent and the memory consumption. We performed three different experiments with streams containing documents from real XML datasets, each one stressing a different aspect of our algorithms. In the first experiment, we analyze how the algorithms handle an increasing number of keyword-based queries. In the second experiment, we study the impact of queries that search for node labels in comparison with queries that only search for keywords on the node contents. Finally, in the third experiment, we observe the behavior of the algorithms as the number of search terms in each query increases.

Similarly to BStream and MKStream, KStream and CKStream implement a parsing stack whose entries are bitmaps. These bitmaps are also associated with visited nodes during document transversal. However, their bitmap configurations are different. Like BStream, the KStream bitmap represents all query terms individually. However, it sets simultaneously all bits corresponding to query terms that match the XML document being processed while BStream sets separately each one of those bits. CKStream and MKStream use similar bitmap configurations. However, CKStream uses a single parsing stack in which all processed nodes are pushed down while MKStream uses multiple parsing stacks in which only specific nodes are pushed down.

3.5.1 Setup

All algorithms were implemented using Java and the SAX API from Xerces Java Parser. The query indexes and other data structures are kept entirely in memory. All experiments were performed in an Intel Dual Core 2.53 GHz computer with 4 GB of memory on Mac OS.

Datasets. The experiments employ three distinct datasets. The first, called *ICDE*, consists of metadata from papers published in the proceedings of the ICDE conference, which were extracted from DBLP[2]. The second one, *ISFDB*[3], consists of bibliographic data from fiction books available on the ISFDB Web site. The last one, *SIGMODR*, contains data from the table of contents of past SIGMOD Record issues. We organized the data on ICDE papers by year of publication into a stream of 14 different XML documents, from 1995 to 2008. Similarly, in ISFDB, we separated books published between 2000 and 2009 into a stream of 10 different XML documents. Finally, we organized SIGMODR into as stream of 18 XML documents, each containing data from one issue. The average sizes of ICDE, ISFBD and SIGMODR documents are, respectively, 75 KB, 1.3 MB and 58 KB. Details on the three datasets are presented in Table 3.2.

Dataset	Docs	Avg. Height	# Elem.	Avg. Nodes	Avg. Objects	Avg. Size
ICDE	14	2	12	2073	119	75 KB
ISFDB	10	2	11	41637	4110	1.3 MB
SIGMODR	18	7	13	1697	167	58 KB

Table 3.2. Details of the datasets used for the experiments.

As we can see from Table 3.2, according to Barbosa et al. [2006a], the height of the documents in our datasets is compatible with those of typical XML datasets found on the Web. SIGMODR has the deepest documents. In ISFDB and ICDE, most book and paper elements are flat. Regarding the number of distinct elements, notice that elements that have different full paths from the root node are considered distinct. ISFDB has the largest XML documents. Hence, it has much more objects

[2]http://www.informatik.uni-trier.de/~ley/db/

[3]http://www.isfdb.org

and nodes than the other two. Furthermore, although ICDE has, on average, more nodes than SIGMODR, the latter has more sub-trees that characterize objects.

Queries. We randomly generated sets of queries using data available from each dataset. For example, using data from ISFDB, the following query could be generated: "*Author*::*jose* ::*saramago* *Title*::*blindness*". Depending on the experiment, we generated different sets of queries by using as parameters the number and the type of terms to appear in the queries.

Metrics. In order to evaluate the performance of the algorithms, we measured, for each dataset, the time spent for processing all XML documents on a given stream. Notice that this excludes the time spent to create the query indexes, which is done only once by the time the operations start. Similarly, we measure the average memory usage while processing each XML document. We recall that, as discussed in Section 3.3.1, each document is individually processed against all queries at once. Therefore, although we have used several documents per stream to obtain significant measures, there was no need for varying the number of documents per stream.

3.5.2 Results

3.5.2.1 Varying the Number of Queries

The first experiment aims at analyzing how the algorithms handle an increasing number of keyword-based queries. Notice that we consider BStream as the baseline since it is a straightforward multi-query implementation of the SLCA algorithm proposed by Vagena and Moro [2008], which processes a single query over XML streams.

In this experiment, we submitted up to 50,000 queries, each with up to 4 terms, which include only query terms as *::k*. Queries that search for labels, i.e., queries that involve structural terms such as ℓ*::* or ℓ, are addressed in Section 5.3.2.2. Figure 3.11 presents the comparison of time and memory spent by the algorithms over each of the datasets.

The time curves show that MKStream outperforms the other algorithms. Specifically, MKStream shows a significant advantage over BStream since BStream matches the query term occurrences in query_bitmap by scanning individually their correspondent bits, while MKStream avoids or reduces drastically these individual scanning by using the query_index structure. Moreover, MKStream query_index contributes to this better performance because it reduces the popping and pushing operations. Fewer

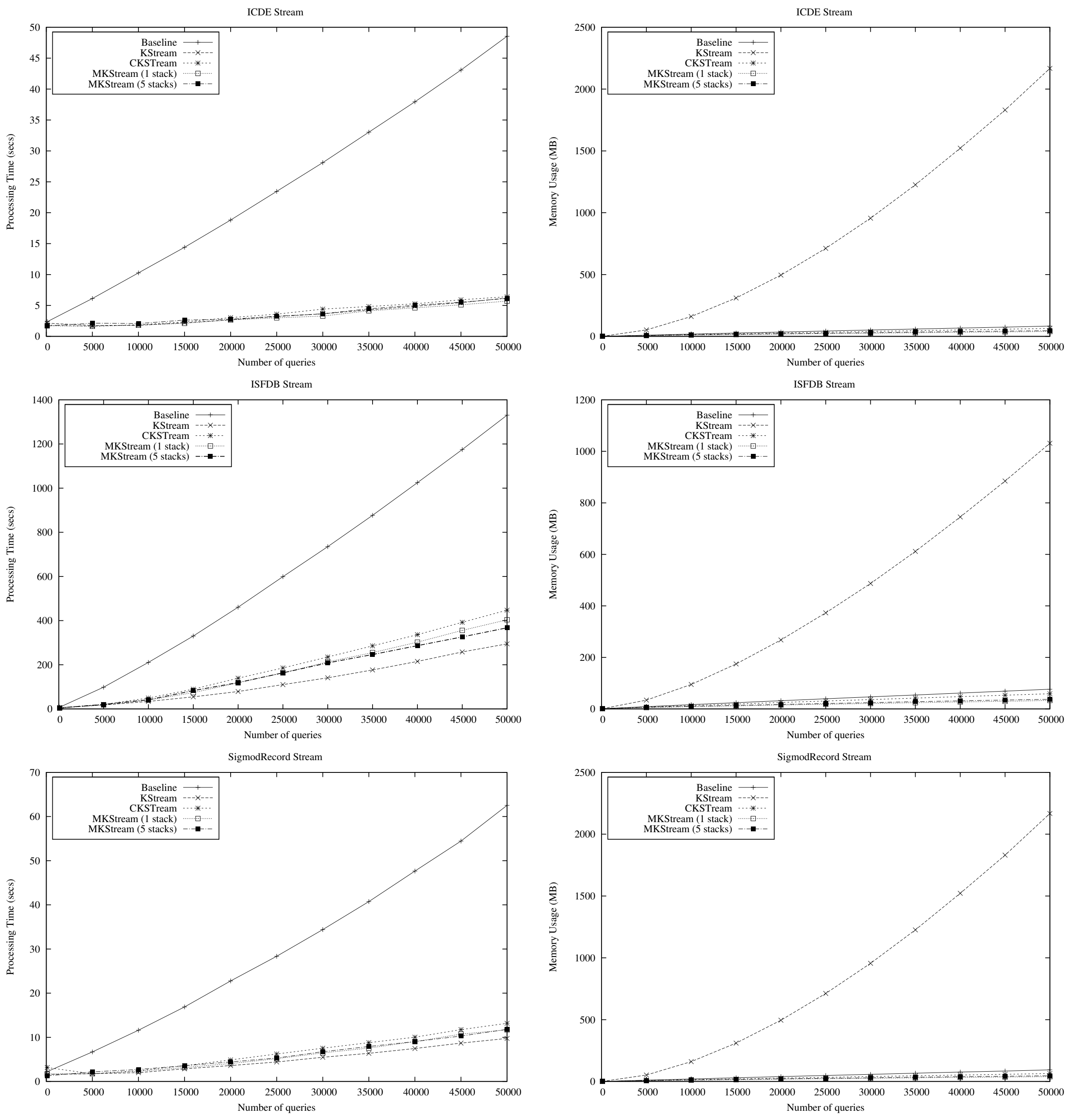


Figure 3.11. Varying the number of queries: time (left) and memory spent (right) for each algorithm in each dataset.

stack operations reduce the number of query evaluations, thus improving MKStream performance.

To evaluate MKStream behavior with more than one query group, this and the other two experiments consider one and five query groups, processed by one and five parsing stacks respectively. MKStream experiments with two, three, four and more than five query groups presented no expressive performance time gain. Therefore, our experiments do not consider such query group configurations.

Comparing the three datasets, the time performance in ISFDB is the worse. This is explained because the documents in this dataset are larger, i.e., there are more nodes that need to be processed. Also, the performance of the algorithms over the ICDE dataset is slightly better than their performance over the SIGMODR dataset. This is due to the number of nodes that satisfy the queries, which is larger in SIGMODR than in ICDE.

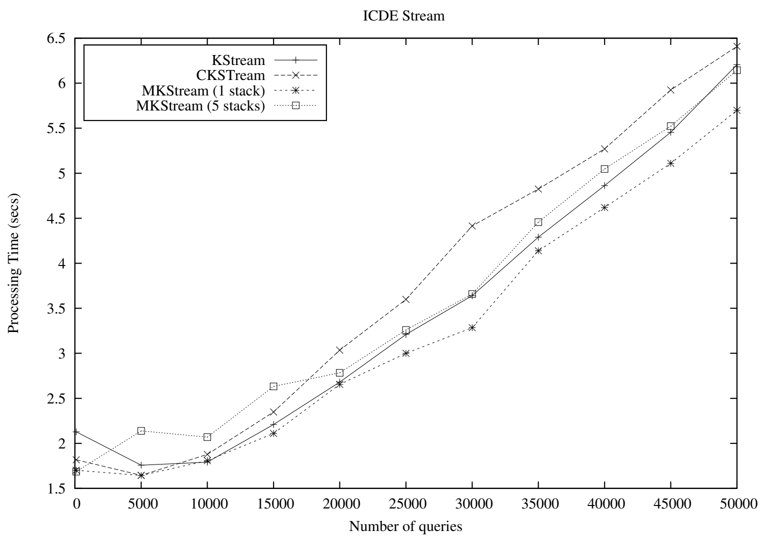
Notice that the BStream is so much slower than MKStream and the other two algorithms (Figure 3.11 – left), which makes the corresponding curves hard to be distinguished. The performance evaluation can be better observed in Figure 3.12, which shows graphs with these curves except for BStream. Figure 3.12 also shows that MKStream performs slightly faster than CKStream in all datasets because MKStream reduces the number of stack operations and query evaluations. Specifically, MKStream uses additional stacks, in which the number of queries is smaller than that processed by CKStream, since it evaluates all queries. For MKStream, the five stack configuration shows a slightly better performance than the single stack configuration when the number of the processed nodes is very large as illustrated by the ISFDB time curves.

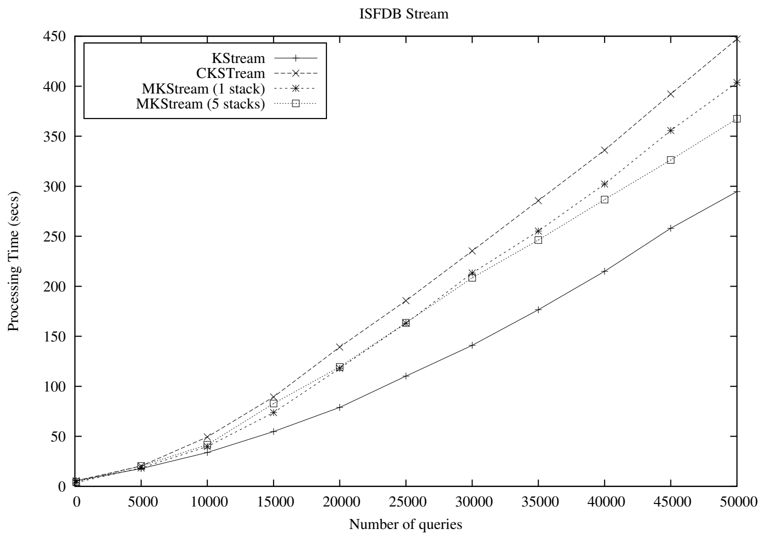
Regarding memory consumption, Figure 3.11 (right) shows that BStream, CKStream and MKStream have similar memory performance and that KStream has the worst performance. As Figure 3.13 shows, when compared only with CKStream, MKStream saves more memory. This saving is due to MKStream specific features that avoid unnecessary pushing operations, thereby reducing stack entries. Figure 3.13 also shows that, for MKStream, more stacks degrades memory performance.

3.5.2.2 Searching with Structural Constraints

The second experiment analyzes the impact of using structural terms in queries. We generated 50,000 queries with five query terms each and vary the number of structural query terms from 0 to 3. Figure 3.14 presents the comparison of time and memory spent by the three algorithms over each dataset. Notice that we do not consider BStream in this experiment, since, as shown in Section 5.3.2.1, it has a poor time performance in comparison to the other algorithms.

The results show the significant impact of structural terms in all three algorithms. Such an impact happens because, usually, structural terms occur more frequently in XML documents, when compared to non-structural ones. Consequently, the algorithms have to evaluate more queries, which means that the End callback functions have to iterate over more queries. Note that this kind of term was present in each of the 50,000 queries to stress the performance of our algorithms. In contrast, the first experiment





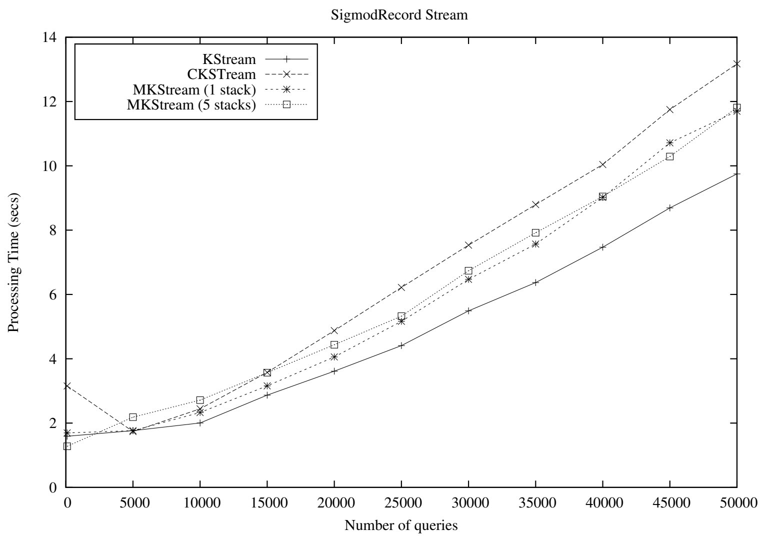
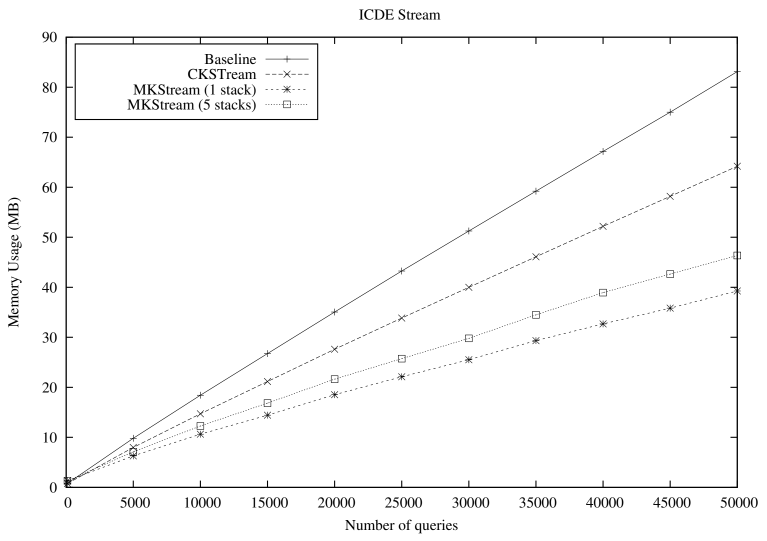
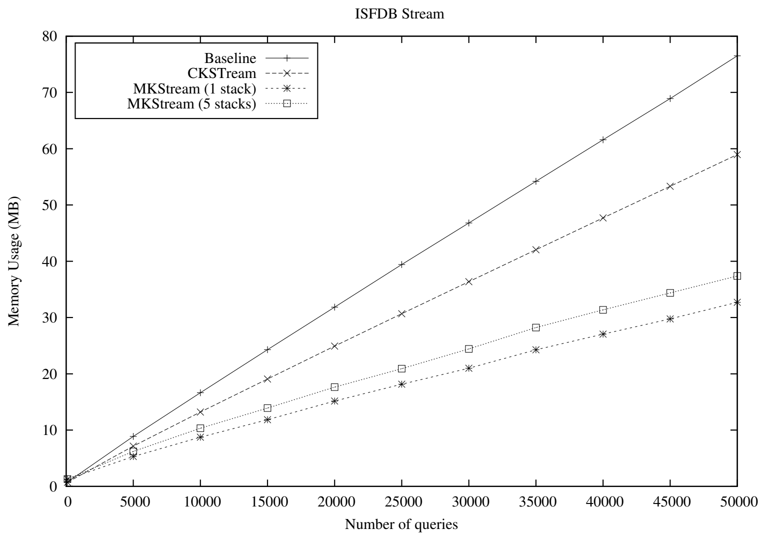


Figure 3.12. Time spent by KStream, CKStream and MKStream processing time when varying the number of queries.





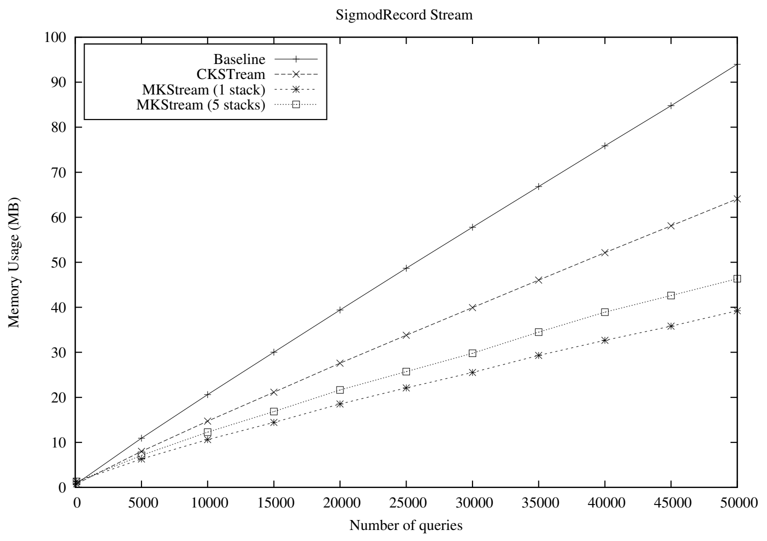


Figure 3.13. BStream, CKStream and MKStream memory usage when varying the number of queries.

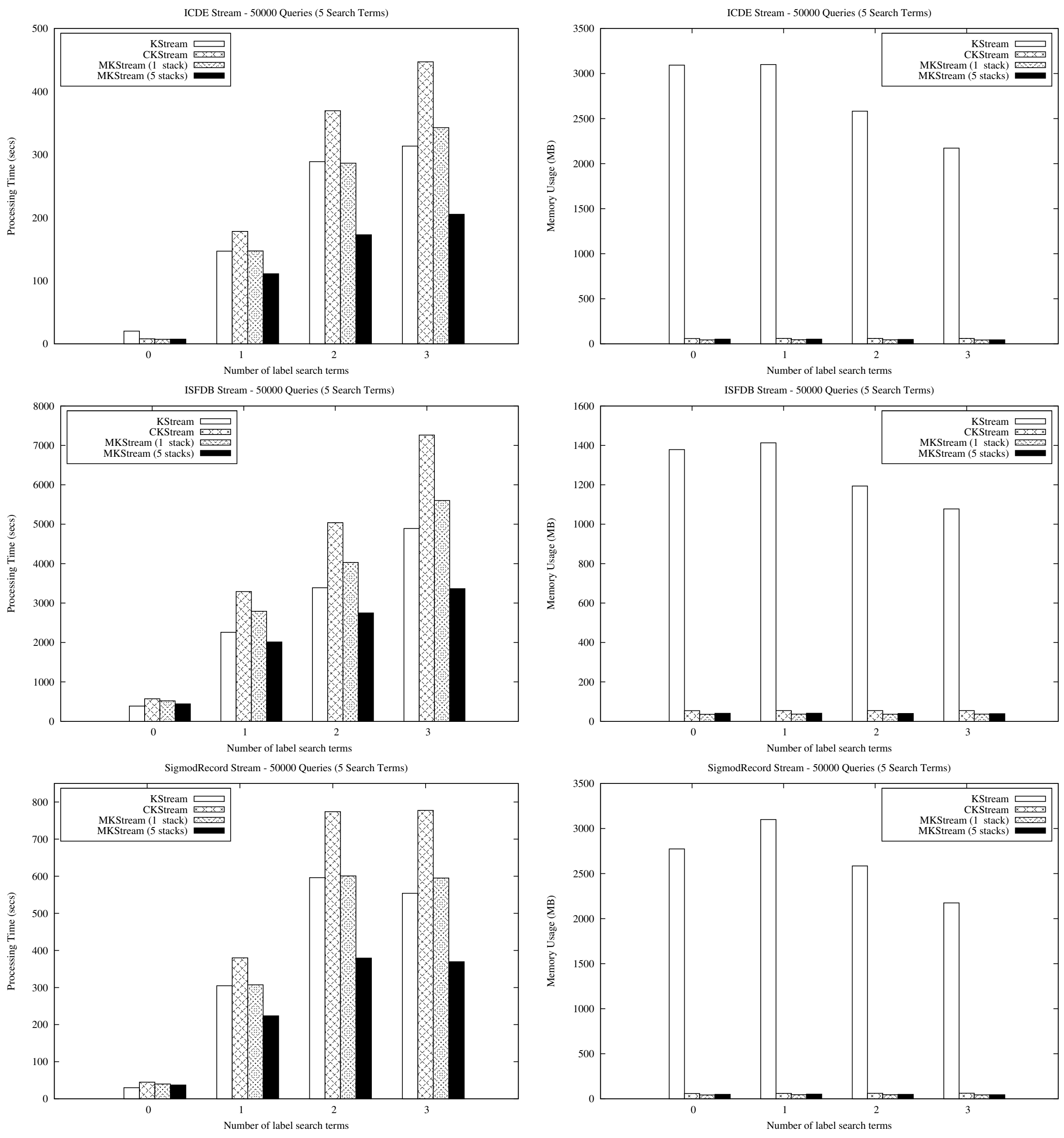


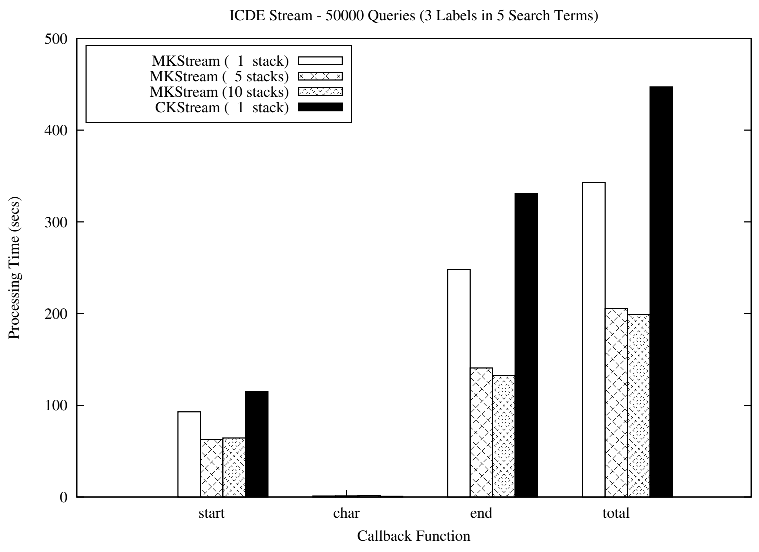
Figure 3.14. Varying the number of structural terms: time (left) and memory spent (right) for each algorithm.

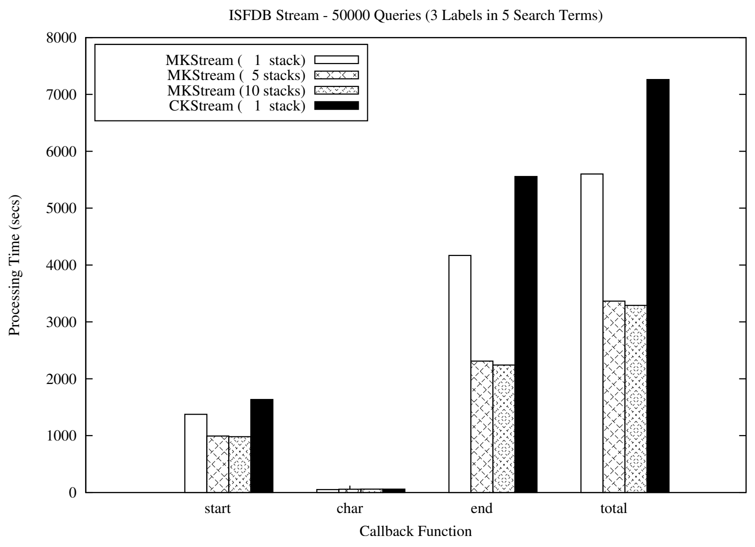
controls fewer queries because they include only non-structural terms. Since these queries were randomly generated from all documents, only few XML nodes being processed do match some of the query terms. As a consequence, fewer queries are kept in the parsing stacks, which means fewer query evaluations.

The presence of structural terms in the queries has led to a larger number of query evaluations in this experiment, thus highlighting the better time and memory performances of MKStream. Due to a small number of pushing operations, when it uses a single stack, MKStream performs similarly to KStream. However, MKStream is faster when using more than one stack. When using five stacks, response time is significantly reduced in comparison to KStream and CKStream for all datasets, i.e., 35% and 54% respectively.

Regarding memory consumption, Figure 3.14 shows that KStream has a decreasing memory usage, since adding more structural terms implies in reducing the total number of distinct terms and, therefore, the total size of the query indexes. Figure 3.14 also shows an almost constant memory usage by CKStream and MKStream algorithms when we vary the number of terms, since the number of terms has a small impact on this resource for both algorithms. However, according to Figure 3.16, MKStream presents a better memory performance due to a small number of stack pushing operations. Figure 3.16 does not show graphs for the KStream algorithm since, as show in Figure 3.14, it presents a much higher memory consumption than the other two. As expected and similarly to the first experiment, Figure 3.16 shows that MKStream presents a slight increase in memory consumption when the number of stacks is increased.

Even processing multiple stacks, MKStream processes fewer queries per stack when compared to KStream and CKStream, which use a single stack. Since not all parsing stacks are evaluated, fewer queries are evaluated by its End callback function. Thus, MKStream evaluates far fewer queries than the other two algorithms. Figure 3.15 confirms the impact of using multiple stacks when callback functions involve a larger number of queries. As specific parsing stacks are processed, fewer queries are evaluated, thus improving performance. This figure shows the processing time for each callback function type for 50,000 queries in all datasets. It only shows results for the CKStream and MKStream algorithms because they consume much less memory. Specifically for the End callback function, MKStream presents a high processing time gain since its used_queries sets hold fewer queries to be evaluated. Note that Figure 3.15 also includes MKStream results for ten stacks which slightly improves processing compared to MKStream with five stacks. We have also run MKStream with 15 and 20 stacks. However, there was no performance gain. As a result, this experiment confirms that MKStream can be customized for a large number of queries, obtaining better results than the other two algorithms. Figure 3.15 also shows the low impact of the stack pushing operations on the Start callback function for MKStream, even when a single stack is used.





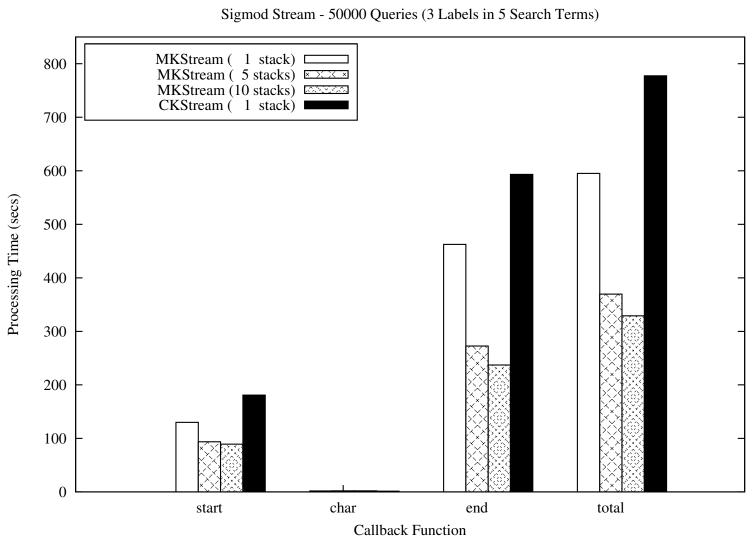


Figure 3.15. Time spent by the algorithm functions when varying the number of queries.

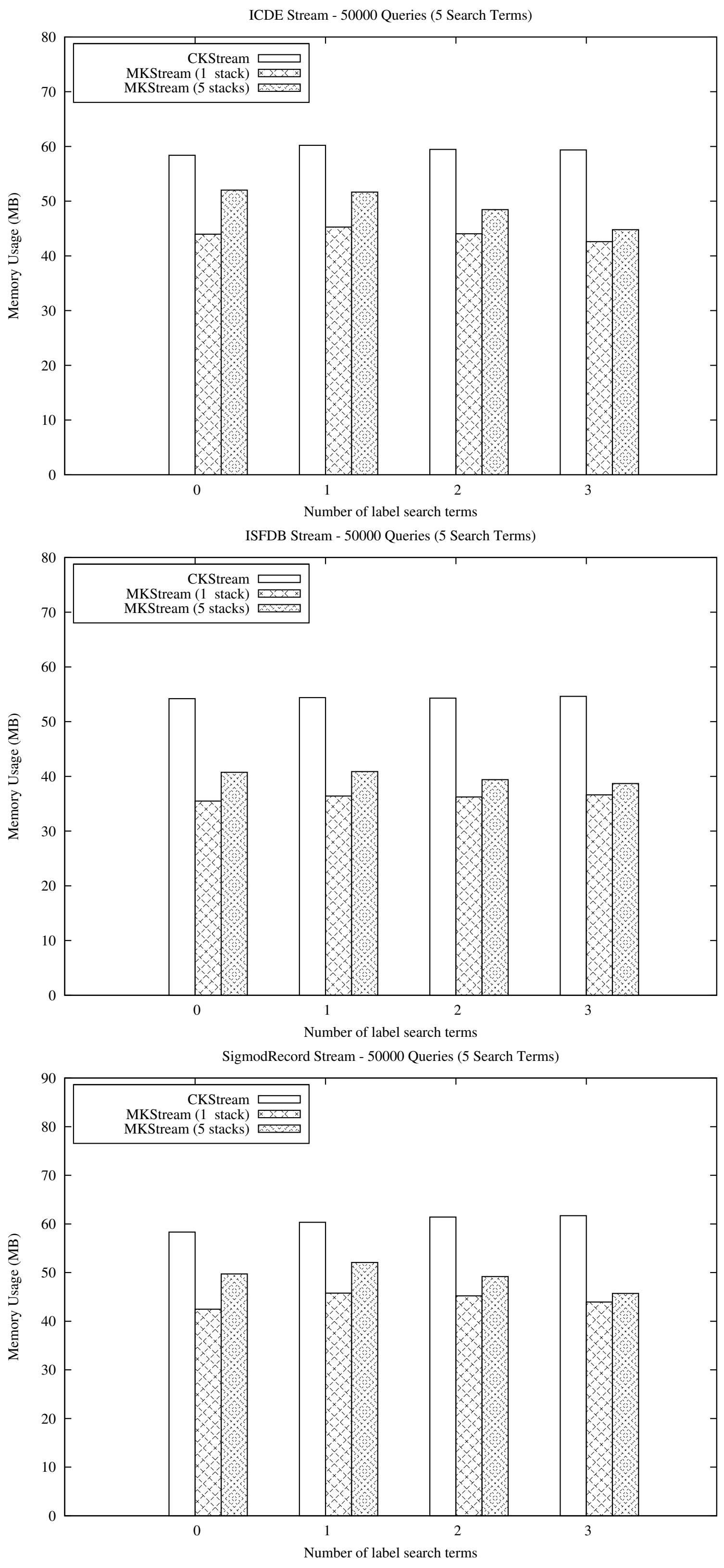


Figure 3.16. CKStream and MKStream memory usage for 50,000 queries up to 3 labels.

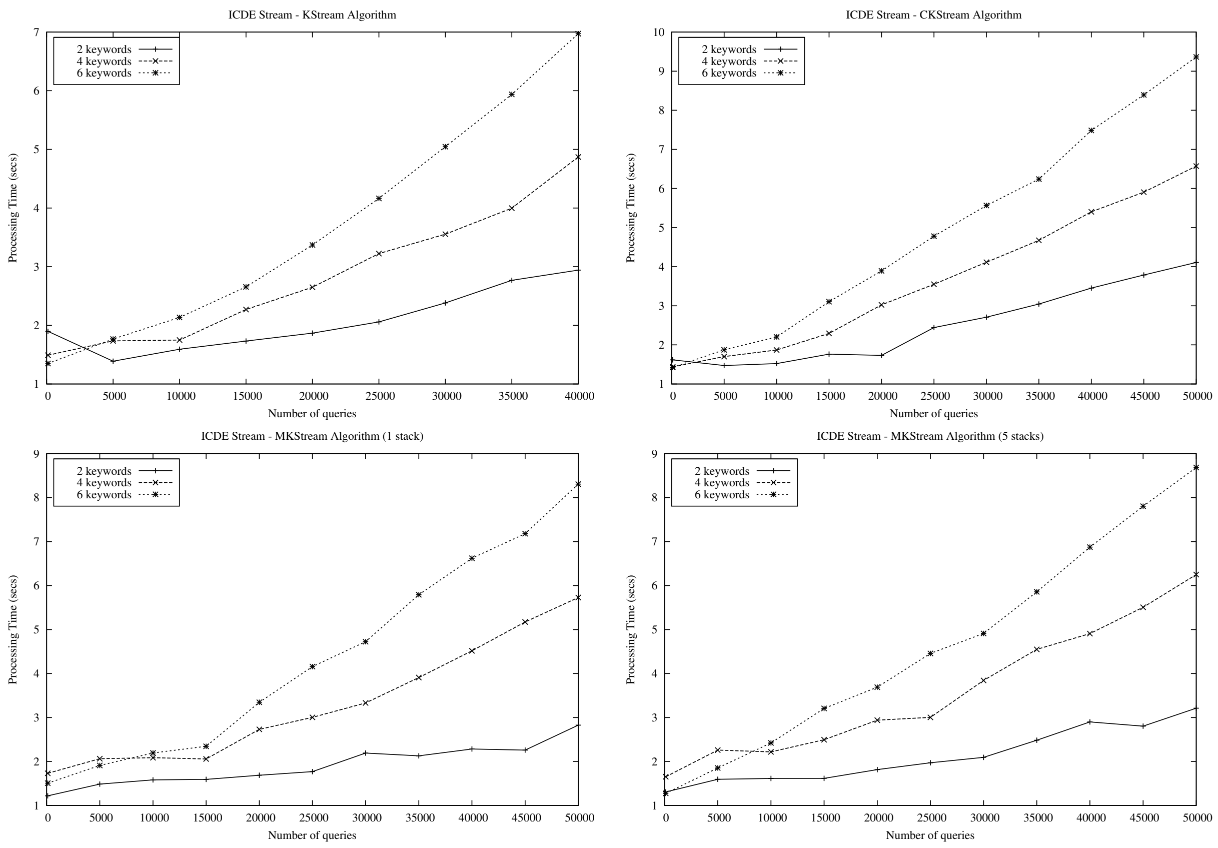


Figure 3.17. Queries with 2, 4 and 6 terms: processing time for the ICDE dataset.

3.5.2.3 Varying the Number of Terms

In this experiment, we aim to verify how MKStream scale with the number of distinct terms in the queries. This is an important factor in both time and space consumption while processing keyword-based queries, when compared to KStream and CKStream. The experiment compares the impact of using queries with two, four and six terms. We argue that six is a reasonable limit for the number of terms one typically uses when specifying a query. Similar to the first and second experiments, initially we used up to 50,000 queries. In here, queries use only terms of the form *::k*. Figures 3.17, 3.18 and 3.19 present a time comparison for KStream, CKStream and MKStream over the three datasets. Figures 3.20, 3.21 and 3.22 present a memory consumption comparison over the same three datasets.

As expected, the performance of the algorithms is affected as the number of terms increases. This effect is even bigger in the ISFDB dataset, whose documents contain more nodes. Nonetheless, comparing the three algorithms, MKStream scale well with the number of queries and the number of query terms, even though their space and time performance have some degradation. Specifically, the growth rates of the 6-term curves

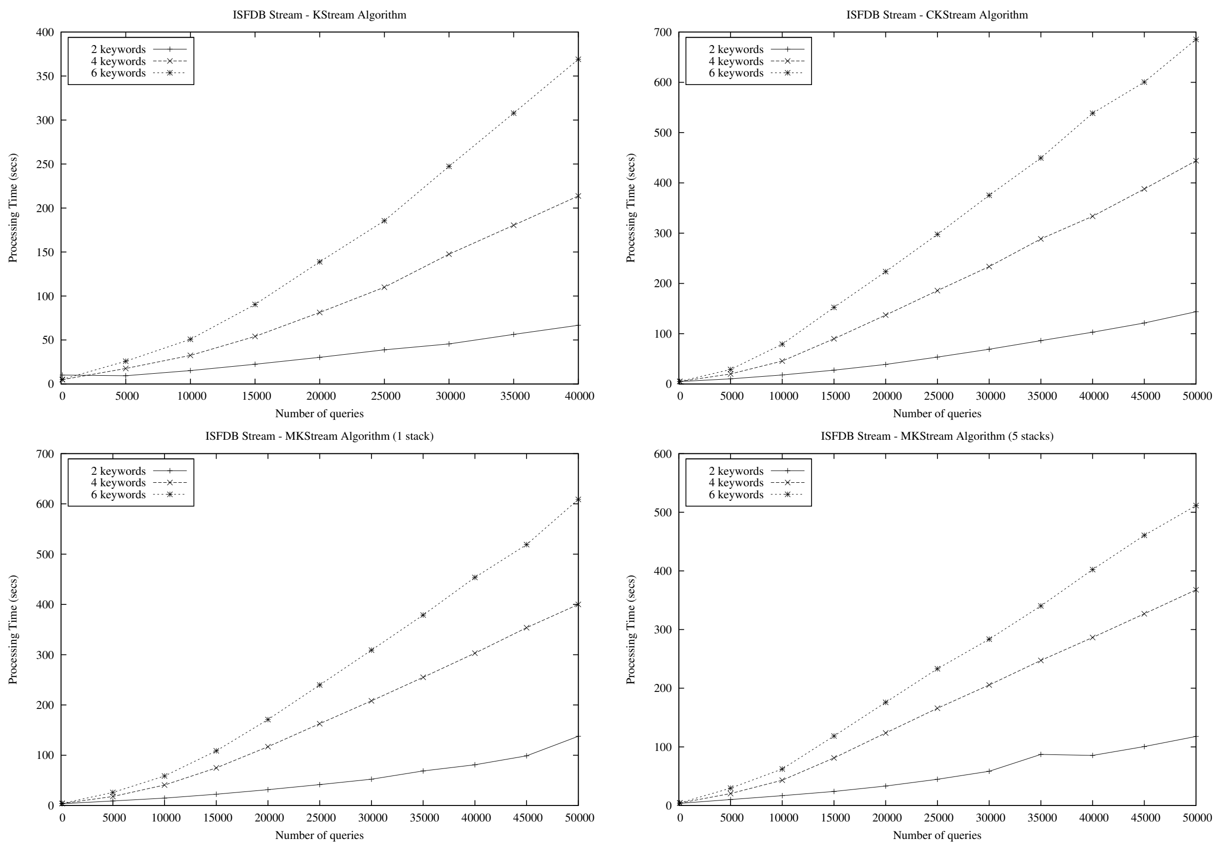


Figure 3.18. Queries with 2, 4 and 6 terms: processing time for ISFDB dataset.

are the worst. However, according to the complexity analysis presented by Barros et al. [2012a], the response time linearly increases when the number of queries increases.

Notice that KStream degrades processing time significantly when it requires more space for the Java heap. In this scenario, KStream demands a specific Java heap memory configuration, especially to finish the KStream experiments for 45,000 and 50.000 queries. As this configuration diverges from the default Java heap size used in the other experiments, Figures 3.17 to 3.19 only shows BStream time curves up to 40,000 queries.

Regarding memory usage, Figures 3.20, 3.21 and 3.22 confirm the expected behavior. MKStream suffers a larger impact than when increasing the number of terms. However, according to Figure 3.23, which excludes KStream, MKStream presents the lowest memory consumption because of fewer stack pushing operations. Similar to the first two experiments, when MKStream increases the number of stacks, its memory consumption also increases.

For a better comparison of the algorithms, Figure 3.23 presents the processing time and memory consumption of each algorithm in each dataset for up to 50,000 queries. In this figure, KStream and MKStream perform slightly better than CK-Stream. Particularly, in the ISFDB dataset, whose documents have more nodes, MK-

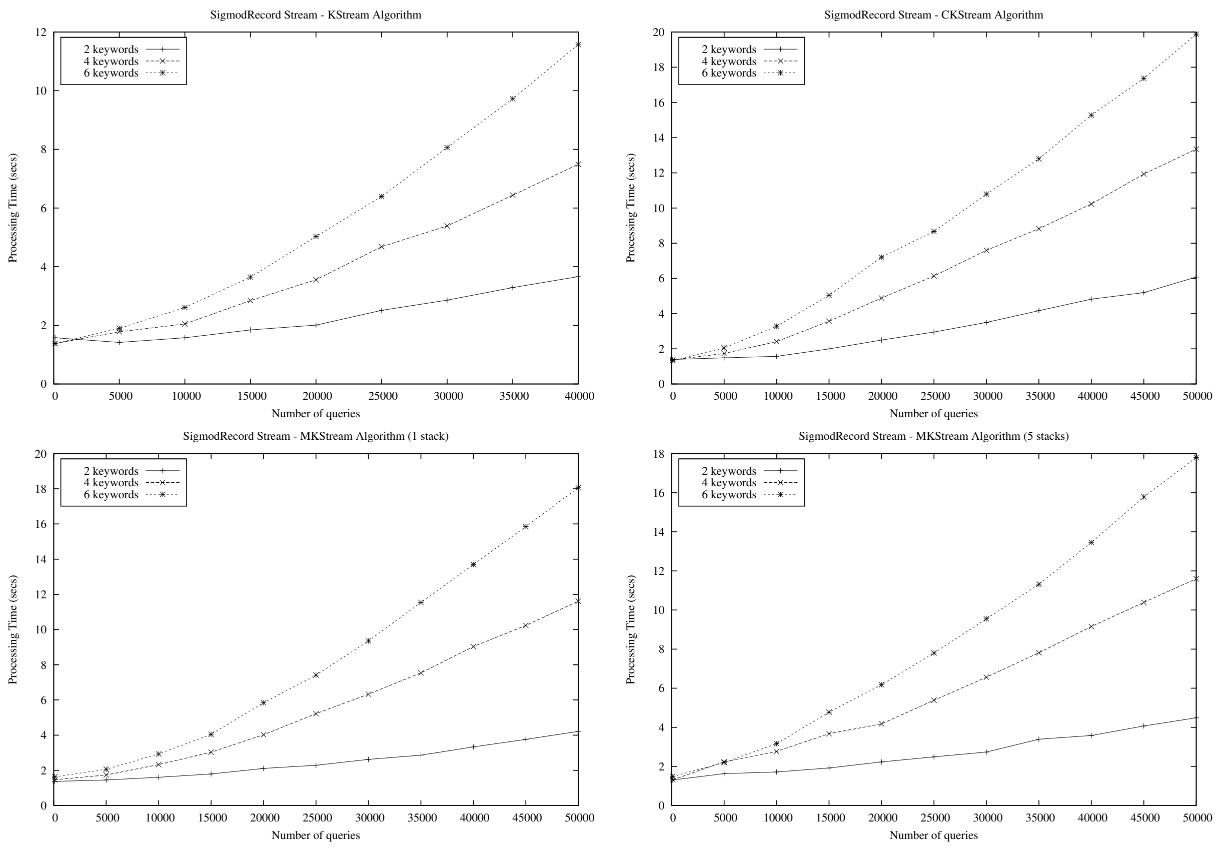


Figure 3.19. Queries with 2, 4 and 6 terms: proc. time for SIGMOD dataset.

Stream performs better than KStream. Specifically, this improvement is higher when MKStream uses five stacks. For the two other datasets, which have far fewer nodes, MKStream shows no expressive performance gain by using more than one stack.

3.5.2.4 Remarks

The experiment results indicate that BStream algorithm has a prohibitive processing time compared with MKStream, regardless of its reasonable memory performance. In scenarios where few submitted queries are evaluated, MKStream is slightly faster than the state-of-the-art algorithms and uses less memory than them. In scenarios where all queries are evaluated, independently from the XML document size, MKStream is significantly better than the state-of-the-art algorithms.

Barros et al. [2012a] present the complexity analysis for BStream and MKStream algorithms in detail. Specifically, BStream time complexity is $\mathcal{O}(N \times Q \times T)$, being N the number of nodes in the XML document, Q the number of queries and T the number of distinct terms that occur in the queries. MKStream time complexity is $\mathcal{O}(N \times Q)$, being N and Q linear factors.

In conclusion, MKStream offers the best compromise between performance and

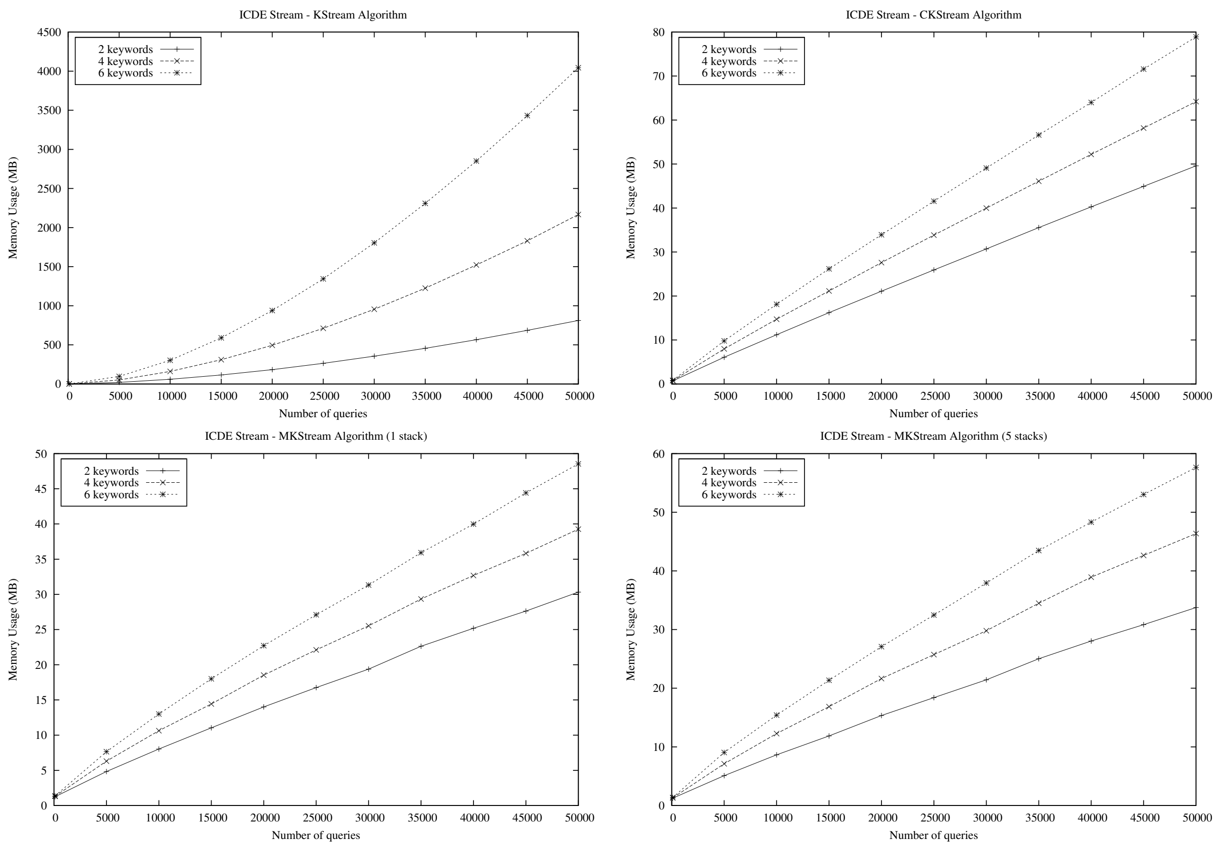


Figure 3.20. Queries with 2, 4 and 6 terms: memory used for the ICDE dataset.

memory usage. It is faster than the state-of-the-art algorithms when the number of evaluated queries increases, independently of the XML document size. Particularly, MKStream allows adjusting the number of parsing stacks for a better trade-off between processing time and memory usage.

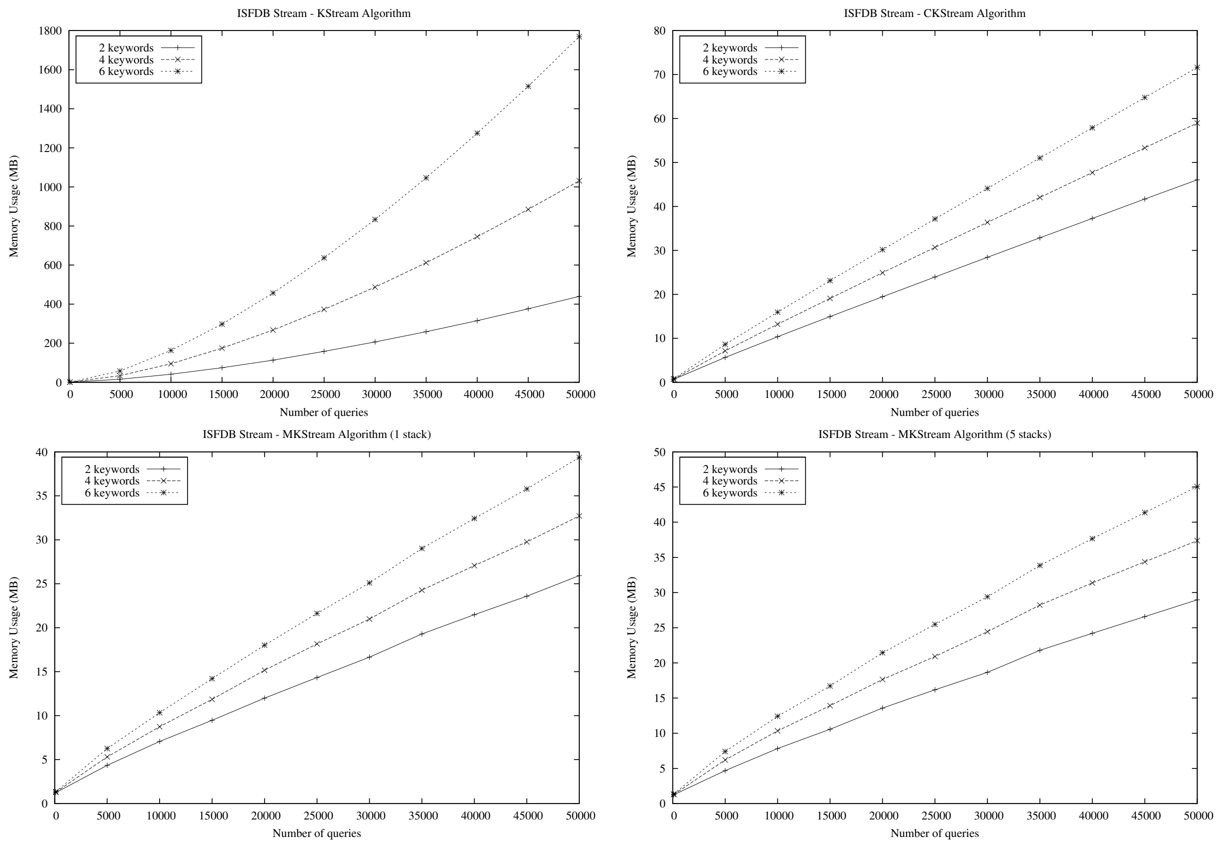


Figure 3.21. Queries with 2, 4 and 6 terms: memory used for ISFDB dataset.

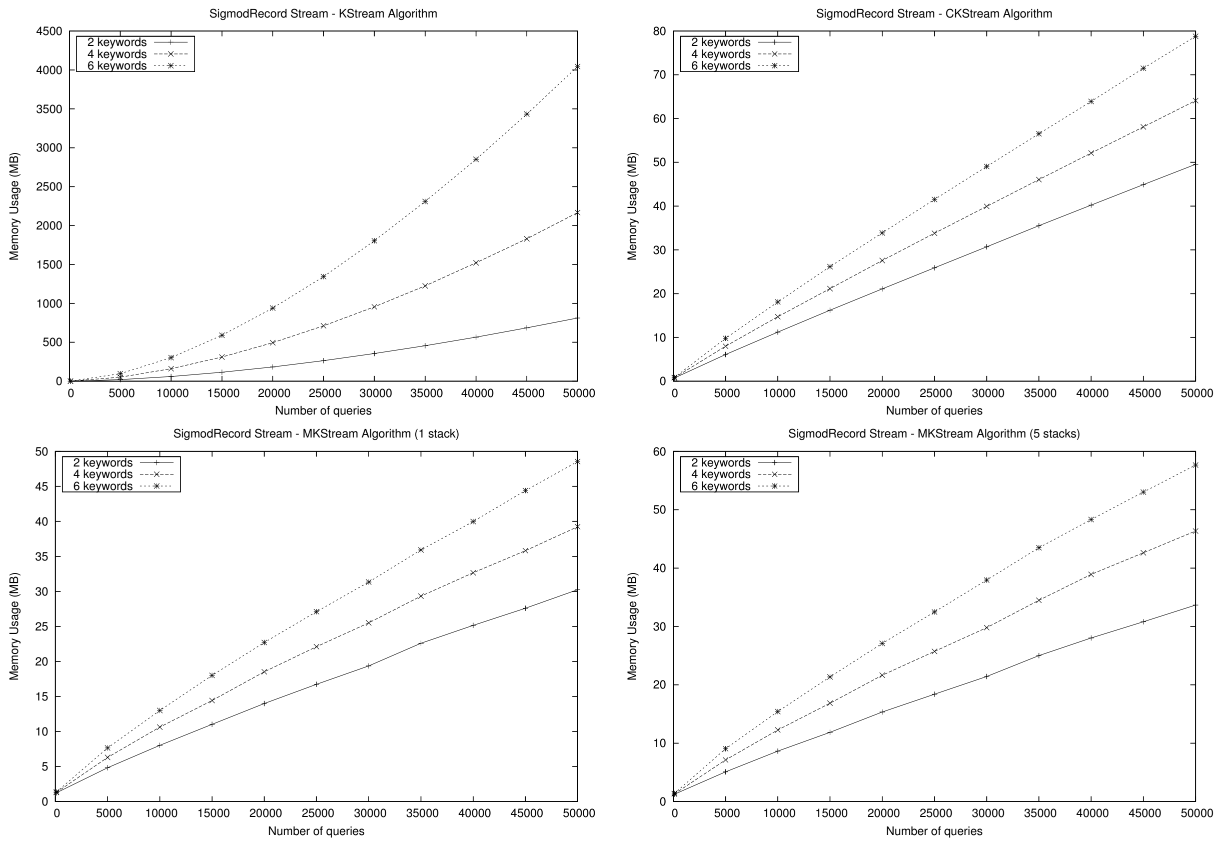


Figure 3.22. Queries with 2, 4 and 6 terms: memory used for SIGMOD dataset.

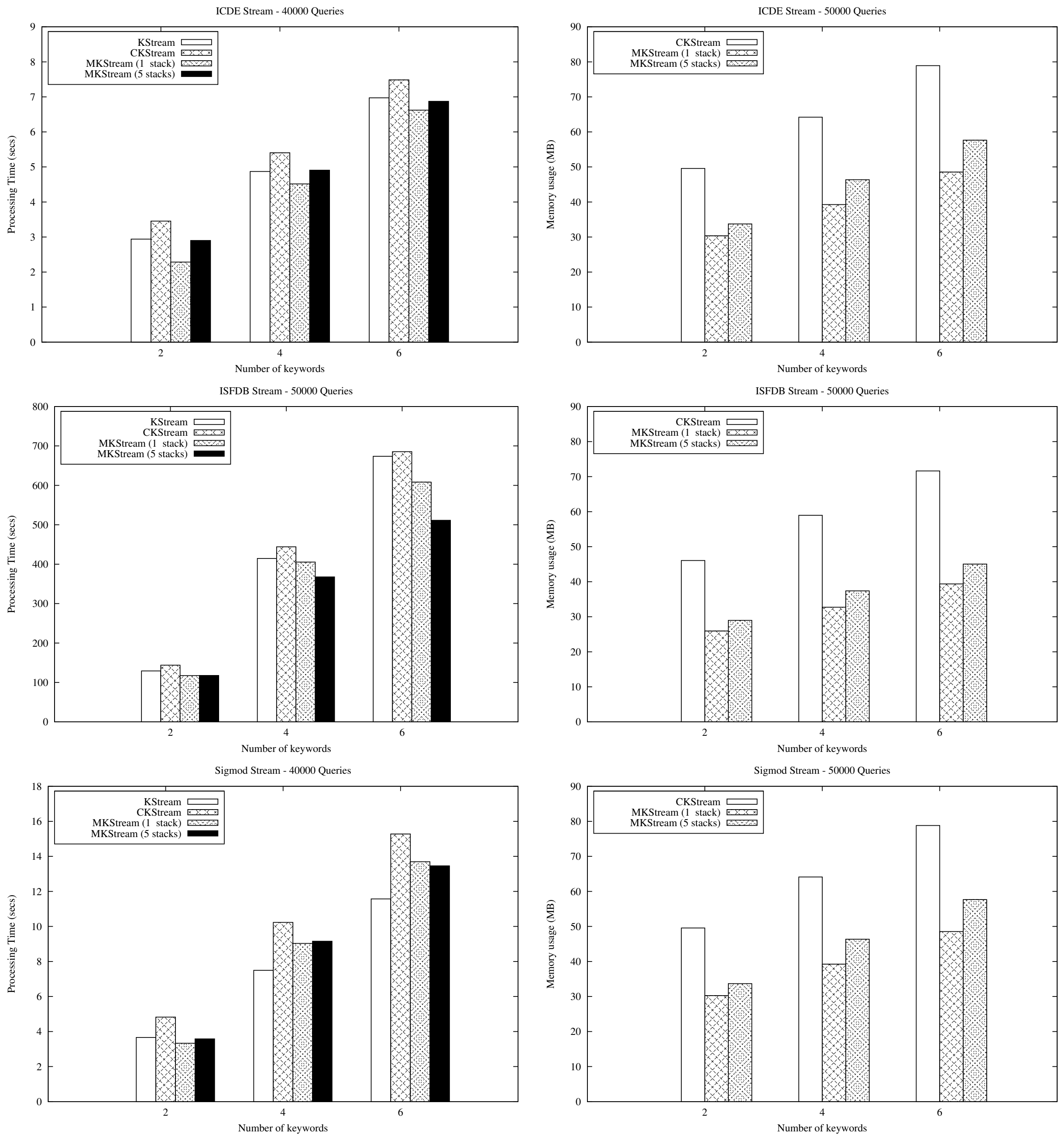


Figure 3.23. Varying the Number of Terms for up to 50,000 queries: response time (left) and memory usage (right).

Chapter 4

An SLCA Algorithm based on Stream Processing Properties

This chapter presents an SLCA-based algorithm, called SLCAStream, for efficient processing of multiple keyword-based queries over XML streams. This algorithm proposes a new approach to keyword-based query processing which is based on LCA stream processing properties. As a consequence, this approach eliminates the traditional bitmap processing strategy [Barros et al., 2010, Hummel et al., 2011, Vagena and Moro, 2008, Zhou et al., 2010]. SLCAStream also introduces optimization strategies that improve its overall performance. This approach and its optimizations allow to improve the algorithm response time and memory consumption when compared to MKStream (our better algorithm based on the traditional bitmap strategy presented in Chapter 3).

This chapter is organized as follows. Section 4.1 presents the stream processing properties based on the LCA semantics for SLCA node identification. Section 4.2 presents SLCAStream, its data structures and callback functions. Then, Section 4.3 describes the experiment we conducted, which compares SLCAStream with MKStream, in terms of response time and memory usage.

4.1 SLCA Stream 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 as described in Section 3.3.1. Here, we are particularly interested in the *startElement*() event. 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 Figure 4.1).

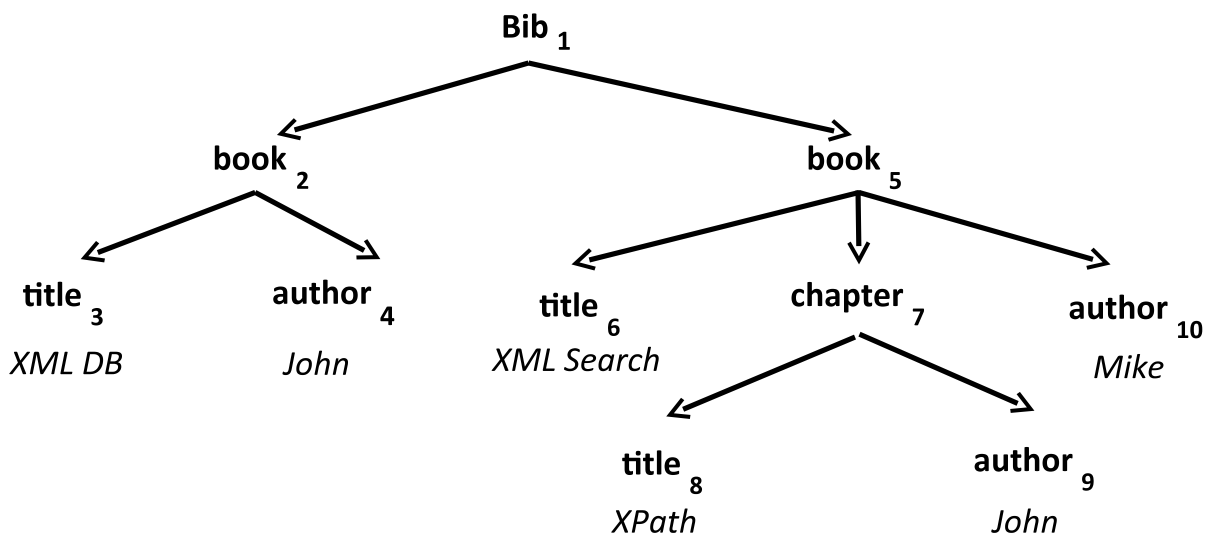


Figure 4.1. Tree representation of an XML document.

The *id* values have important properties for the SLCA semantics, as discussed and exemplified next. The examples are based on the query q={title, author} and the XML document in Figure 4.1. Although we discuss these properties considering a single query, they equally apply to multiple queries.

Property 1. *Given an XML document d, the id of any node v in d is smaller than the id of any of its descedants and greater than the id of all its previously processed nodes (see Figure 4.1).*

Property 2. *Let v be an SLCA node returned as a result for the query $q=\{t_1, \ldots, t_n\}$. Then, id_v is less than or equal to the id of its descendant nodes that satisfy a term t_i $(1 \leq i \leq n)$ and greater than the id_u of the previous SLCA node u that satisfies q.*

Property 2 guarantees that if node v satisfies query q, this node or some of its descendants satisfies all query terms, meaning that its *id* is less than or equal to the *id* of each node that satisfies a query term. Notice that Property 2 also guarantees the previous u SLCA node has been processed before v since id_u is less than id_v. Thus, v satisfies the SLCA semantics. In Figure 4.1, $chapter_7$ satisfies Property 2 since its *id* is less than or equal to the *ids* of its descendants that match query q terms. In addition, the *id* of node $chapter_7$ is greater than the *id* of the previous SLCA node that satisfies q, which is $book_2$.

4.2 Proposed Algorithm

In this section, we present SLCAStream, a new algorithm for multiple keyword-based query processing that implements Property 2 and incorporates optimization techniques to improve query evaluation performance. This property allows to identify SLCA nodes based only on their *id* values, thus avoiding the usual bitmap processing strategy. In addition, SLCAStream uses a single parsing stack for query evaluation. In what follows, we describe SLCAStream, including its data structures.

4.2.1 Data Structures

4.2.1.1 Parsing Stack

Like BStream and MKStream, SLCAStream uses a parsing stack S for keeping the XML nodes open during the SAX parsing. Each entry is popped from the stack when its corresponding node and all its descendants have been visited. However, in contrast to BStream and MKStream, by using LCA stream processing properties, SLCAStream eliminates all data structures that support the bitmap processing strategy, such as the query and CAN_BE_SLCA bitmaps, both used by BStream and MKStream parsing stacks. Consequently, SLCAStream eliminates all workload related to the bitmap processing strategy. However, SLCAStream includes a new field matching_terms in each parsing stack entry. This field is used by the proposed optimization techniques, thus allowing SLCAStream to evaluate fewer queries than BStream and MKStream, as described as follows. For SLCAStream, the parsing stack S includes the following information concerning its final configuration:

- The XML node label;
- A set used_queries containing the queries whose terms match keywords in the node or any its descendants, either as labels or values;
- a matched_terms field that stores the number of matching query terms on the node being processed or on its descendants. Upon closing this node, SLCAStream only evaluates the queries whose number of terms is less than or equal to this field. This constitutes one of the two optimization techniques that contribute to improve SLCAStream response time. To better illustrate this, consider the query $q = \{title, author\}$ and the XML document in Figure 4.1. At node $title_3$ opening, the $matched\_terms$ is 1 since keyword *title* was found in $title_3$ as its label. At $title_3$ closing, the algorithm skips query q evaluation since the node only matches a single q term (number of q terms is greater than the current $matched\_terms$ value).

Similarly, at $author_4$ closing, the algorithm skips q evaluation. However, the node $book_2$ has two descendants that partially or totally match q terms. Therefore, its $matched\_terms$ value is 2 (due to $title_3$ and $author_4$). Thus, when closing $book_2$, since the number of q terms is equal to the current $matched\_terms$ value, the algorithm evaluates each q term individually since $book_2$ descendants can have some repeated term among them also added to $matched\_terms$[1]. Note that this optimization has no effect on the SLCA semantics since SLCAStream guarantees its compliance.

4.2.1.2 Query Index

Similar to BStream, SLCAStream also relies 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. Thus, its query_index is used to look for queries matching text elements or labels.

4.2.1.3 Hash Table

As shown in Section 4.1, by only knowing the nodes that satisfy the query terms, it is possible to establish if a candidate node is SLCA. For this, during a document processing, SLCAStream requires a global hash table G to store the *ids* of query term occurrences. Its keys represent all distinct query terms and the respective values correspond to the *id* of the last query term occurrences. For instance, consider the XML document in Figure 4.1 and the query q=$\{title, author\}$. When closing node $chapter_7$, the *id* for the last *title* occurrence is $G[title]$=$\{8\}$ and for *author* is $G[author]$=$\{9\}$. Thus, according to Property 2, $chapter_7$ is SLCA for q since its *id* (7) is less than the last *id* occurrences $(8, 9)$ of the q terms and greater than the previous SLCA result node *id* (id_{book_2}=2).

4.2.2 SLCAStream

As already mentioned, SLCAStream presents a new strategy for SLCA evaluation which avoids the usual bitmap processing. SLCAStream is based on the SLCA semantic Property 3, as presented in Subsection 4.1. In addition, it uses two optimization techniques to improve time performance and memory consumption, which are described next.

[1]We consider there is no term repetition in queries.

SLCAStream consists of three callback functions, each one corresponding to a distinct SAX parser event. Function SLCAStream.Start handles *starElement* events, SLCAStream.Text handles *characters* events and SLCAStream.End handles *endElement* events. These functions are described by Algorithms 8, 9 and 10 respectively.

Algorithm 8 SLCAStream.Start Callback Function

```
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.  $\ell$ := label($e$)
2.  $id := id + 1$ {next $id$ for the new node}
3.  node_path[$e.height$].id=$id$
4.  if query_index[$\ell$] $\neq \emptyset$ or query_index[$\ell$::] $\neq \emptyset$ then
5.    sn.id := $id$
6.    sn.height := $e.height$
7.    terms := {$\ell$, $\ell$::}
8.    for all $t \in$ terms do
9.      $Q$ := query_index[$t$] {$Q$ get queries of the term $t$}
10.     if $Q \neq \emptyset$ then
11.       sn.used_queries := sn.used_queries $\cup$ $Q$
12.       $G[t] := id$
13.       sn.matched_terms = sn.matched_terms + 1
14.     end if
15.   end for
16.   S.push(sn) {create new stack entry}
17. end if
```

SLCAStream.Start sets label ℓ to the corresponding label of the node e being processed (Line 1) and generates the *id* value for this node (Line 2). SLCAStream.Start also registers on the node_path array the *id* value of the current node e (Line 3), which corresponds to the last node on the path to the root. We use this array for implementing an optimization technique in the SLCAStream.End function. Next, it checks whether label ℓ occurs in some query term (Line 4). If so, SLCAStream.Start initializes the sn stack entry (Lines 5 and 6), which is pushed to S (Line 16). This entry corresponds to the node being processed.

Similar to MKStream, SLCAStream reduces stack operations because it pushes entries to the stack only if necessary. However, SLCAStream further reduces the entries in the parsing stack since it only keeps the nodes requiring evaluation. This is our first optimization technique which improves response time and memory usage. SLCAStream.Start gets the queries in query_index that contain ℓ or ℓ:: terms and stores them in the set Q (Line 9). If Q is not empty, SLCAStream.Start includes them in the

Algorithm 9 SLCAStream.Text Callback Function.

```
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.  $\ell$ := label($e$)
2.  new_stack_entry := $false$
3.  if S.height = $e.height$ then
4.     sn := *S.top() {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.  $K$ := set of tokens in node $e$
10. for all $k \in K$ do
11.    terms := {$k$, ::$k$, $\ell$::$k$} {possible terms containing $k$}
12.    for all $t \in$ terms do
13.       $Q$ := query_index[$t$] {$Q$ get queries of the term $t$}
14.       if $Q \neq \emptyset$ then
15.          new_stack_entry := $true$
16.          $G[t]$ := $id$
17.          sn.used_queries.add($Q$)
18.          sn.matched_terms = sn.matched_terms + 1
19.       end if
20.    end for
21. end for
22. if new_stack_entry then
23.    S.push(sn) {create new stack entry}
24. end if
```

used_query set of the sn entry (Line 11). It also adds id to the global hash table G, marking the corresponding node as a query term occurrence for ℓ or ℓ:: terms (Line 12). Finally, it also records in the sn.matching_terms field that the ℓ or ℓ:: term occurred in node e (Line 13). Upon closing this node, SLCAStream.End only evaluates the queries whose number of terms is less than or equal to this field (Line 13). This constitutes our second optimization technique, which improves response time.

SLCAStream.Text processes tokens found in the text of the node e being processed. Differently from SLCAStream.Start, it considers all possible terms containing text tokens, which are k, ::k, and ℓ::k, being ℓ the label of node e (Line 11) and k a text token. Like SLCAStream.Start, SLCAStream.Text only pushes entries down the stack if necessary. It only pushes node e down the stack if its corresponding entry sn is not on the stack top. If node e is the top node, sn references it (Line 4). Otherwise,

Algorithm 10 SLCAStream.End Callback Function

```
Callback Function SLCAStream.End
Input: The parsing stack S
Input: The XML node $e$ that is ending
if S.height = $e.height$ then
   sn := S.pop() {pops the top entry in the stack to sn}
   $id$ := sn.id
   $\ell$ := sn.label
   for all $q \in$ sn.used_queries do
      if sn.matched_terms $\geq$ $q$.terms.size() then
         COMPLETE := $true$
         for all $t \in q$.terms do
            if $id < G[t]$ then
               COMPLETE:=$false$
               break
            end if
         end for
         if COMPLETE then
            {checks if $id$ is an SLCA result}
            if $id > q$.last_result_id then
               $q$.results.add($sn$)
               $q$.last_result_id := $id$
            end if
         end if
      end if
   end for
   tn := *S.top() {tn points to the top entry in the current stack}
   if (sn.height - tn.height) = 1 then
      tn.used_queries.add(sn.used_queries)
      tn.matched_terms = tn.matched_terms + sn.matched_terms
   else
      sn.height = sn.height - 1
      sn.id = node_path[$e.height$-1].$id$
      S.push(sn)
   end if
end if
```

SLCAStream.Text initializes the stack entry sn (Lines 6 and 7), which is pushed into S (Line 23) if the possible terms match a query. SLCAStream.Text knows that e is on the top when the stack height equals S height (Line 3).

SLCAStream.Text pushes only the necessary entries to the stack, i.e., only those nodes e whose entry sn is not on the stack top. In addition, such nodes must have queries to be evaluated. Furthermore, SLCAStream.Text inserts the id of a node e in the global hash table G for term t if e matches t (Line 16). It also stores the queries that match e (Line 17) on the top entry sn and updates the number of terms that match any query (Line 18).

Finally, SLCAStream.End evaluates which nodes or their descendants match the submitted queries, following the SLCA semantics. For this, it only pops the stack top entry sn, if sn corresponds to node e (Line 2). After that, SLCAStream.End only evaluates queries stored in the used_queries set whose number of terms is less than or equal to the sn.matched_terms value (Line 6). This value corresponds to the number of occurrences of query terms in e or its descendants. For the current query q, if this value is less than the number of q terms, it means that e has insufficient contributing nodes to satisfy q. Otherwise, SLCAStream.End evaluates whether e or its descendants have occurrences for all q terms (Lines 8 to 13). If e satisfies Q completely, SLCAStream.End must evaluate whether it is an SLCA node. In addition, according to Property 2, to be considered SLCA, e must have its *id* greater than the previous SLCA node *id* for q (Line 16). If e is an SLCA node, SLCAStream.End adds it to its result list (Line 17) and records its *id* as the last SLCA result id (Line 18). At this point, SLCAStream parsing stack stores the nodes that require being evaluated up to the root node. Thus, SLCAStream.End must update the parsing stack correctly. If the new top node tn is parent of sn (Line 24), SLCAStream.End adds sn queries to tn (Line 25) and also adds the number of matching terms up to sn (Line 26). If tn is not the parent of sn, sn is pushed to the stack again (Line 30). In this case, sn becomes the previous node from e. However, its *id* and height must be changed. SLCAStream.End decreases sn height by 1 (Line 28) and updates its *id* to the previous *id* (Line 29) by using the array node_path. This array maintains all nodes on the path from e to the root and is updated by SLCAStream.Start and SLCAStream.Text functions.

4.3 Experimental Evaluation

Similar to the performance evaluation conducted in Chapter 3, SLCAStream is experimentally evaluated in terms of processing time and memory space. The experiments compare SLCAStream with MKStream in its five-stack configuration, which outperformed the stated-of-the-art algorithms as shown in Chapter 3. Thus, the experiments confront the LCA stream processing property strategy of SLCAStream with the bitmap processing strategy of MKStream.

Like in Chapter 3, we performed three experiments with streams containing documents from real XML datasets, each one focusing on a different aspect. The first

Dataset	Docs	Avg. Height	Num. of # Elem.	Avg. Nodes	Avg. Objects	Avg. Size
SIGMODR	18	7	13	1697	167	58 KB
XMARK	3	5	74	10312	5515	280 KB
ISFDB	10	2	11	41637	4110	1.3 MB

Table 4.1. Details of the datasets used for the experiments.

experiment analyzes the impact of increasing the number of pure keyword queries. The second experiment analyzes the algorithm performance by using structural keyword queries. The third experiment evaluates SLCAStream scalability by varying the number of query terms. All experiments also exclude the time spent for creating the query indexes. For memory usage, we also consider the average memory used during XML document processing, including the memory used by index structures.

4.3.1 Setup

In this section, we describe the experimental evaluation, its datasets, queries and metrics. All algorithms used in this evaluation were implemented in Java using 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.

Dataset. Similar to Chapter 3, we also used the SIGMOD and ISFDB datasets but we replaced the ICDE dataset by XMARK because the ICDE and SIGMOD datasets have similar sizes. In addition, the accuracy evaluation, presented in this chapter, is based on the XMARK dataset. Table 4.1 presents the dataset details. According to this table, XMARK contains more objects than the other two datasets. Note that the datasets have different sizes.

Queries. We used the same random queries generated from the SIGMOD and ISFDB datasets for the experimental evaluation in Chapter 3. However, we replaced the ICDE queries used in that chapter by XMARK queries, which were also randomly generated according to the three specific experiments.

Metrics. Similar to Chapter 3, we measured, for each dataset and experiment, the time spent for processing all XML documents on a given stream. We also excluded the time spent to create the query indexes. Regarding memory consumption, we measured the average memory usage while processing each XML document.

4.3.2 Results

4.3.2.1 Varying the Number of Queries

The first experiment aims at analyzing the time response as the number of keyword-based queries increases. It considers random queries, each one with up to four terms. Each query term is of the form *::k*.

Figure 4.2 (left) compares time spent by SLCAStream and MKStream with five stacks, over each dataset. The time curves show a clear SLCAStream advantage in the three datasets due to a more efficient SLCA semantics implementation since it requires no bitmap, reduces stack operations and evaluates only plausible queries.

Regarding memory consumption, Figure 4.2 (right) shows that SLCAStream saves more memory when compared with MKStream since it requires no bitmap and reduces stack memory usage. Although MKStream reduces stack operations and, consequently, memory usage, SLCAStream further reduces memory usage since it only keeps the nodes requiring evaluation.

4.3.2.2 Searching with Structural Constraints

The second experiment analyzes the impact of using structural terms, such as $\ell{::}k$. The experiment considers 50,000 queries with 5 query terms each and varies the number of structural query terms from 0 to 3. Queries containing 1, 2 or 3 structural terms are always evaluated because at least one of their terms match some of the labels in the documents. Thus, all 50,000 queries containing 1, 2 or 3 structural terms are evaluated, representing the worst case scenario.

Notice that MKStream usually processes multiple stacks, thus evaluating fewer queries than SLCAStream. However, as presented in Figure 4.3 (left), SLCAStream has a significant performance gain, because it requires no bitmap operations, reduces further stack operations and evaluates only plausible queries.

Regarding memory usage, Figure 4.3 (right) shows that SLCAStream saves slightly more memory for the same reasons pointed out in the previous experiment.

4.3.2.3 Varying the Number of Terms

This experiment evaluates the scalability of MKStream (in its five-stack configuration) and SLCAStream 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*.

Figure 4.4 presents MKStream (left) and SLCAStream (right) response times. As expected, increasing the number of terms affects the performance of the algorithms.

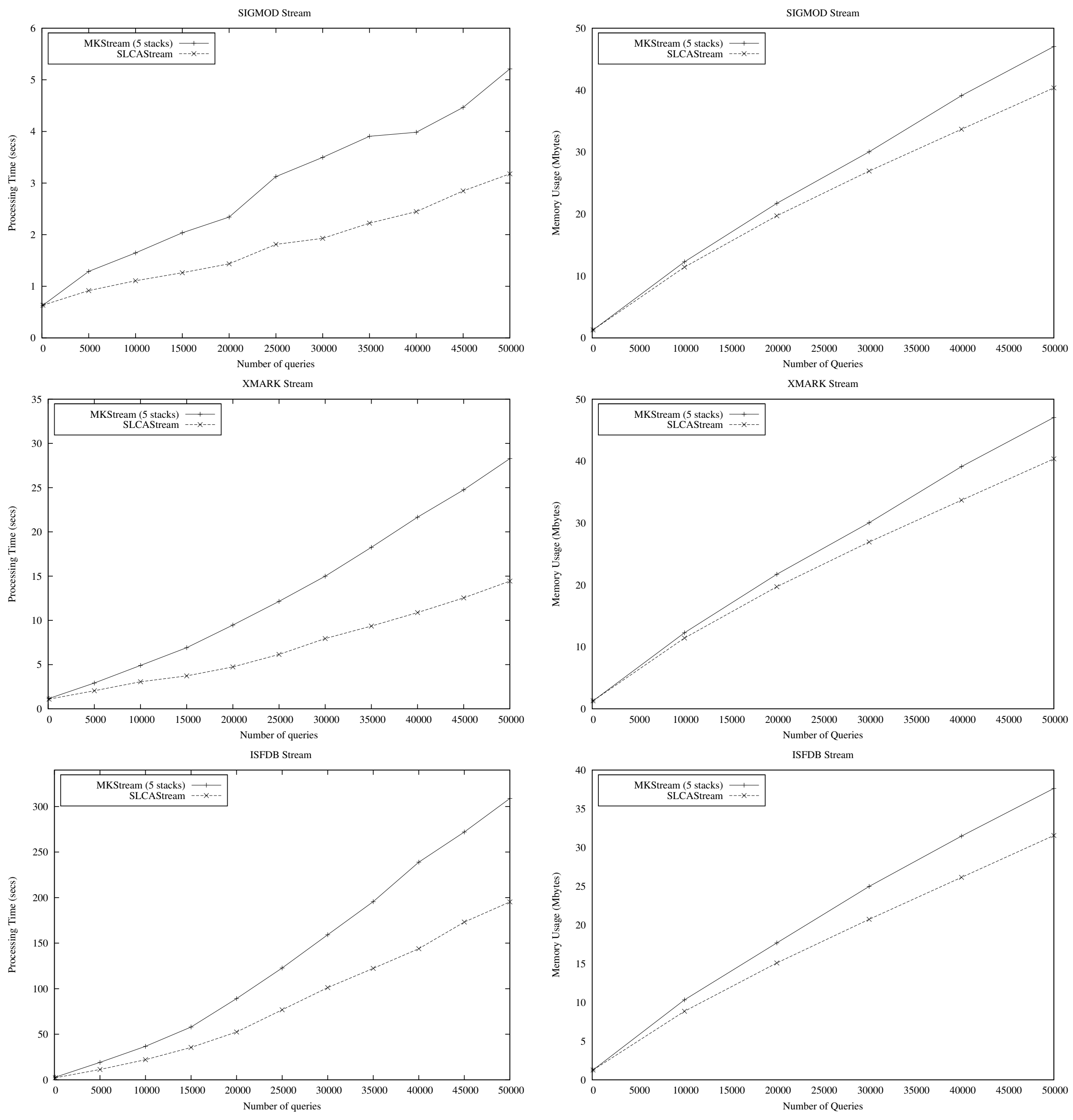


Figure 4.2. First performance experiment: response time (left) and memory usage (right) for MKStream (5 stacks) and SLCAStream in the three datasets.

However, SLCAStream performs faster than MKStream (left graph) in all datasets due to its optimization techniques and absence of bitmap processing.

Regarding memory usage, Figure 4.5 (right) shows that SLCAStream uses slightly less memory. As mentioned in the previous experiment, this memory saving is due to not using bitmap structures and keeping a reduced number of stack entries.

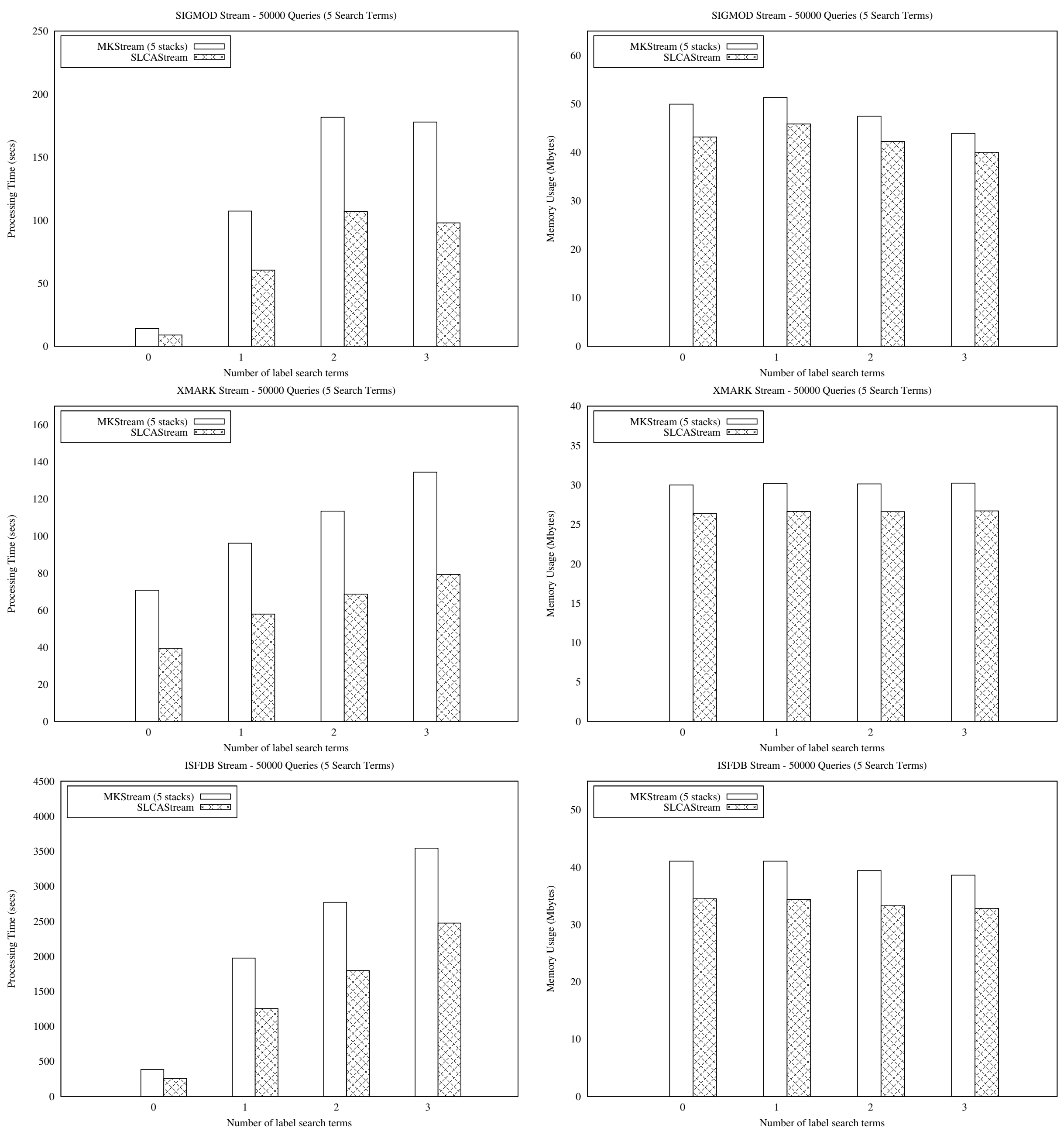


Figure 4.3. Second performance experiment: response time (left) and memory usage (right) for MKStream (5 stacks) and SLCAStream in the three datasets.

4.3.2.4 Remarks

The experiment results indicate that SLCAStream improves response time significantly when compared with MKStream, since it requires no bitmap operations, further reduces stack entries and evaluates only plausible queries. For the same reasons, SLCAStream spends less memory than MKStream.

In conclusion, SLCAStream offers the best compromise between performance and

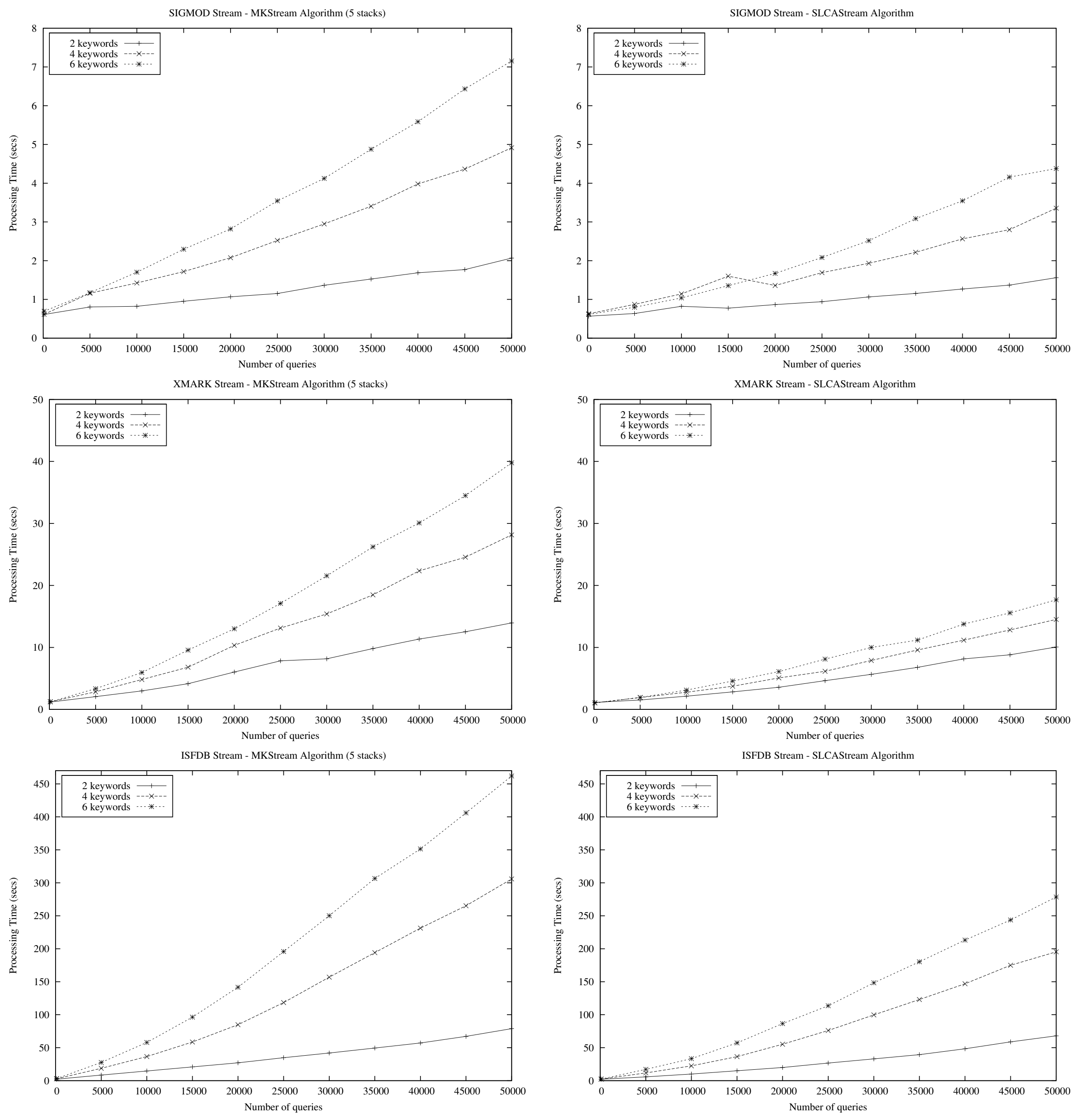


Figure 4.4. Third performance experiment: response time for MKStream (5 stacks) (left) and SLCAStream (right) algorithms in the three datasets.

memory usage. Particularly, it uses optimization techniques and explores a new strategy for SLCA evaluation that avoids the usual bitmap processing. These techniques and the new SLCA evaluation strategy improve SLCAStream overall performance.

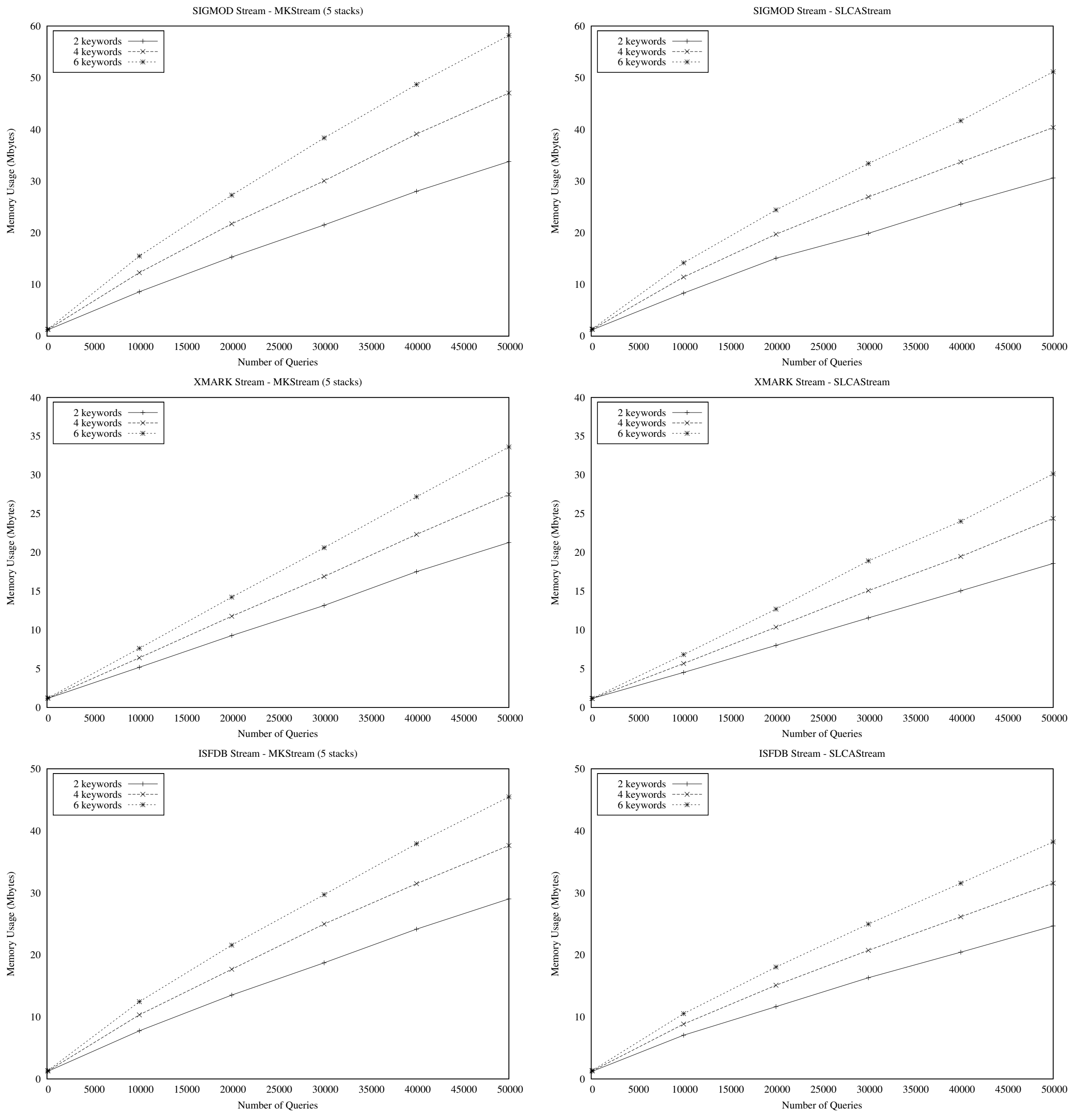


Figure 4.5. Third performance experiment: memory usage for MKStream (5 stacks) (left) and SLCAStream (right) algorithms in the three datasets.

Chapter 5

ELCA Algorithms

This chapter presents the algorithms ELCABStream and ELCAStream for processing multiple keyword-based queries over XML streams. Both are based on the ELCA (Exclusive LCA) semantics [Guo et al., 2003]. ELCABStream is a basic ELCA implementation obtained from MKStream. However, it uses a single bitmap and a single parsing stack for query evaluation. ELCAStream enhances ELCABStream by using LCA stream processing properties instead of bitmaps for query evaluation. This chapter also compares both algorithms though an experimental evaluation concerning response time and memory usage.

This chapter is organized as follows. Section 5.1 presents the stream processing properties based on the LCA semantics that identify ELCA results without using traditional bitmaps. Section 5.2 presents the algorithms ELCABStream and ELCAStream, their data structures and callback functions. Then, Section 5.3 presents the experimental evaluation which compares both algorithms and two baselines in terms of response time and memory usage.

5.1 ELCA Stream Processing Properties

In Section 4.1, we defined the SLCA semantics by means of LCA stream processing properties. Likewise, the ELCA semantics can also be defined by similar properties. Next, we define the ELCA semantics and its corresponding stream processing properties. Although we discuss these properties considering a single query, they equally apply to multiple queries.

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,* *directly or indirectly, at least one occurrence of each query term as their own node, i.e., all LCA descendent nodes that also match the query terms are disconsidered.*

As an example, consider the subtree rooted at node $book_5$ in the XML document in Figure 4.1 and the query q=$\{title, author\}$. The LCA nodes for query q are $chapter_7$ and $book_5$ since their descendant labels match the keywords *title* and *author*. Following this LCA example, the ELCA nodes are $chapter_7$, which satisfies both query terms and has no descendants, and $book_5$, which, excluding its LCA descendant subtree $chapter_7$, also satisfies the query terms. Notice that $chapter_7$ is also SLCA because it contains no LCA descendants. Node Bib_1, on the other hand, 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, which means that it includes as part of a query result all SLCA nodes that match the query terms. Thus, we start identifying ELCA nodes by identifying SLCA nodes, which are also LCA nodes. Notice that only ELCA handles the ambiguity that might exist among keywords [Bao et al., 2009]. As a consequence, ELCA is more comprehensive than SLCA, thus producing more query results. This makes ELCA a more appropriate semantics than SLCA when querying an XML document that contains the same content at different levels. For example, in Figure 4.1, *title* occurs as *book* child and *chapter* child.

Property 3. *Let q be a query consisting of the set of terms $\{t_1, t_2, \ldots, t_n\}$ and IdList[t_i] ($t_i \in q$) be the id list of nodes where t_i occurs in LCA descendants of node v. For each term t_i, if there is at least one node u, where u is a v descendant or equal to v, that satisfies t_i and id_u does not occur in IdList[t_i], then v is an ELCA node.*

According to Property 3, v is an ELCA node for a query q 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 3, consider v=$book_5$. When closing this node, its descendants that satisfy the query terms are respectively $title_6$ and $author_{10}$. Notice that these two nodes do not occur in node $chapter_7$, which is an LCA descendant of $book_5$. Thus, $book_5$ is an ELCA node.

5.2 Proposed Algorithms

The ELCABStream and ELCAStream algorithms evaluate multiple keyword-based queries over XML streams and are completely based on Property 3. In other words, both algorithms consider a node v an ELCA result if v has at least one occurrence for each query term, excluding term occurrences that contribute to its LCA descendant nodes.

As already mentioned, SLCA nodes are also ELCA nodes. Thus, ELCABStream and ELCAStream begin an ELCA node evaluation by processing SLCA nodes. Specifically, they identify part of the ELCA nodes by using SLCA processing strategies. The remaining ELCA nodes are identified by applying the ELCA stream processing properties (e.g., Property 3). Thus, both algorithms use SLCA processing strategies and ELCA stream processing properties.

Specifically, ELCABStream extends MKStream. However, it uses only a single parsing stack, while MKStream may use multiple stacks. Since ELCABStream uses a bitmap processing strategy, we consider it as a baseline for the ELCA semantics. On the other hand, ELCAStream extends SLCAStream by applying the ELCA stream processing properties, which avoids using bitmap processing strategies. In addition, it incorporates SLCAStream optimization techniques.

5.2.1 Data Structures

5.2.1.1 Parsing Stack

Like other algorithms, ELCABStream and ELCAStream use a parsing stack S for keeping the XML nodes open during the SAX parser. Each entry is popped from the stack when its corresponding node and all its descendants have been visited. However, in contrast to ELCABStream, ELCAStream eliminates the query_bitmap used by the parsing stack S since it evaluates SLCA semantics based on LCA stream processing properties. ELCAStream also eliminates the bitmap CAN_BE_SLCA used by ELCABStream. Thus, in ELCAStream, each S entry keeps the same information contained in the SLCAStream parsing stack, presented in Section 4.2.1.1. This information includes the XML node label and the used_queries set. ELCAStream, however, also includes the matched_terms field, which is used in one of two optimization techniques.

5.2.1.2 Query Index

Like MKStream, in ELCABStream the query_index structure stores the position of the bit corresponding to query terms in query_bitmap. ELCABStream also uses the index auxiliary_index to speed up the search for queries that contain a certain query term. Similar to MKStream, the auxiliary_index in ELCABStream is implemented using an inverted list where each index entry represents a query term and refers to the queries in which this term occurs.

Similar to BStream, ELCAStream uses its `query_index` only to look for queries matching text elements or labels since this avoids the use of bitmaps. In this index, each entry represents a query term and refers to queries in which this term occurs.

5.2.1.3 Inverted Lists

As shown in Section 5.1, by knowing only the nodes where the query terms occur, it is possible to establish if a candidate node is ELCA. For this, ELCABStream and ELCAStream require inverted lists on query term occurrences during a document processing. Specifically, they use two types of inverted list. The first one is the global inverted list G. Its entries represent the query terms that match the processed nodes and its values correspond to the *id* of such nodes. For instance, consider the XML document in Figure 4.1 and the query q=$\{title, author\}$. When processing the node $title_8$, the *title* inverted list is $G[title]$=$\{3, 6, 8\}$ and the *author* inverted list is $G[author]$=$\{4\}$. Notice that ELCABStream and ELCAStream replace the hash table G used by SLCAStream (see Section 4.2.1.3) by the inverted list G, since ELCA evaluation requires all *ids* that correspond to nodes that satisfy query term occurrences, not only the last ones.

The second type of inverted list is built for each query q=$\{t_1, t_2, \ldots, t_n\}$, where $q \in Q$, being Q the set of all queries. The q inverted list entries represent the q terms $t_1, t_2, \ldots, t_n$ and its values correspond to node *ids* that match q up to the node being processed that contributes to an LCA result. Following our example, after closing $chapter_7$, the q inverted list for *title* is $q[title]$=$\{3, 8\}$ since these values correspond respectively to nodes that contribute to $book_2$ and $chapter_7$ SLCA results, which are also LCA results for q. When closing $chapter_7$, the q inverted list for *author* is $q[author]$=$\{4, 9\}$. Similarly, considering the *author* term, these values correspond to nodes that contribute to $book_2$ and $chapter_7$ SLCA results, which are also LCA results for q.

5.2.2 Using Inverted Lists to Identify ELCA Nodes

For most ELCA node evaluations, Property 3 can be directly assessed by using the global and query term inverted lists. However, we can start evaluating an ELCA node by using a faster and more practical strategy, i.e., by starting with SLCA nodes, which are also ELCA nodes. For this reason, ELCABStream and ELCAStream use the SLCA evaluation strategies of MKStream and SLCAStream respectively. Before presenting ELCABStream and ELCAStream, we describe how they use the inverted lists to evaluate ELCA nodes.

The inverted lists keep the node *id* values in ascendent order as they are added to them. Such a feature provides an essential property for evaluating ELCA nodes. Using the global inverted list G and applying Property 1 (see Section 4.1) it is possible to identify the node occurrences for a term t that are descendant nodes of the v node being closed (all descendant nodes have already been closed). Thus, a search in $G[t]$ inverted list 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 Figure 4.1 and the query q=$\{title, author\}$. When closing $book_5$ (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 $\{9, 10\}$.

The q inverted list makes possible to identify the LCA contributing nodes for term t, which are descendant nodes of the v node being closed. For instance, when closing $book_5$, a search for *title* in the q inverted list returns the set of nodes $\{8\}$ and for *author*, the set of nodes $\{9\}$.

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

5.2.3 ELCABStream

The ELCABStream algorithm consists of three callback functions, each one related to a distinct event. Function ELCABStream.Start handles *starElement* events, ELCAB-Stream.Text handles *characters* events and ELCABStream.End handles *endElement* events. These functions are described by Algorithms 11, 12 and 13 respectively.

Initially, ELCABStream.Start generates the *id* value for node e (Line 1) and sets the label (Line 2) and id of the new stack entry *sn* (Line 3), which corresponds to node e. After that, the function sets CAN_BE_SLCA (Line 5) and query_bitmap (Line 8) for all queries and pushes the entry *sn* into S (Line 9). ELCABStream.Start obtains the queries that contain ℓ or ℓ:: terms in auxiliary_index (Line 12). It also adds *id* to the global inverted list G, recording the e node *id* as a query term occurrence for ℓ or ℓ:: terms (Line 14). ELCABStream.Start also sets to *true* the bit at position i that corresponds to ℓ or ℓ:: terms in query_bitmap (Line 16). Finally, all queries containing ℓ or ℓ:: terms are added to the used_queries set (Line 17) for evaluation at node e closing.

ELCABStream.Text processes tokens found in the text of node e. Its points

Algorithm 11 ELCABStream.Start Callback Function

```
Callback Function ELCABStream.Start
Input: The parsing stack S
Input: The XML node $e$ being processed
Input: The node identification $id$ for node $e$
1.  $id := id + 1$ {next $id$ for the new node}
2.  $\ell :=$ label($e$)
3.  sn.id := $id$
4.  for all $q_i \in$ queries being processed do
5.     sn.CAN_BE_SLCA[$q_i$]:= $true$
6.  end for
7.  $N :=$ number of distinct terms in all queries being processed
8.  sn.query_bitmap[0, ..., N-1] := $false$
9.  S.push(sn) {create new stack entry}
10. terms := {$\ell$, $\ell$::}
11. for all $t \in$ terms do
12.    $Q :=$ auxilary_index[$t$] {$Q$ get queries of the term $t$}
13.    if $Q \neq \emptyset$ then
14.       $G[t]$.add($id$) {$G$ is a hash table for the last node $id$ that satisfies $t$}
15.       $i :=$ query_index[$t$].asLabel {$i$ gets the position of term $j$}
16.       sn.query_bitmap[$i$] := $true$
17.       sn.used_queries.add($Q$)
18.    end if
19. end for
```

to S top entry, whose bitmap query_bitmap will record the token occurrences in the corresponding query term bits (Line 2). Henceforth, this callback function proceeds similarly to ELCABStream.Start. However, it considers all possible terms, k, $::k$, $\ell::k$, containing text tokens, being ℓ the e node label (Line 5).

Finally, function ELCABStream.End evaluates which nodes or their descendants match the queries. This function pops the stack top entry corresponding to the node being closed (Line 1). As ELCABStream.End performs some operations on the new stack entry tn, it creates a reference to it (Line 2). ELCABStream.End also retrieves the queries from the used_queries set for ELCA evaluation (Line 3). For each query q, this function verifies if the corresponding query bits in query_bitmap are complete (Line 6). If complete, ELCABStream.End also evaluates if Endsn can be SLCA for q (Line 7). If so, this node is a q result (Line 8). The SLCA node sn is also ELCA and LCA. Thereby, ELCABStream.End adds to the inverted lists q all node id values that match q terms, thus contributing to the result sn (Lines 9 to 11). In addition, ELCABStream.End defines the sn ancestor (previous entry in the parsing stack S) as a non-SLCA node of q (Line 12). If the sn entry bitmap is complete for q but it is a non-SLCA node, it

Algorithm 12 ELCABStream.Text Callback Function

```
Callback Function ELCABStream.Text
Input: The parsing stack S
Input: The XML node $e$ being processed
1.  $\ell$ := label($e$)
2.  sn := *S.top() {sn points to the top entry in the stack}
3.  $K$ := set of tokens in node $e$
4.  for all $k \in$ text do
5.     terms := {$k$, ::$k$, $\ell$::$k$}
6.     for all $t \in$ terms do
7.        $Q$ := auxiliary_index[$t$] {$Q$ get queries of the term $t$}
8.        if $Q \neq \emptyset$ then
9.           $G$[$t$].add(sn.id) {$G$ is a hash table for the last node $id$ that satisfies $t$}
10.          $i$ := query_index[$t$] {$i$ gets the position of term $t$ in the bitmap}
11.          sn.query_bitmap[$i$] := $true$
12.          sn.used_queries.add($Q$)
13.       end if
14.    end for
15. end for
```

can be ELCA. Thus, ELCABStream.End verifies if *sn* has its own contributors for all *q* terms (Lines 16 to 24). Specifically, ELCABStream.End creates temporary, empty *id* inverted lists (*IdList*) for the *q* terms (Line 15). These lists records the *id* values in *G* greater than or equal to *sn.id* (Line 17). Additionally, ELCABStream.End removes from *id* lists all *ids* of LCA contributing nodes which are *sn* descendants. That means removing from these lists all *id* values greater than or equal to *sn.id* for the *q* terms[1]. If any *id* is empty, node *sn* is a non-ELCA. At the end, the *id* list contains a new *q* LCA contributor, which are added to its *q* inverted lists (Line 22). If all *id* lists have values, *sn* is an ELCA node (Line 26). In addition, ELCABStream.End defines the *sn* ancestor as a non-SLCA of *q* (Line 27). Finally, ELCABStream.End adds all queries from the sn used_queries set to the tn used_queries set (Line 32).

As ELCABStream.End uses bitmap structures for SLCA evaluations, it also propagates the true bits to the *tn* entry, which is an *sn* ancestor. It propagates these bits using an **or** operation between *tn* and *sn* bitmaps (Line 33). Moreover, it propagates *sn* CAN_BE_SLCA bitmap to *tn* CAN_BE_SLCA bitmap since non-SLCA nodes in *sn* also are non-SLCA nodes in *tn*. It propagates these bits by using an **or** operation between *tn* and *sn* bitmaps (Line 34).

[1]Searching for *id* values is easily done by binary searches as all inverted lists are naturally ordered.

Algorithm 13 ELCABStream.End Callback Function

```
Callback Function ELCABStream.End
Input: The parsing stack S and The XML node $e$ that is ending
1.  sn := S.pop() {pops the top entry in the stack to sn}
2.  tn := *S.top() {tn points to the top entry in the stack}
3.  for all $q \in$ sn.used_queries do
4.    let $\{j_1, \dots, j_N\}$) be the positions of the bits corresponding to terms from query
      $q$ in query_bitmap
5.    COMPLETE := sn.query_bitmap[$j_1$] and ... and sn.query_bitmap[$j_N$]
6.    if COMPLETE then
7.      if sn.CAN_BE_SLCA[$q$] then
8.        $q$.results.add(sn)
9.        for all $t \in q$.terms do
10.         $q[t]$.add($\{id_1, \dots, id_k\}$), where $id_i$ $(1 \le i \le k) \in G[t]$ and $id_i \ge$ sn.$id$
11.       end for
12.       tn.CAN_BE_SLCA[$q$]:= false
13.     else
14.       can_be_ELCA := true
15.       IdList[ $q$.terms[1], ..., $q$.terms[$q$.terms.size()] ] := $\emptyset$
16.       for all $t \in q$.terms do
17.         IdList[$t$].add($\{id_1, \dots, id_k\}$), where $id_i$ $(1 \le i \le k) \in G[t]$ and $id_i \ge$ sn.$id$
18.         IdList[$t$].remove($\{id_1, \dots, id_k\}$), where $id_i$ $(1 \le i \le k) \in q[t]$ and $id_i \ge$
            sn.$id$
19.         if IdList[$t$] = $\emptyset$ then
20.           can_be_ELCA := false
21.         else
22.           $q[t]$.add(IdList[$t$])
23.         end if
24.       end for
25.       if can_be_ELCA then
26.         $q$.results.add(sn)
27.         tn.CAN_BE_SLCA[$q$]:= false
28.       end if
29.     end if
30.   end if
31. end for
32. tn.used_queries.add(sn.used_queries)
33. tn.query_bitmap := tn.query_bitmap or tn.query_bitmap
34. tn.CAN_BE_SLCA:=tn.CAN_BE_SLCA and sn.CAN_BE_SLCA
```

5.2.4 ELCAStream

As mentioned before, ELCAStream is based on LCA stream processing properties for query evaluation. In addition, ELCAStream includes the two optimization techniques

Algorithm 14 ELCAStream.Start Callback Function

```
Callback Function ELCAStream.Start
Input: The parsing stack S
Input: The XML node $e$ being processed
Input: The node identification $id$ for node $e$
1.  $\ell$ := label($e$)
2.  $id$ := $id + 1$ {next $id$ for the new node}
3.  node_path[$e.height$].id=$id$
4.  if query_index[$\ell$] $\neq \emptyset$ or query_index[$\ell$::] $\neq \emptyset$ then
5.    sn.id := $id$
6.    sn.height := $e.height$
7.    terms := {$\ell$, $\ell$::}
8.    for all $t \in$ terms do
9.      $Q$ := query_index[$t$] {$Q$ get queries of the term $t$}
10.     if $Q \neq \emptyset$ then
11.       sn.used_queries := sn.used_queries $\cup$ $Q$
12.       $G[t]$.add($id$)
13.       sn.matched_terms = sn.matched_terms + 1
14.     end if
15.   end for
16.   S.push(sn) {create new stack entry}
17. end if
```

used by SLCAStream, which pushes entries to the stack only if necessary and only evaluates plausible queries.

Similar to ELCABStream, ELCAStream consists of three callback functions, each one corresponding to a distinct event. ELCAStream.Start and ELCAStream.Text functions are equivalent to their counterparts SLCAStream.Start and SLCAStream.Text respectively. However, they replace hash tables by inverted lists for correct ELCA evaluation. Specifically, ELCAStream.Start replaces the hash table in SLCAStream.Start (Line 12 in Algorithm 8) by the global inverted list G (Line 12 in Algorithm 14). Similarly, ELCAStream.Text replaces the hash table in SLCAStream.Text (Line 16 in Algorithm 9) by the global inverted list G (Line 16 in Algorithm 15).

In contrast to ELCABStream, ELCAStream uses the matched_terms field to evaluate plausible queries. Therefore, their functions ELCAStream.Start and ELCAStream.Text include operations to update this field. Specifically, in Algorithm 14 (ELCAStream.Start) and Algorithm 15 (ELCAStream.Text), these operations are performed in Lines 13 and 18, respectively.

Algorithm 16 describes the function ELCAStream.End, which pops up the stack top entry sn, if sn corresponds to the node e being processed (Line 2) and evaluates the queries stored in the used_queries set whose number of terms are less than or equal

Algorithm 15 ELCAStream.Text Callback Function.

```
Callback Function ELCAStream.Text
Input: The parsing stack S
Input: The XML node $e$ being processed
Input: The node identification $id$ for node $e$
1.  $\ell$ := label($e$)
2.  new_stack_entry := false
3.  if S.height = $e.height$ then
4.    sn := *S.top() {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.  $K$ := set of tokens in node $e$
10. for all $k \in K$ do
11.   terms := {$k$, ::$k$, $\ell$::$k$} {possible terms containing $k$}
12.   for all $t \in$ terms do
13.     $Q$ := query_index[$t$] {$Q$ get queries of the term $t$}
14.     if $Q \neq \emptyset$ then
15.       new_stack_entry := true
16.       $G[t]$.add($id$)
17.       sn.used_queries.add($Q$)
18.       sn.matched_terms = sn.matched_terms + 1
19.     end if
20.   end for
21. end for
22. if new_stack_entry then
23.   S.push(sn) {create new stack entry}
24. end if
```

to sn.matched_terms value (Line 5). After this, ELCAStream.End tests if e or some of its descendants have occurrences for all q terms (Lines 7 to 12). If e satisfies q completely, ELCAStream.End tests if it is an SLCA node (Line 14). As an SLCA node, e is also ELCA and LCA. Therefore, ELCAStream.End adds the id of e to the result list (Line 15) and records it as the last SLCA result (Line 16). It also stores all LCA contributors of e in the q inverted list (Line 18). Like ELCABStream.End, if e is a non-SLCA node, ELCAStream.End verifies if it is an ELCA node (Lines 20 to 27). This consists in finding at least one node that satisfies each query term t in e or one of its descendants, excluding descendant nodes of e that contribute to LCA results of q (Line 23)[2]. For each term t, ELCAStream.End searches for all nodes in the subtree rooted by e that satisfies t (Line 17 in Algorithm 13) and that are not LCA contributing

[2]Repetition of the Lines 16 to 24 from ELCABStream.End.

Algorithm 16 ELCAStream.End Callback Function.

Callback Function ELCAStream.End

Input: The parsing stack S **and** the XML node e that is ending

```
if S.height = $e.height$ then
  sn := S.pop() {pops the top entry in the stack to sn}
  $id$ := sn.id
  for all $q \in$ sn.used_queries do
    if sn.matched_terms $\geq q$.terms.size() then
      COMPLETE := $true$
      for all $t \in q$.terms do
        if $id <$ last($G[t]$) then
          COMPLETE:=$false$
          break
        end if
      end for
      if COMPLETE then
        if $id > q$.last_result_id then
          $q$.results := $q$.results $\cup$ { sn }
          $q$.last_result_id := $id$
          for all $t \in q$.terms do
            $q[t]$.add($\{id_1, \ldots, id_k\}$), where $id_i$ $(1 \leq i \leq k) \in G[t]$ and $id_i \geq$ sn.$id$
          end for
        else
          can_be_ELCA := $true$
          IdList[ $q$.terms[1], ..., $q$.terms[q.terms.size()] ] := $\emptyset$
          {Repeat lines 16 to 24 from ELCABStream.End}
          if can_be_ELCA then
            $q$.results.add(sn)
          end if
        end if
      end if
    end if
  end for
  tn := *top(S) {tn points to the top entry in the stack}
  if (sn.height - tn.height) = 1 then
    tn.matched_terms = tn.matched_terms + sn.matched_terms
    tn.used_queries = top.used_queries $\cup$ sn.used_queries
  else
    sn.height = sn.height - 1
    sn.id = node_path[$e.height$-1].$id$
    S.push(sn)
  end if
end if
```

nodes for t (Line 18 in Algorithm 13). If the search result is empty, e is not an ELCA node for q (Line 20 in Algorithm 13). In addition, ELCAStream.End stores the result nodes in the inverted list $q[t]$ since they constitute new LCA contributing nodes for t (Line 22 in Algorithm 13). If e is an ELCA node, ELCAStream.End adds it to the result list (Line 25).

Finally, ELCAStream.End must update the parsing stack correctly. If the new top node tn is parent of sn (Line 32), ELCAStream.End adds the number of matching terms up to sn (Line 33) and also adds sn queries to tn (Line 34). If tn is not the parent of sn, sn is pushed to the stack again (Line 38). In this case, sn becomes the previous node from of e. However, its *id* and height must be changed. ELCAStream.End decreases sn height by 1 (Line 36) and updates its *id* to the previous *id* by using the array node_path (Line 37). This array maintains all nodes on the path from e to the root and is updated by ELCAStream.Start and ELCAStream.Text functions.

5.3 Experimental Evaluation

5.3.1 Setup

In this section, we describe the experimental evaluation and its datasets, queries and metrics. In this evaluation, all algorithms were implemented in Java using the SAX API from Xerces Java Parser. The query indexes and other data structures were kept entirely in memory. In this evaluation, all experiments were performed in an Intel 1.8 GHz Core i7 computer with 4 GB of memory.

Dataset. Similar to SLCAStream, ELCABStream and ELCAStream are evaluated in terms of processing time and memory space. Like in Section 4.3, the experiments consist of processing XML document streams against a set of multiple queries over the *SIGMOD*, *XMARK* and *ISFDB* datasets. The average size of these datasets are 58 KB, 280 KB and 1.3 MB, respectively. Notice that these datasets have different orders of magnitude. Table 4.1 presents details of these three datasets. As already mentioned, according to this table, SIGMODR has the deepest documents. ISFDB has flat documents, while its dataset has the largest XML documents. XMARK contains more objects than the other two datasets. We considered each different XML node path as an object.

Queries. We used the same random queries generated from SIGMOD, XMARK and ISFDB datasets for the experimental evaluation described in Chapter 4.

Metrics. Similar to Chapter 4, we measured, for each dataset and experiment, the time spent for processing all XML documents on a given stream. We also excluded the time spent to create the query indexes. Regarding memory consumption, we measured the average memory usage while processing each XML document.

Performance Evaluation. Similar to the SLCAStream experiments (Section 4.3), we performed three experiments with streams containing documents from real 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 pure keywords. The second experiment analyzes how 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. All experiments exclude the time spent for creating the query indexes, which occurs before query processing starts. When measuring memory usage, we considered the average memory used while processing each XML document, including memory used by all index structures.

5.3.2 Results

5.3.2.1 Varying the Number of Queries

This first experiment aims at analyzing the time response with an increasing number of keyword-based queries. It considers random queries, each one with up to four terms. Each query term is of the form *::k*.

Figure 5.1 (left) compares the time spent by ELCABStream and ELCAStream over each dataset. The time curves show a clear advantage of ELCAStream over ELCABStream on the three datasets, since it requires no bitmap and stack operations, and evaluates only plausible queries.

Regarding memory consumption, Figure 5.1 (right) shows that ELCAStream saves lightly more memory when compared to ELCABStream, since it requires no bitmap and reduces stack memory usage.

5.3.2.2 Searching with Structural Constraints

The second experiment analyzes the impact of using structural terms, such as ℓ::k. The experiment considers $50,000$ queries with 5 query terms each and varies the number of structural query terms from 0 to 3.

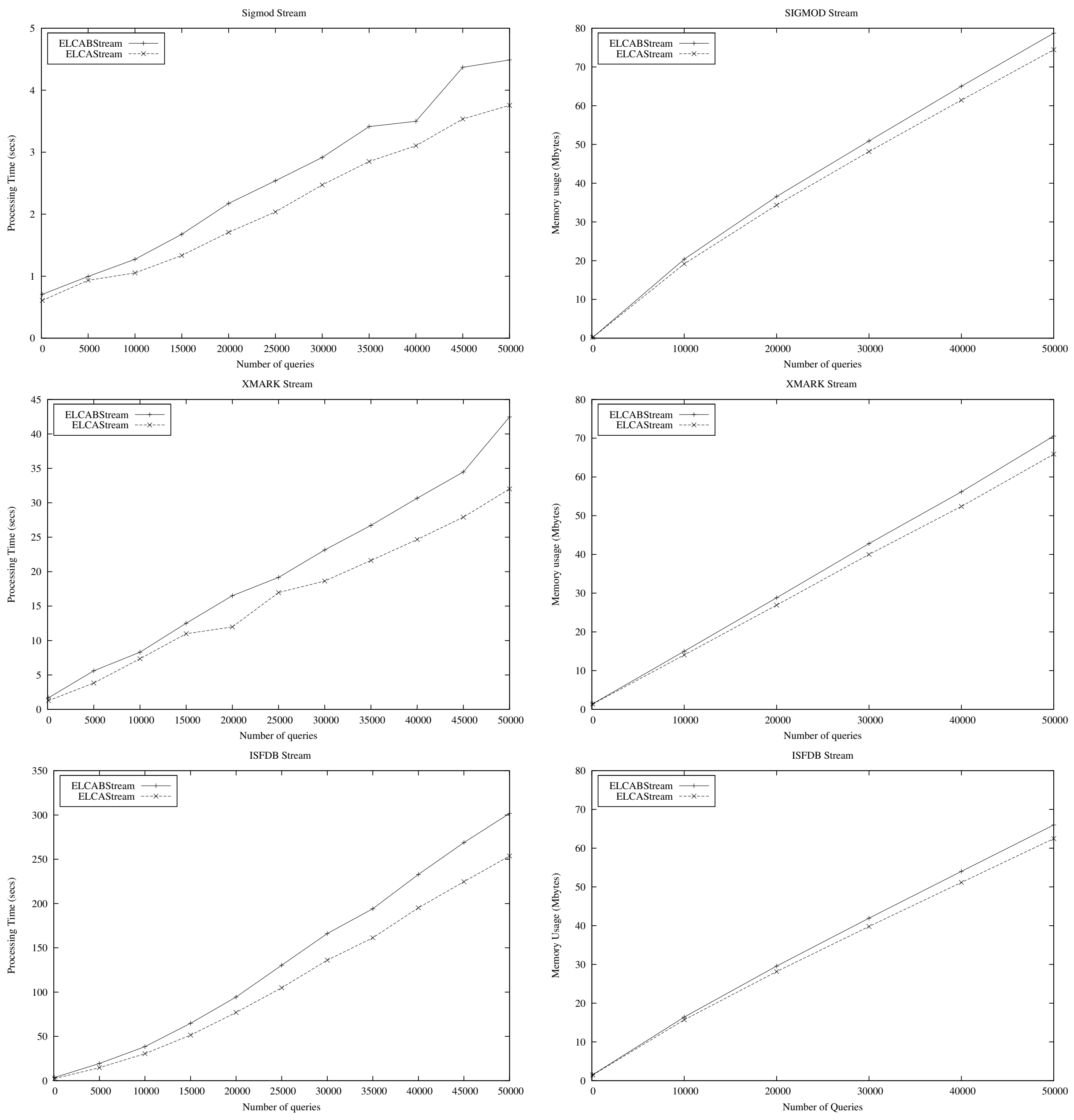


Figure 5.1. First performance experiment: response time (left) and memory usage (right) for ELCABStream and ELCAStream in the three datasets.

Figure 5.2 (left) compares the time spent by ELCABStream and ELCAStream over each dataset. These results also show a clear advantage of ELCAStream for the same reasons pointed out in the previous experiment. In addition, this figure shows that our algorithms perform worst in the ISFDB due to the large number of nodes processed. Figure 5.2 (left) also shows a performance degradation when the number of the structural terms increases. In this experiment, structural query terms are present in each of the 50,000 queries to stress the algorithms' performance, thus representing the worst scenario.

Regarding memory usage, Figure 5.2 (right) shows that SLCAStream saves slightly more memory for the same reasons pointed out in the previous experiment.

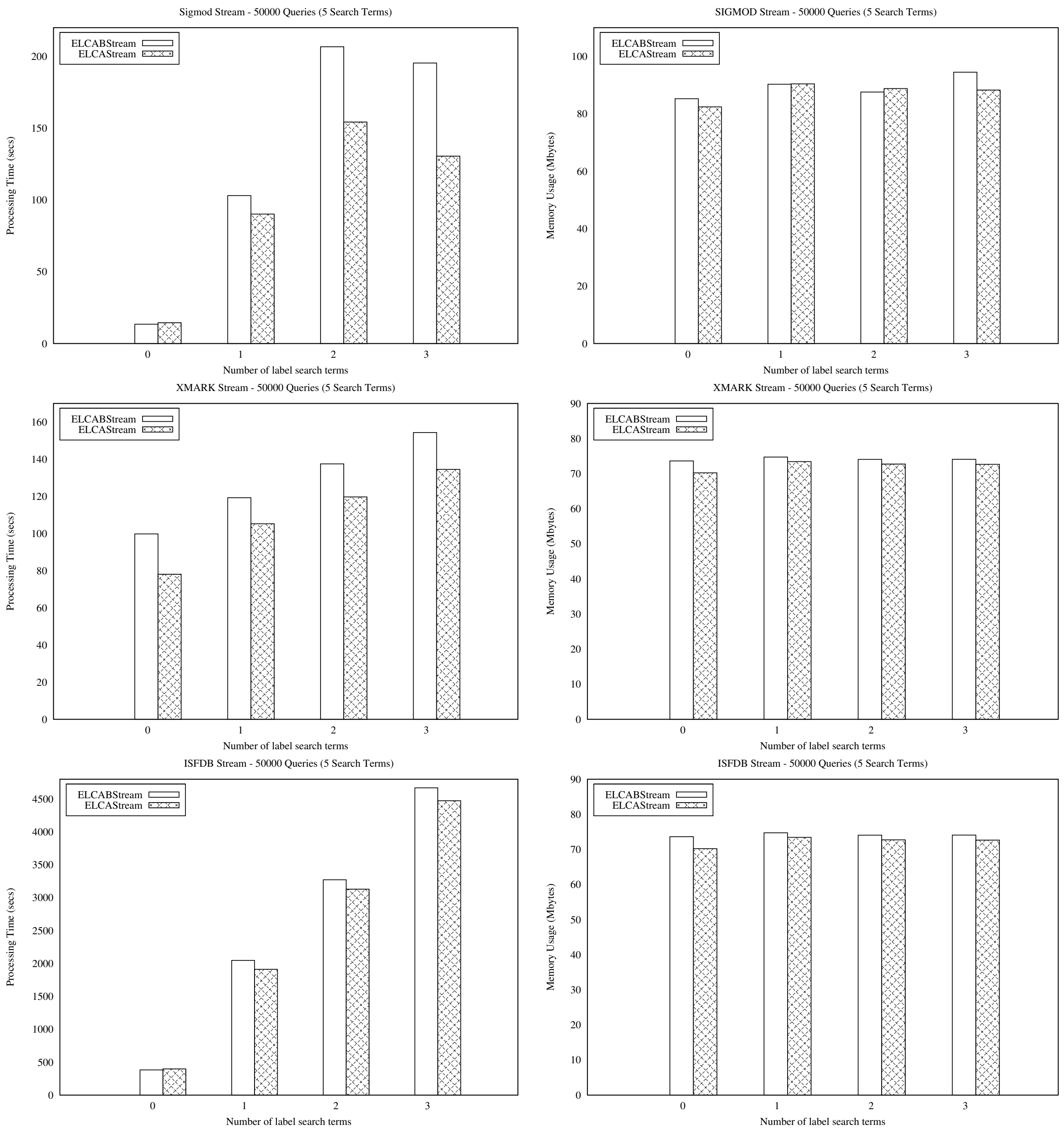


Figure 5.2. Second performance experiment: response time (left) and memory usage (right) for ELCABStream and ELCAStream in the three datasets.

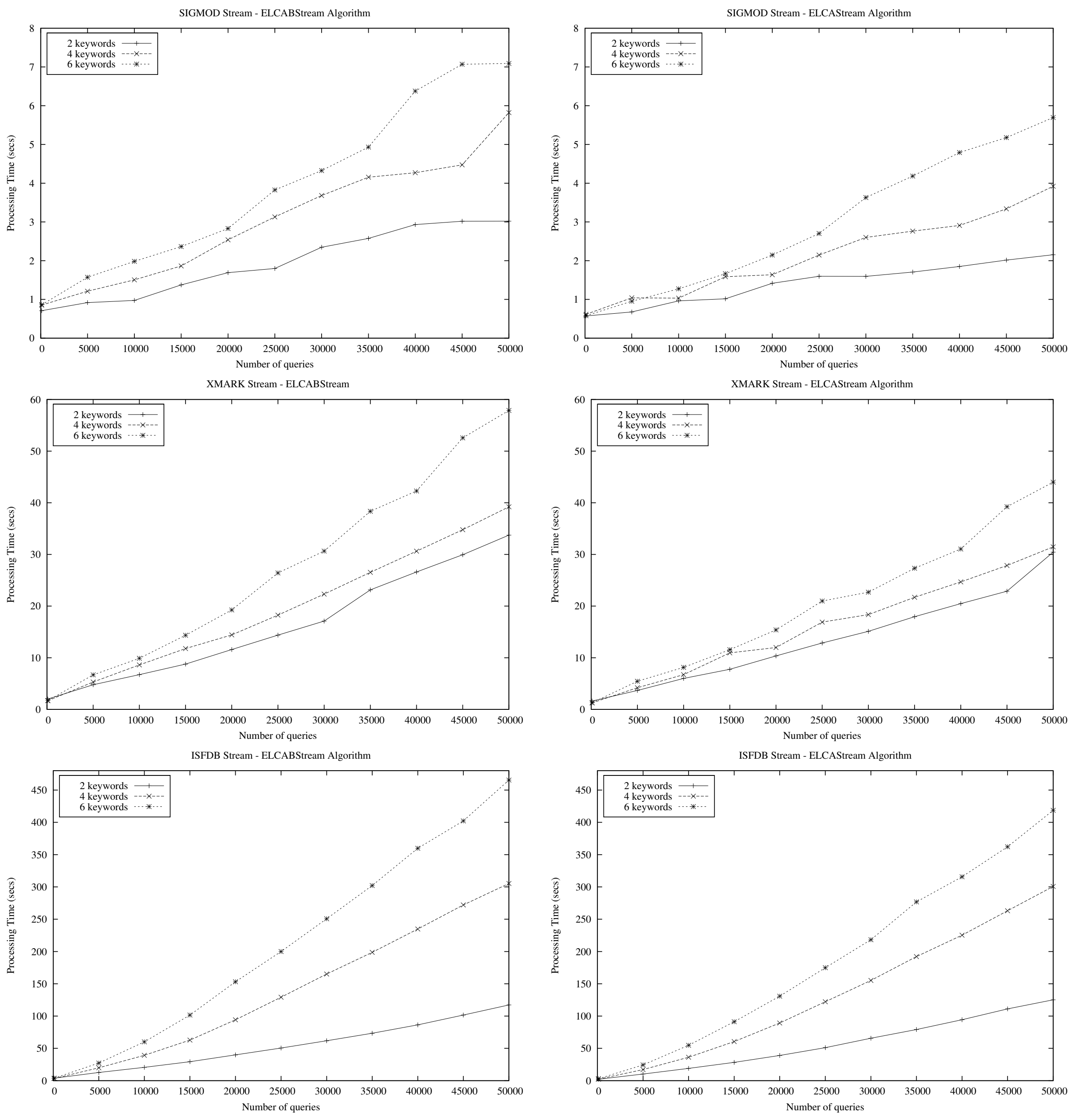


Figure 5.3. Third performance experiment: response times for ELCABStream (left) and ELCAStream (right) algorithms in the three datasets.

5.3.2.3 Varying the Number of Terms

This third 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*. Figure 5.3 presents the results for ELCABStream and ELCAStream on the left and right graphs respectively. As expected, increasing the number of terms affects the performance of the algorithms. However, ELCAStream is slightly better than its baseline, due to its optimization techniques and avoiding

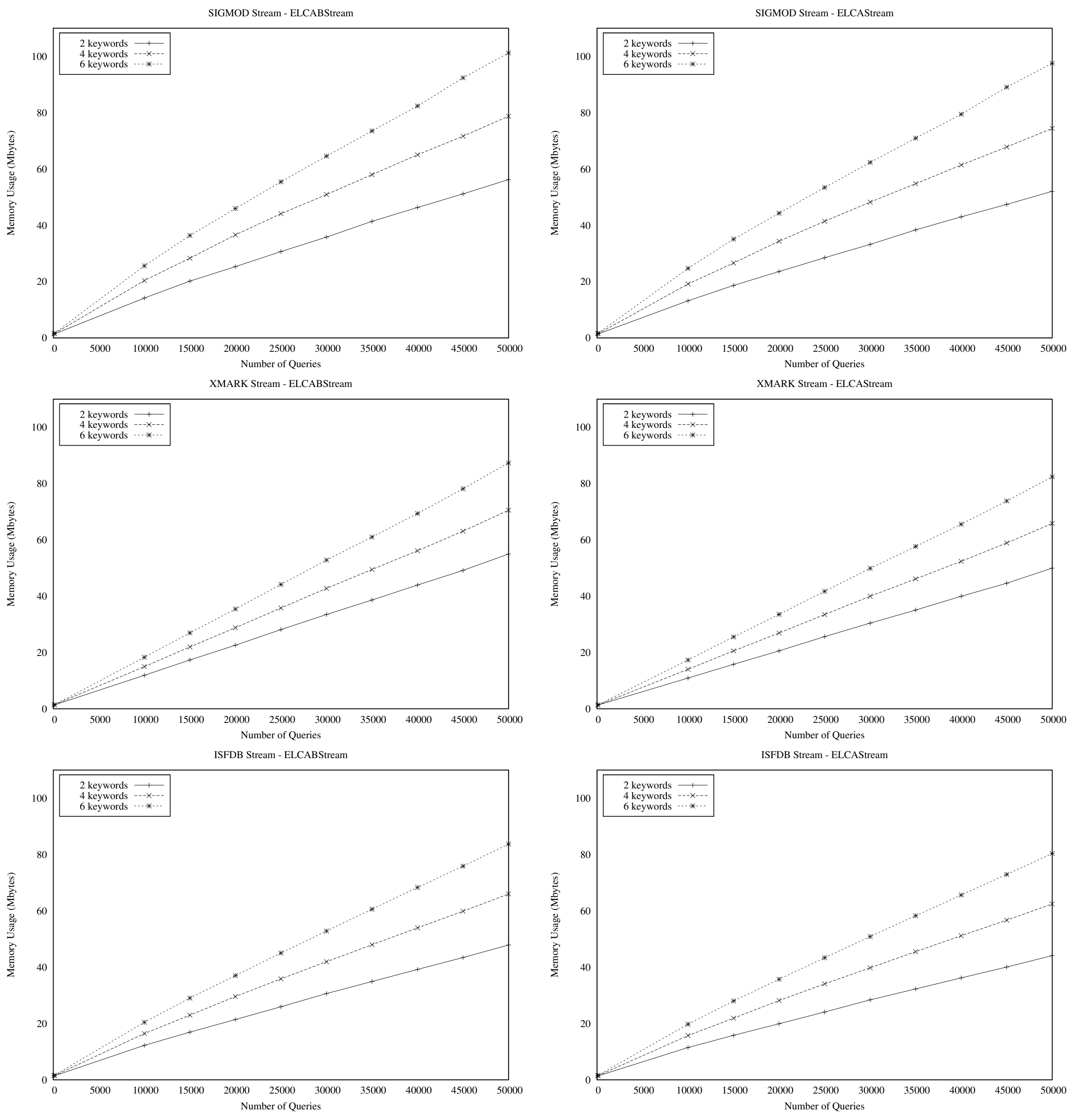


Figure 5.4. Third performance experiment: memory usage for ELCABStream (left) and ELCAStream (right) algorithms in the three datasets.

bitmap processing.

Regarding memory usage, Figure 5.4 shows that ELCAStream has a performance similar to ELCABStream. However, it consumes slightly less memory for not using bitmap structures. Although ELCABStream and ELCAStream use inverted lists for processing queries, which could significantly increase memory consumption when compared to SLCAStream, we notice that this consumption is less than expected. Thus, its implementation is feasible and causes no relevant impact in performance.

5.3.2.4 Remarks

The experiment results indicate that ELCAStream improves ELCABStream response time since it requires no bitmap operations, further reduces stack entries and evaluates only plausible queries. However, the performance gains are relatively small since ELCABStream and ELCAStream work with inverted lists, which consume most of the processing time. Regarding memory, ELCAStream spends less memory than ELCABStream. Although ELCABStream and ELCAStream use inverted lists for processing the queries, their implementations are feasible and cause no relevant impact in performance. In conclusion, ELCAStream offers the best compromise between performance and memory usage. Particularly, it uses optimization techniques and explores a new approach for ELCA evaluation, avoiding the usual bitmap processing strategy, thus improving its overall performance.

Chapter 6

Accuracy Evaluation and Ranking Strategies

This chapter presents an accuracy evaluation concerning recall and precision of the SLCA and ELCA semantics, both adopted by our algorithms for processing multiple keyword-based queries. It also presents the algorithm LCARank and the strategies SLCARank and StreamRank for ranking keyword-based query results. LCARank implements a simple ranking strategy [Barros et al., 2010]. SLCARank is a fine-grained ranking strategy. LCARank and SLCARank focus on ranking XML nodes returned by a keyword query submitted against a single XML document. On the other hand, the StreamRank strategy aims at ranking multiple query results obtained from a set of streams defined by a time slot or specific number of documents.

Tian et al. [2011] reinforce the great interest in improving the result accuracy of keyword-based query processing algorithms. However, these algorithms only process stored XML documents and use stored auxiliary data structures. LCARank and SLCARank are the first strategies to rank results when processing keyword-based queries over XML streams and consider that users require the relevant nodes first. Likewise, StreamRank considers that users require the most relevant nodes first for a set of XML documents which comprise a sliding window, defined by a time slot or specific number of documents.

This chapter is organized as follows. Section 6.1 presents the SLCA and ELCA accuracy evaluation. Section 6.2 presents the LCARank algorithm and its accuracy and performance evaluations. In addition, it presents the SLCARank and StreamRank ranking strategies including its functioning and some examples.

Dataset	Docs	Avg. Avg. Height	Num. of # Elem.	Avg. Num. Avg. Nodes	Avg. Num. Avg. Objects
XMARK	3	5	74	10312	5515

Table 6.1. Details of the XMARK dataset used in our accuracy experiments.

6.1 SCLA and ELCA Accuracy Evaluation

Our proposed algorithms are based on the SLCA and ELCA semantics, which are the most popular LCA-based semantics. Specifically, SLCA returns the lowest subtrees that satisfy all query terms. ELCA not only returns the lowest subtrees but also addresses the ambiguity that might exist in XML data since the same content can occur at different levels. Specifically, in this section we are interested in evaluating the accuracy of both semantics in terms of recall and precision. We evaluated the SLCA and ELCA semantics by comparing the results provided by the SLCAStream and ELCAStream algorithms over the same XMARK dataset that was used in the performance experiments of Chapters 4 and 5. This accuracy evaluation was performed in an Intel 1.8 GHz Core i7 computer with 4 GB of memory.

Dataset. The XMARK dataset used in our accuracy evaluation has important characteristics for our purposes, since its documents, which contain data about auctions, present several elements and deep XML trees as showed in Table 6.1. The average size of the documents in this XMARK dataset is 280 KB.

Queries. We used 15 keyword-based queries adapted from XPath queries specified for the XPathMark benchmark [Franceschet, 2005]. We considered XPath queries in order to be able to provide a baseline for a consistent accuracy evaluation in terms of recall and precision. In our context, given the result of a query, precision measures the percentage of the resulting nodes that are relevant, whereas recall measures the percentage of the relevant nodes that are present in the result. The following list describes these 15 XPath queries and shows their corresponding keyword-based versions according to the query language introduced in Chapter 3:

Q1: All items
XPath: /site/regions/*/item
Keywords: regions:: item::

Q2: Keywords in annotations of closed auctions
XPath: /site/closed_auctions/closed_auction/annotation/
description/parlist/listitem/text/keyword

Keywords: list item:: text:: keyword::

Q3: All keywords
XPath: //keyword
Keywords: keyword::

Q4: Keywords in a paragraph item
XPath: /descendant-or-self::listitem/descendant-or-self::keyword
Keywords: listitem:: keyword::

Q5: Paragraph items containing a keyword
XPath: //keyword/ancestor::listitem
Keywords: keyword:: listitem::

Q6: Mails containing a keyword
XPath: //keyword/ancestor-or-self::mail
Keywords: keyword:: mail::

Q7: North or South American items
XPath: /site/regions/namerica/item | /site/regions/samerica/item
Keywords: namerica:: item:: samerica::

Q8: People having address and either phone or homepage
XPath: /site/people/person[address and (phone or homepage)]
Keywords: address:: phone:: homepage::

Q9: Initial and last bidder of all open auctions
XPath: /site/open_auctions/open_auction/bidder[position()=1 and position()=last()]
Keywords: bidder::

Q10: Items whose description contains ‘gold’
XPath: /site/regions/*/item[contains(description,‘gold’)]
Keywords: item:: description::gold

Q11: Mails sent on the 10th
XPath:/site/regions/*/item/mailbox/mail[substring-before(date,‘/’)=‘10’]
Keywords: mail:: date::10

Q12: Mails sent in 1998

XPath:
/site/regions/*/item/mailbox/mail[substring-after(substring-after(date,'/'),'/')='1998']
Keywords: mail:: date::1998

Q13: Items with descriptions longer than 1000 characters
XPath: /site/regions/*/item[string-length(normalize-space(string(description)))>1000]
Keywords: item:: description::

Q14: People with both an email and a homepage
XPath: /site/people/person[boolean(emailaddress) = true() and not(boolean(homepage)) = false()]
Keywords: person:: emailaddress::

Q15: Person address longer than 30 characters
XPath: /site/people/person[string-length(translate(concat(address/street,address/city,address/country, address/zipcode)," ","")) > 30]
Keywords: person:: address:: street:: city::

Results. Our evaluation used the set of nodes returned by the SAX parser for the XPath queries to determine the relevance of the nodes returned for the keyword queries. Thus, for calculating precision, if the nodes returned by SLCAStream and ELCAStream were present in the set of relevant nodes or in the set of their ancestors, we considered them relevant nodes. If a relevant node is a descendant of, or one of the nodes returned by our algorithms, we considered the returned nodes for calculating recall. Table 6.2 presents precision and recall average figures for both algorithms considering each processed query.

Looking at the results, several queries present low precision. This is mainly due to the fact that some XPath queries can not be translated to our keyword-based query language with exactly the same semantics. For example, queries $Q9$ and $Q13$ are very selective ("Initial and last bidder of all open auctions" and "Items with descriptions longer than 1000 characters"). Thus, when expressed in XPath they return few nodes. On the other hand, their respective keyword versions are unable to express the same restrictions and, therefore, return many more nodes. ELCAStream slightly improves precision for queries $Q2$, $Q4$, $Q5$, $Q6$ and $Q8$. The terms (labels or keywords) in these queries occur simultaneously in different points of the XMARK documents, such as *keyword*, *listitem*, *mail* and *text*. Regarding recall, both algorithms returned 100%

Query	**SLCAStream precision**	**ELCAStream precision**	**SLCAStream recall**	**ELCAStream recall**
1	1.00	1.00	1.00	1.00
2	0.01	0.02	0.00	1.00
3	1.00	1.00	1.00	1.00
4	0.25	0.29	0.23	1.00
5	0.23	0.28	0.19	0.95
6	0.12	0.19	0.17	1.00
7	1.00	1.00	1.00	1.00
8	0.50	0.56	0.20	1.00
9	0.05	0.05	1.00	1.00
10	0.46	0.46	0.74	0.74
11	0.69	0.69	1.00	1.00
12	1.00	1.00	1.00	1.00
13	0.12	0.12	1.00	1.00
14	0.45	0.45	1.00	1.00
15	0.58	0.58	1.00	1.00
Average	0.46	0.47	0.65	0.93

Table 6.2. Average precision and recall for both algorithms.

of the relevant nodes for almost all queries. This was due to the semantics of these algorithms that produces result sets that include more nodes than those for the XPath queries. Queries $Q2$, $Q4$, $Q5$, $Q6$ and $Q8$ present a slightly precision gain over ELCAStream. However, they present a significant recall gain when compared to SLCAStream. This confirms that the ELCA semantics improves precision slightly and recall significantly in relation to the SLCA semantics when the dataset involves keywords occurring simultaneously in different points of the documents. In this experiment, the ELCA semantics improves quality results up to 44% on average, when compared to the SLCA semantics.

6.2 Ranking Algorithms and Strategies

Our first strategy for ranking keyword-based query results from XML streams is the LCARank algorithm, which applies to a single XML document [Barros et al., 2010]. LCARank provides a simple, efficient and effective strategy for ranking XML result nodes obtained by processing a single keyword-based query against a single XML document. It combines the XRANK and SLCA search algorithms proposed by Vagena and Moro [2008], and prioritizes SLCA nodes among other results. Considering that LCARank is a simple ranking algorithm, it is important to propose a fine-grained ranking strategy for improving its results. Thus, we also present the SLCARank strategy and discuss its functioning for future implementation. LCARank and SLCARank focus on ranking the XML nodes returned by a keyword query against a single XML document. They can be used in large XML streams, such as scientific data stored in large XML repositories and documents [Green et al., 2004]. However, both algorithms would be more useful if applied to multiple query results obtained from a set of streams, each one defined by a time slot or a specific number of documents [Li et al., 2007, Singh et al., 2008, Sourlas et al., 2009]. Thus, we also propose the StreamRank strategy, which is applied in this situation.

6.2.1 LCARank Algorithm

As mentioned before, LCARank combines the XRANK and SLCA search algorithms for XML streams proposed by Vagena and Moro [2008]. Their XRANK stream version is based on a search algorithm for stored XML documents proposed by Guo et al. [2003], called XRANK. According to Zhou et al. [2010], the original XRANK algorithm adopts the ELCA semantics.

According to the XRANK algorithm, SLCA nodes and their ancestors, called non-SLCA nodes, are among the ELCA result nodes. However, the XRANK algorithm returns SLCA and non-SLCA results without prioritizing them. Hence, our algorithm LCARank suggests a simple ranking strategy by combining the XRANK and SLCA algorithms. This strategy delivers first SLCA nodes followed by non-SLCA ones. Notice that, only XRANK, which follows the ELCA semantics, specifically handles the ambiguity that might exist among XML labels. This ambiguity has been identified as a semantic problem in XML keyword-based queries [Bao et al., 2009]. Therefore, by combining the XRANK and SLCA algorithms, we are able to handle any label ambiguity that might exist in the XML tree. This way, because the SLCA algorithm returns the smallest XML subtree results that present no keyword ambiguity, they are returned first. The remaining non-SLCA results, generated exclusively by the XRANK algorithm, are returned subsequently. This ranking strategy is the basis of LCARank.

In practice, LCARank works with two result lists, one for SLCA nodes and the other for non-SLCA nodes. When LCARank identifies a result node as being SLCA, the node must stay in the SLCA result list, otherwise it goes to the non-SLCA result list. As its final result, LCARank returns the SLCA nodes first and then the non-SLCA ones.

Algorithm 17 General Single Query Procedure.

```
{Input: Set of Query Terms $q$}
{          Stream of XML documents $D$}
{Output: Set of result XML nodes $R$}
$R := \emptyset$ {initialize result set}
{process each document $d$ in $D$}
for $d \in D$ do
    $S.clear()$ {initialize node stack}
    $R := R \cup SAX\_Parse(d, q, S, R)$ {merge new results with previous ones}
end for
$return\ R_{slca}, R_{non\_slca}$
```

Algorithm 18 LCARank.Start Callback Function.

```
{Input: Accessed Document Node $n$}
{          Set of Query Terms $q$}
{          Stack Node $sn$}
S.push($sn$)
$sn.label := n.label$
$sn.CAN\_BE\_SLCA := TRUE$
$sn.term\_instances := \emptyset$
if $\exists\ l{::} \in q : l == n.label$ then
    $sn.term\_instances(l{::}) := FOUND$
else if $\exists\ k \in q : k == n.label$ then
    $sn.term\_instances(k) := FOUND$
end if
```

6.2.1.1 Callback Functions

The general operation of LCARank is described by Algorithm 17. It evaluates a query q over a stream of documents D. It uses internally a global node stack S and a data structure R, which contains two lists, R_{slca} and R_{non_slca}. Each document in D is sequentially processed by a SAX parser (Line 8). Like our other proposed algorithms, the SAX parser triggers the LCARank callback functions. However, LCARank processes only one query per time over the XML stream. LCARank also adopts the keyword-based query language presented in Section 3.2.

The callback functions LCARank.Start, LCARank.Text and LCARank.End are described by Algorithms 18, 19 and 20, respectively. These functions access the global node stack S, which maintains the opened nodes during the document processing. The top entry of this stack maintains the most recent node being processed. Thus, each stack entry corresponds to a node and includes its label and query term bitmap, whose bits individually map each user query term.

Algorithm 19 LCARank.Text Callback Funtion

```
{Input: Textual Content $text$}
{       Node Stack $S$}
{       Set of Query Terms $q$}
for $word \in$ tokenize($text$) do
  if $\exists$ ::$k \in q : k == word$ then
    $S.top().term\_instances(::k) := FOUND$
  else if $\exists\ k \in q : k == word$ then
    $S.top().term\_instances(k) := FOUND$
  else if $\exists\ l::k \in q : (l == S.top().label\ AND\ k == word)$ then
    $S.top().term\_instances(l::k) := FOUND$
  end if
end for
```

LCARank.Start pushes the current node into the stack (Line 4). It also verifies if the current node label matches a keyword term of the form k or k:: (Lines 8 to 12). In this case, the respective query term bit in the bitmap is set to 1 (Line 9 or 11). For simplicity, we omit the code that handles XML attributes, as the corresponding process is identical to visiting a leaf node.

LCARank.Text accesses the top node on the stack and verifies if its text tokens match query terms of the form k, ::k or l::k (Lines 4 to 12). Similarly, the matched query terms have their respective bits set to 1 in the current bitmap (Lines 6, 8 or 10).

Finally, LCARank.End finishes the current node and removes it from the node stack (Line 3). If its bitmap has only $1's$, then that node satisfies all query terms and, therefore, is a resulting SLCA or non-SLCA node (Lines 4 to 11). Otherwise, its $1's$ are copied to the node on the top of the stack, which is the parent of the popped node from the XML tree (Lines 15 to 19). Function LCARank.End ranks the resulting nodes by using the two lists R_{slca} and R_{non_slca} which group SLCA results and non-SLCA results respectively (Lines 5 to 9).

When LCARank.End finds a result node and its SLCA flag (CAN_BE_SLCA) is set to TRUE (Line 5), this means that this node is an SLCA result (Line 6). According to the SLCA semantics, its parent is therefore a non-SLCA node. Thus, LCARank.End sets the SLCA flag of its XML parent node (the next on the stack top) to FALSE (Line 10). Nodes with an incomplete bitmap and a FALSE SLCA flag have the SLCA flags of their parent nodes set to FALSE (Lines 12 to 14). Every new visited node is potentially an SLCA node, thus LCARank.Star always sets the new node's SLCA flag to TRUE (Line 6 in Algorithm 18).

Algorithm 20 LCARank.End Callback Function.

```
1.  {Input: Set of result XML nodes $R$}
2.  {get the top node of the stack, which is being finalized}.
3.  $sn := s.pop()$
4.  if $sn.term\_instances == COMPLETE$ then
5.     if $sn.CAN\_BE\_SLCA == TRUE$ then
6.        $R_{slca} := R_{slca} \cup \{r_n\}$
7.     else
8.        $R_{no\_slca} := R_{no\_slca} \cup \{r_n\}$
9.     end if
10.    $S.top().CAN\_BE\_SLCA := FALSE$
11. else
12.    if $sn.CAN\_BE\_SLCA == FALSE$ then
13.       $S.top().CAN\_BE\_SLCA := FALSE$
14.    end if
15.    for $tk \in sn.term\_instances.keys$ do
16.       if $sn.term\_instances(tk) == FOUND$ then
17.          $S.top().term\_instances(tk) := FOUND$
18.       end if
19.    end for
20. end if
```

6.2.1.2 Experiments

To evaluate the efficiency of our ranking strategy, we run the LCARank algorithm for processing the keyword queries described in Section 6.1. As expected, LCARank improves the original XRANK ranking algorithm as the internal SLCA nodes are always returned before the non-SLCA ones. To demonstrate this improvement, we calculated the normalized discounted cumulative gain (DCG) [Järvelin and Kekäläinen, 2002] between the XRANK and LCARank results, considering the LCARank ranking as ideal. Particularly, DCG evaluates the ranking for a result list, summing up the document relevance values in a result list. Each document in the result list has a relevance value based on its position in the list. The premise behind DCG is that highly relevant documents appearing lower in a result list should be penalized as the relevance value is logarithmically reduced with respect to the position of the result. DCG is calculated using Equation 6.1, where p is a particular ranking position, rel_i is the relevance weight for the document in the position p and rel_1 is the relevance weight for the first document.

$$DCG_p = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{\log_2 i} \tag{6.1}$$

Document Size (MB)	Mean XRANK nDCG
0.12	0.98
0.21	0.95
0.47	0.96
0.91	0.96
1.89	0.97

Table 6.3. Mean Normalized DCG per Document

The comparison between the XRANK and LCARank results is clearly interpreted if we use the normalized DGC, which is done assuming one of the algorithms as the one that produces the ideal results. In this case, LCARank is that algorithm, since it returns the smaller subtrees first. For a specific query, the normalized discounted cumulative gain ($nDCG$) is calculated using Equation 6.2, where $IDCG_p$ is the LCARank DCG, given as the ideal one.

$$nDCG_p = \frac{DCG_p}{IDCG_p} \tag{6.2}$$

In our context, we adapted the DCG metric to consider retrieved nodes from a single XML document instead from a document list. For each query, we calculated its DCG based on the retrieved nodes from a single XML document. As relevance weights we used 1.0 for SLCA nodes, 0.5 for non-SCLA nodes and zero for irrelevant nodes. Even if XRANK returned an SCLA node as a non-SLCA one, its relevance weight was set to 1.0 in order to keep the results fair.

Table 6.3 presents the mean normalized DCG for the XRANK algorithm considering five XMARK documents, being three of them the some used in the first accuracy study presented in Section 6.1. We included two more XMARK documents to provide more data for our analysis. For all documents and regardless of their sizes, XRANK accuracy is inferior to LCARank accuracy, considered in this experiment as the algorithm that provides the ideal results, which means that LCARank DCG is equal to 1.0. Thus, these results demonstrate that LCARank, despite its simple strategy, improves the raking when compared to the original XRANK algorithm.

We also evaluate the LCARank performance. To do so, we compared its mean elapsed time with that of the XRANK and SLCA algorithms. We processed the 15 keyword-based queries presented in Section 6.1 against the XMARK dataset and measured the mean elapsed time for each document. Notice that, for this experiment, the XMARK dataset includes two additional documents that are larger than the ones used in Section 6.1. Figure 6.1 presents the respective mean elapsed times, which involve

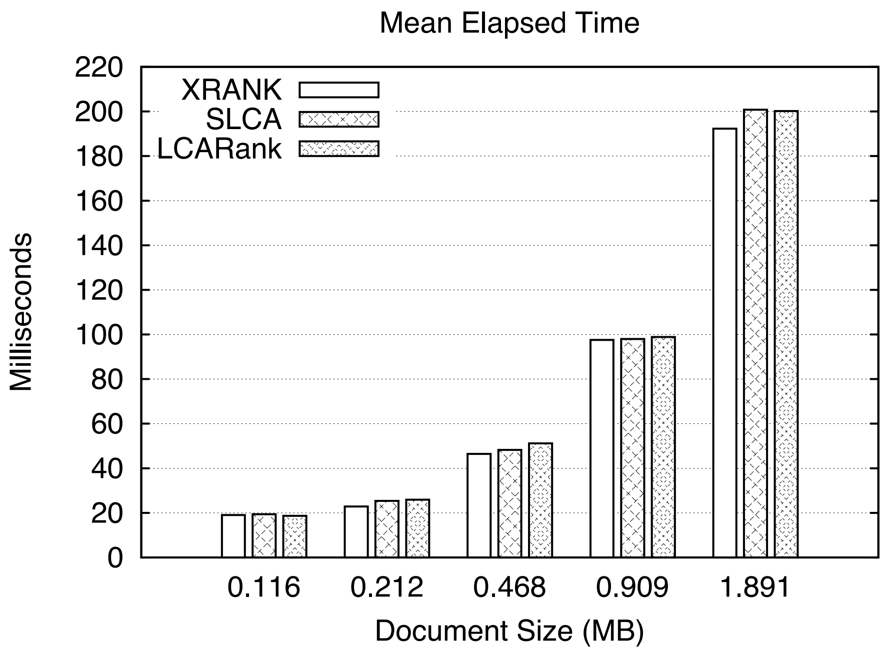


Figure 6.1. Mean query time per document size

the parsing and matching processes. The results show that the XRANK, SLCA and LCARank algorithms have similar response time, however only LCARank implements a ranking strategy. These results are acceptable considering that typical XML streaming applications manipulate small documents [Lenkov, 2003].

6.2.2 SLCARank Strategy

In this section, we propose a new SLCA ranking strategy, called SLCARank, which prioritizes the shortest SLCA nodes in the query result. In the following, we describe this strategy and discuss some examples.

6.2.2.1 Strategy Functioning

LCARank improves the original XRANK algorithm ranking since its internal SLCA nodes always are returned before the non-SLCA nodes. However, its SLCA nodes are returned without applying any ranking strategy. Therefore, we propose the SLCARank strategy, a more fine-grained ranking strategy for SLCA nodes that are internally returned by the LCARank algorithm. Basically, this strategy prioritizes the shortest subtrees in the result sets. Thus, the tallest nodes occupy the last positions in the ranked result sets.

Our proposed ranking strategy is based on the TF-IDF term weighting scheme [Salton and Buckley, 1988], borrowed from information retrieval. This approach normally relies on inverted lists built on the a whole set of documents. However, when searching over streams, these indexing structures do not exist. The absence of inverted lists prevent the use of IDF (Inverted Document Frequency) terms. These terms inversely correlate a keyword frequency in the document with its frequency in the whole set of documents. We adapted the TD-IDF scheme by considering a single document instead of a set of documents. Initially, we use Equations 6.3 and 6.5 to evaluate the ranking score for each returned SCLA node.

$$Score(r,q) = \sum_{i=1}^{k} \frac{weight(t_i, r)}{dist(r, n_i)} \tag{6.3}$$

$$weight(t_i, r) = (1 + \log f_{t_i,r}) * \log\left(\frac{|V|}{|V_{t_i}|}\right) \tag{6.4}$$

$$dist(r, n_i) = h_{n_i} - h_r + 1 \tag{6.5}$$

In Equation 6.3, q is the set of term queries $\{t_1, t_2, t_3, \ldots, t_k\}$, where k is the number of query terms and r is an SLCA result node for q. As a ranking criterion, the nodes with the highest ranking score are returned first. In Equation 6.4, $f_{t_i,r}$ is the frequency of the query term t_i in the node r, $|V|$ is the number of nodes in the XML document and $|V_{t_i}|$ is the number of nodes that contain the term t_i as their children or descendants. The expression $(1 + \log f_{t_i,r})$ represents the TF component in the TF-IDF scheme and $\log(\frac{|V|}{|V_{t_i}}|)$ represents the IDF component. Equation 6.5 determines the distance between the SLCA node r and the node n_i, where the query term t_i occurs. Specifically, this distance is $h_{n_i} - h_r + 1$, where h_{n_i} is the height of n_i and h_r is the height of r in document tree. We add 1 to avoid zero, when $h_{n_i} = h_r$.

6.2.2.2 Examples

In this subsection, we show the SLCARank effectiveness using two examples. The first is the simplest one and refers to keywords occurring just once in the XML nodes. The second shows that SLCARank also works well when a single query term occurs more than once in a node or when all query terms occur in a single node.

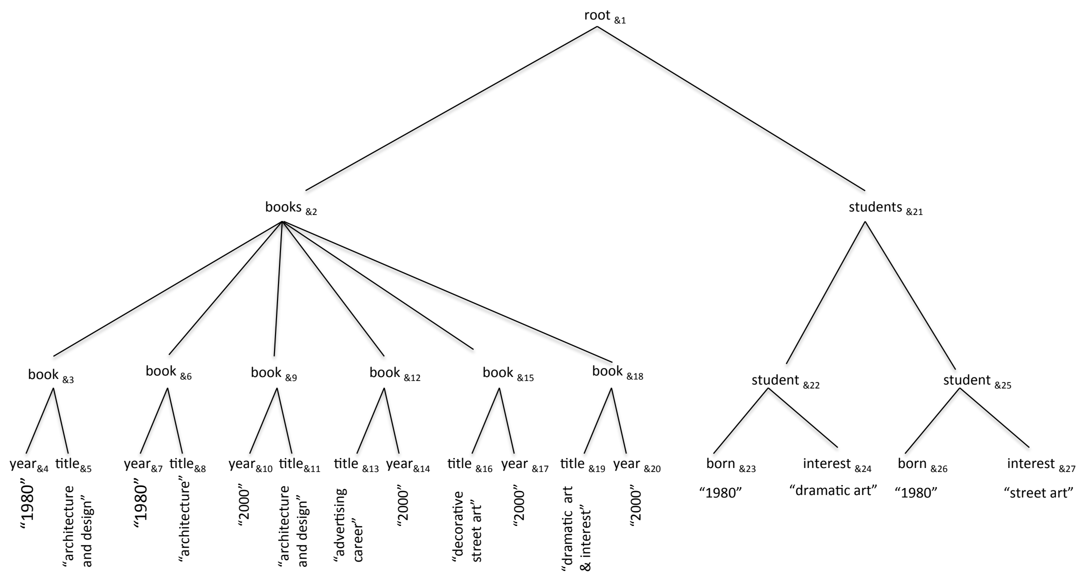


Figure 6.2. XML document with six books and two students

Example 1. *Suppose a user submits the query $q = \{1980, art\}$ to be processed by LCARank against the XML document in Figure 6.2. The result will be the SLCA nodes $books_{\&2}$, $student_{\&22}$ and $student_{\&25}$, in this order. However, $student_{\&22}$ and $student_{\&25}$ subtrees are shorter than $books_{\&2}$. As shortest subtrees involve more significant results, $student_{\&22}$ and $student_{\&25}$ are returned first by the SLCARank strategy.*

The keyword *1980* occurs once in each *student* node in Figure 6.2. Following Equation 6.3, we have $\frac{weight(1980,student)}{dist(student,born)} = 0.20$ for nodes $student_{\&22}$ and $student_{\&25}$ as they have the same structure and similar content as shown in Figure 6.2. The Equation

6.6 shows how this value is calculated.

$$\begin{aligned}
\frac{weight(1980, student)}{dist(student, born)} &= \frac{(1+\log f_{1980,student}) * \log\left(\frac{|V|}{|V_{1980}|}\right)}{dist(student, born)} \\
&= \frac{(1+\log 1) * \log\left(\frac{|27|}{|11|}\right)}{2} \\
&= \frac{(1+0) * \log 2.46}{2} \\
&= \frac{1 * 0.39}{2} \\
&= 0.20
\end{aligned} \tag{6.6}$$

The terms in Equation 6.6 are calculated as follows. The term frequency for keyword *1980* is $f_{1980,student} = 1$. The inverted frequency for keyword *1980* is $log(\frac{|V|}{|V_{1980}|})$, since the number of nodes in the XML document is $|V| = 27$ and the number of nodes or its descendants containing the keyword *1980* is $|V_{1980}| = 11$. Moreover, the distance between the *student* node and the leaf node *born* is $dist(student, born) = 2$, since the keyword *1980* occurs once in the node *born*, child of *student*. Specifically, the node *born* has $h_{born} = 3$ and the node *student* has $h_{student} = 2$. As a result, $dist(student, born) = h_{born} - h_{student} + 1 = 3 - 2 + 1 = 2$.

Similarly, we show in Equation 6.7 how the expression $\frac{weight(art,student)}{dist(student,interest)}$ is calculated for the keyword *art* in the node *student*. This equation considers the distance between the nodes *student* and *interest*, in which the keyword *art* occurs.

$$\begin{aligned}
\frac{weight(art, student)}{dist(student, interest)} &= \frac{(1+\log f_{art,student}) * \log\left(\frac{|V|}{|V_{art}|}\right)}{dist(student, interest)} \\
&= \frac{(1+\log 1) * \log\left(\frac{|27|}{|11|}\right)}{2} \\
&= \frac{(1+0) * \log 2.46}{2} \\
&= \frac{1 * 0.39}{2} \\
&= 0.20
\end{aligned} \tag{6.7}$$

The terms in Equation 6.7 are calculated as follows. The term frequency for the keyword *art* is $f_{art,student} = 1$, since this keyword occurs once in each node *student*. The inverted frequency for the keyword *art* is $log(\frac{|V|}{|V_{art}|})$, where $|V| = 27$ and $|V_{art}| = 11$.

Moreover, similar to keyword 1980, $dist(student, interest) = 2$, since the keyword *art* occurs once in the node *interest*, child of *student*.

Having the values from Equations 6.6 and 6.7, we calculate the ranking score for each node *student* using Equation 6.3, since nodes $student_{\&22}$ and $student_{\&25}$ have the same structure and similar content. Following Equation 6.8, we have:

$$\begin{aligned} Score(student, \{1980, art\}) &= \sum_{i=1}^{2} \frac{weight(t_i, student)}{dist(student, n_i)} \\ &= \frac{weight(1980, student)}{dist(student, born)} + \frac{weight(art, student)}{dist(student, interest)} \\ &= 0.20 + 0.20 \\ &= 0.40 \end{aligned} \tag{6.8}$$

Example 1 also includes the node $books_{\&2}$ as its result, whose score value is calculated following Equation 6.9:

$$\begin{aligned} Score(books_{\&2}, \{1980, art\}) &= \sum_{i=1}^{2} \frac{weight(t_i, books_{\&2})}{dist(books_{\&2}, n_i)} \\ &= \frac{weight(1980, books_{\&2})}{dist(books_{\&2}, year)} + \frac{weight(art, books_{\&2})}{dist(books_{\&2}, title)} \\ &= \frac{(1 + \log tf_{1980, books_{\&2}}) * \log\left(\frac{|V|}{|V_{1980}|}\right)}{dist(books_{\&2}, year)} \\ &+ \frac{(1 + \log tf_{art, books_{\&2}}) * \log\left(\frac{|V|}{|V_{art}|}\right)}{dist(books_{\&2}, title)} \\ &= \frac{(1 + \log 2) * \log(\frac{27}{11})}{3} + \frac{(1 + \log 2) * \log(\frac{27}{11})}{3} \\ &= \frac{2 * (1 + 0.30) * 0.39}{3} \\ &= \frac{1.0}{3} \\ &= 0.33 \end{aligned} \tag{6.9}$$

The terms in Equation 6.9 have been calculated as follows. The term frequencies for keywords *1980* and *art* are $f_{1980, books_{\&2}} = 2$ and $f_{art, books_{\&2}} = 2$ as they occur two times in node $books_{\&2}$. The number of nodes or theirs descendants containing each keyword are, respectively, $|V_{1980}| = 11$ and $|V_{art}| = 11$. Finally, the distance

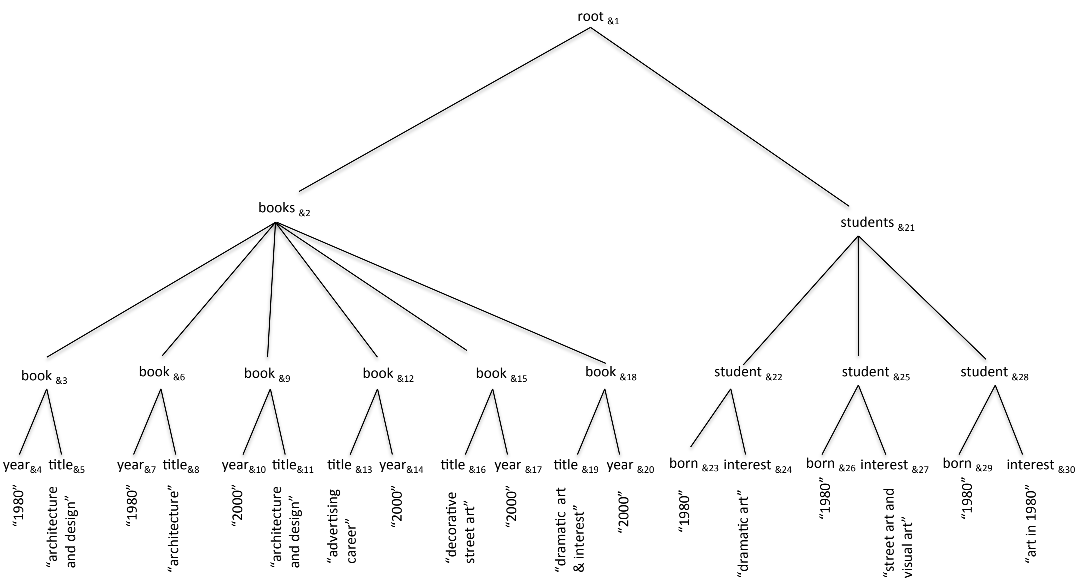


Figure 6.3. XML document with six books and three students

between node $books_{\&2}$ and the node where the keyword occurs is $dist(books, year) = 3$ for keyword *1980* and $dist(books, title) = 3$ for keyword *art*, as their parent's nodes are grandchildren of $books_{\&2}$ node. Specifically, node *year* has $h_{year} = 3$, node *title* has $h_{title} = 3$ and node *books* has $h_{books_{\&2}} = 1$. As a result, $dist(books_{\&2}, born) = h_{books_{\&2}} - h_{student} + 1 = 3 - 1 + 1 = 3$. Similarly, $dist(books_{\&2}, title)$ is calculated, which value is also three.

According to the above calculated scores, the nodes $student_{\&22}$ and $student_{\&25}$ are more relevant than $books_{\&2}$, since $Score(student_{\&22}, \{1980, art\}) = Score(student_{\&25}, \{1980, art\}) = 0.40$ is greater than $Score(books_{\&2}, \{1980, art\}) = 0.33$. Thus, $student_{\&22}$ and $student_{\&25}$ will be ranked higher than $books_{\&2}$, differently from the original LCARank implementation that first shows the node $books_{\&2}$ followed by $student_{\&25}$.

Our SLCARank strategy also works well when a query term occurs more than once in a node or all query keywords occur in a single node. To better illustrate each case, we discuss next Example 2, which involves the same query $q = \{art, 1980\}$ and the XML document presented in Figure 6.3. Contrasting this document with that in Example 1, we have node $student_{\&25}$ with two occurrences of the keyword *art*, and the new node $student_{\&28}$, which includes a node *interest*, containing both keywords. The other nodes remain the same.

Example 2. *Consider the query $q = \{art, 1980\}$ and the XML document in Figure*

6.3. The expected result according to the SLCARank strategy are the nodes $interest_{\&30}$, $student_{\&25}$, $student_{\&22}$ and $books_{\&2}$, in this order. Node $interest_{\&30}$ comes first because it includes both keyword occurrences in a single node and $student_{\&25}$ comes after since it includes two occurrences of the keyword art. Thus, these two nodes are considered more relevant than the other ones.

The difference between Examples 1 and 2 is how the term frequency varies for the keyword *art* in some nodes. In Example 1, the term frequency $f_{art,student_{\&25}}art = 1$, while in Example 2 $f_{art,student_{\&25}}art = 2$. This difference affects the weight of *art* in $student_{\&25}$, as presented in Equation 6.10.

The other terms in Equation 6.10 are described as follows. The number of nodes in the XML document is $|V| = 30$, the number of nodes that contain the keyword *1980* is $|V_{1980}| = 14$. Similarly, $|V_{art}| = 14$. Finally, the value of the distance function for nodes $student_{\&25}$ and $born_{\&26}$, where the keyword *1980* occurs, is $dist(student_{\&25}, born_{\&26}) = 2$. For the keyword *art*, $dist(student_{\&25}, interest_{\&27}) = 2$.

$$\begin{aligned}
Score(student_{\&25}, \{1980, art\}) &= \sum_{i=1}^{2} \frac{weight(t_i, student_{\&25})}{dist(student_{\&25}, n_i)} \qquad (6.10)\\
&= \frac{weight(1980, student_{\&25})}{dist(student_{\&25}, born_{\&26})} + \frac{weight(art, student_{\&25})}{dist(student_{\&25}, interest_{\&27})}\\
&= \frac{(1+\log tf_{1980,student_{\&25}}) * \log\left(\frac{|V|}{|V_{1980}|}\right)}{dist(student_{\&25}, born_{\&26})}\\
&+ \frac{(1+\log tf_{art,student_{\&25}}) * \log\left(\frac{|V|}{|V_{art}|}\right)}{dist(student_{\&25}, interest_{\&27})}\\
&= \frac{(1+\log 1) * \log(\frac{30}{14})}{2} + \frac{(1+\log 2) * \log(\frac{30}{14})}{2}\\
&= \frac{(1+0) * 0.36}{2} + \frac{(1+0.301) * 0.36}{2}\\
&= \frac{0.36}{2} + \frac{0.47}{2}\\
&= 0.18 + 0.24\\
&= 0.42
\end{aligned}$$

Similarly, we evaluate the ranking score for nodes $interest_{\&30}$, $student_{\&22}$ and $books_{\&2}$, which are, respectively, $Score(interest_{\&30}, \{1980, art\}) = 0.73$, $Score(student_{\&22}, \{1980, art\}) = 0.36$ and $Score(books_{\&2}, \{1980, art\}) = 0.32$. According to these values, the ranking order, as expected, is $interest_{\&30}$, $student_{\&25}$,

$student_{\&22}$ and $book_{\&2}$.

The SLCARank strategy involves specific document information, such as the number of nodes in an XML document, the term frequency in an LCA node, the distance between LCA and matching nodes for each query keyword, and the number of nodes containing a query term as its own or in its descendants. We propose to obtain this information during the XML document traversal by using specific counters for term occurrences and specific variables to store node heights.

6.2.3 StreamRank Strategy

StreamRank is a ranking strategy applicable to multiple query results obtained from a set of streams defined by a time slot or a specific number of documents. In what follows, we describe and discuss an example that presents this strategy.

Our strategy uses the resulting XML subtrees obtained from the SLCAStream or ELCAStream algorithms. It calculates the TF-IDF similarity between documents in streams and queries. The similarity between a query q and a document d expresses the relevance of the document d for query q. The more similar the document d and the query q are, the greater the relevance of d is. StreamRank strategy is based on TF-IDF similarity and uses Equations 6.11 to 6.15. An important consideration is that the TF-IDF similarity is calculated over a set of documents obtained in a time slot or after a given number of sub-trees arrive. A time slot is defined by a time interval. Both, this time interval and the number of resulting XML subtrees, are also known as a stream sliding window [Li et al., 2007, Singh et al., 2008, Sourlas et al., 2009].

$$W_{d,k} = 1 + \ln(f_{d,k}) \tag{6.11}$$

$$W_{q,k} = \ln(1 + \frac{N}{f_k}) \tag{6.12}$$

$$W_q = \sqrt{\sum_{k \in q} W_{q,k}^2} \tag{6.13}$$

$$W_d = \sqrt{\sum_{k \in d} W_{d,k}^2} \tag{6.14}$$

$$\rho(q,d) = \frac{\sum_{k \in q \cap d} W_{q,k} * W_{d,k}}{W_q * W_d} \tag{6.15}$$

Equation 6.11 calculates the frequency of a keyword k in the document d, being $f_{d,k}$ the term frequency (*TF - Term Frequency*). Equation 6.12 calculates the inverted frequency (*IDF - Inverse Document Frequency*) of keyword k, being N the number of XML documents in a window and f_k the number of documents containing k. The logarithms in both equations normalize high values of f_k and $f_{d,k}$. Equations 6.13 and 6.14 define the normalization factors used in Equation 6.15 to calculate the TF-IDF similarity between the document d and the query q, which contains all keywords k. The relevance of the document d for query q is based on the similarity between d and q.

In the following, we present an StreamRank example, which considers the query $q = \{1980, art\}$ and the XML documents in Figures 6.2 and 6.3.

Example 3. *Consider an stream composed of the XML documents in Figures 6.2 and 6.3, which are processed in this order. Now, suppose that the SLCARank algorithm processes the query q against these two documents. The results are the subtrees from the first document (Figure 6.2), followed by the resulting subtrees from the second one (Figure 6.3). However, considering the Equation 6.15, the TF-IDF similarity for the first document is* 0.26 *and for the second is* 0.30*. Thus, the resulting subtrees from the second document should be delivered first since it is the most relevant one, according to TF-IDF similarity. Basically, this new order happens because the frequency of the query q keywords in the second document is larger than in the first one. After returning result nodes from the second document, StreamRank returns result nodes from the first document.*

The StreamRank ranking strategy involves several issues that should be addressed in future work. Some of them include: (i) balance between the size of the document windows and the effectiveness of the relevance ranking strategy adopted, (ii) accuracy evaluation for ranked results of keyword queries processed over XML streams and (iii) choice of an appropriate similarity model.

Chapter 7

Conclusions and Future Work

XML streams have become a relevant research topic due to the widespread use of applications such as online news, RSS feeds and dissemination systems. Such streams must be processed rapidly and without retention. Retaining streams could cause data loss due to the large data traffic in continuous processing. This context becomes more complex when thousands of queries must be processed simultaneously. Different approaches explore simultaneous multiple query processing. However, they are based on structured languages such as XPath and SQL, which require knowledge of their syntax and the data schemas to formulate queries. Keyword-based languages are a common approach to submit queries informally, because keyword-based queries require minimal or no schema knowledge to formulate queries.

Recent work on XML querying has focused on efficient algorithms for processing queries over XML streams [Wu and Theodoratos, 2012]. However, most of the algorithms process one query at a time or consider only queries expressed by structured languages. Thus, keyword-based query languages are natural alternative on such an environment, since they are more convenient for users to specify their information needs. This results in other another emerging research topic which is related to dynamically ranking XML nodes returned by keyword-based queries.

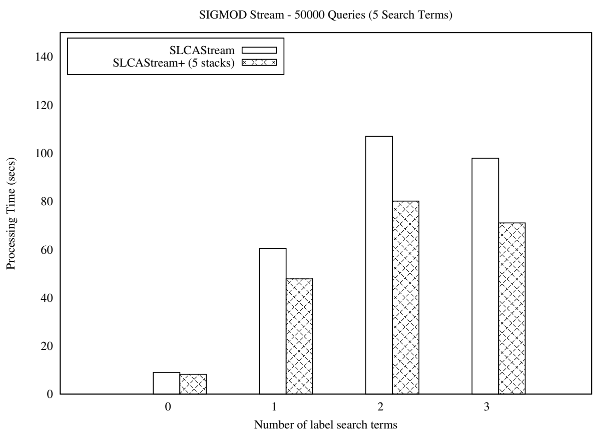
In this thesis, we addressed the problem of processing multiple keyword-based queries over XML document streams. We proposed five algorithms for simultaneously processing several keyword-based queries over XML streams. These new algorithms allow multiple users with disparate interests to define their profiles by specifying queries informally since only minimal data schema knowledge is required. They adopt the two most used LCA (lowest common ancestor) semantics for keyword-based query processing over XML streams: SLCA (smallest LCA) and ELCA (exclusive LCA). Regarding performance, we conducted a comprehensive set of experiments to demonstrate their behavior regarding memory usage and processing time.

For the SLCA semantics, we proposed three algorithms, each addressing different issues and processing strategies. BStream is the first algorithm and uses traditional bitmaps for multiple query processing over streams. BStream is a basic multi-query implementation for one of the first SLCA algorithms for processing single keyword-based queries over XML streams [Vagena and Moro, 2008]. BStream evaluates all submitted queries, representing their terms in a single bitmap. MKStream, our second algorithm, extends BStream since it represents all query terms in a single compact bitmap, without query term repetition. Moreover, MKStream simultaneously uses multiple parsing stacks. Such strategies allow an overall MKStream performance improvement when compared to BStream and recent algorithms proposed for multiple keyword-based query processing over XML streams [Barros et al., 2012a]. SLCAStream, our third SLCA algorithm, extends BStream by using unexplored LCA stream processing properties for query evaluation instead of a bitmap strategy. In addition, it incorporates optimization strategies that improve overall performance when compared to MKStream.

For the ELCA semantics, we proposed two algorithms, which are the first ones in the literature. The first algorithm, ELCABStream, is a basic ELCA-based implementation obtained from MKStream. However, it uses a single bitmap and a parsing stack for query evaluation. ELCAStream enhances ELCABStream by using the same unexplored LCA stream processing properties for query evaluation adopted by SLCAStream [Barros et al., 2012b]. It also uses a single parsing stack for query evaluation and incorporates optimization strategies that improve overall performance.

SLCAStream and ELCAStream are the latest in their respective semantics and the most efficient alternatives for processing multiple keyword-based queries over XML streams. However, both adopt a single parsing stack. Preliminary experiments involving multiple parsing stacks show that SLCAStream and ELCAStream performances can be improved. We have just finalized the implementation and response time evaluation of SLCAStream and ELCAStream extensions, called SLCAStream+ and ELCAStream+, that are based on multiple parsing stacks. Figure 7.1 presents a significant performance gain when SLCAStream (left chart) and ELCAStream (right chart) are extended by adopting multiple parsing stacks. Specifically, results in Figure 7.1 consider 50,000 queries with five terms run on the SIGMOD dataset. Multiple parsing stacks provide such gains since they allow the distribution of all submitted queries among parsing stacks which are only evaluated when necessary, thus decreasing the number query evaluations.

Specifically, the multiple parsing stack approach allows adjusting the number



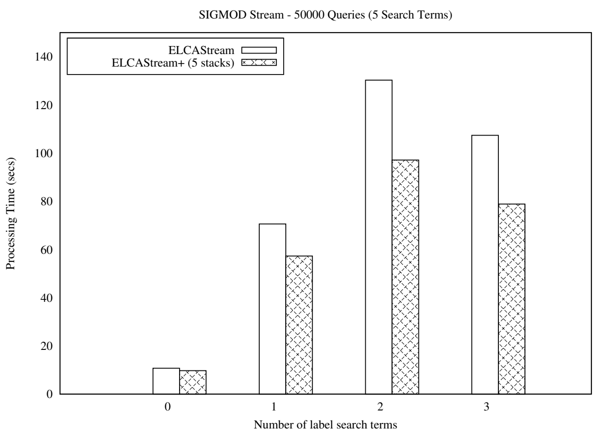


Figure 7.1. Preliminary results for SLCAStream and ELCAStream when using simultaneous multiple parsing stacks.

of parsing stacks for a better trade-off between processing time and memory usage. Moreover, this approach allows parallel processing via multithreaded and multicore CPUs [Sodan et al., 2010], resulting in additional performance increase. In a mono processing environment, multiple parsing stacks requiring evaluation are processed sequentially. In multithreaded and multicore environments, all parsing stacks can be processed simultaneously, each stack being processed by a thread or core.

This thesis also presents an accuracy evaluation for the SLCA and ELCA semantics [Barros et al., 2012b] as well as algorithms and strategies for ranking query results from XML streams. Regarding the accuracy evaluation, we evaluated the SLCA and ELCA semantics for precision and recall. Both semantics are adopted in our multiple keyword-based query processing algorithms. This evaluation has shown that ELCA semantics improves precision slightly and recall significantly compared to the SLCA semantics. This occurs when query keywords occur simultaneously in different document parts. Basically, this heuristic prioritizes the shortest resulting subtrees and those with larger query term frequency, followed by other resulting subtrees.

Regarding raking algorithms and heuristics, we proposed the algorithm LCARank and the heuristics SLCARank and StreamRank. LCARank implements a basic ranking strategy for keyword-based query results over XML streams. This new strategy combines the XRANK and SLCA algorithms [Barros et al., 2010] and prioritizes SLCA results among XRANK results, since SLCA returns smaller subtrees. We demonstrated experimentally that LCARank, despite its simple ranking strategy, improves results when compared to the original XRANK algorithm. We also proposed the SLCARank strategy which is a fine-grained ranking strategy compared to LCARank. LCARank and SLCARank focus on ranking the XML nodes returned by a keyword query submited to a single XML document. Finally, we proposed the StreamRank ranking strategy, which is applicable to multiple query results obtained from a set of streams defined by a time slot or specific number of documents.

As future work, we plan to implement SLCARank and StreamRank ranking strategies and evaluate their accuracy. Additionally, we plan to develop a complete parallel framework for processing multiple queries over XML streams based on our algorithms and heuristics. We also plan to incorporate our algorithms to Watershed [Ramos et al., 2011], a distributed computing framework developed at the UFMG Department of Computer Science. Currently, this framework only accepts queries written in structured query languages. By adopting a keyword-based query language, Watershed would cover more applications and users since this kind of query language requires no language syntax knowledge and minimal or no schema knowledge to formulate queries.

Bibliography

[Altinel and Franklin, 2000] Altinel, M. and Franklin, M. J. (2000). Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 53--64, Hong Kong, China.

[Babcock et al., 2002] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and Issues in Data Stream Systems. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 1--16, Wisconsin, USA.

[Bao et al., 2009] Bao, Z., Ling, T. W., Chen, B., and Lu, J. (2009). Effective XML Keyword Search with Relevance Oriented Ranking. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 517--528, Shangai, China.

[Bao et al., 2010] Bao, Z., Lu, J., and Ling, T. W. (2010). XReal: an Interactive XML Keyword Searching. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, pages 1933--1934, Montreal, Canada.

[Barros et al., 2012a] Barros, E. G., Hummel, F. C., Silva, A. S., Moro, M. M., and Laender, A. H. F. (2012a). Algorithms for Efficiently Processing Multiple Keyword Queries over XML Streams. *Submitted for publication.*

[Barros et al., 2010] Barros, E. G., Moro, M. M., and Laender, A. H. F. (2010). An Evaluation Study of Search Algorithms for XML Streams. *Journal of Information and Data Management*, 1(3):507 -- 522.

[Barros et al., 2012b] Barros, E. G., Moro, M. M., Laender, A. H. F., and Silva, A. S. (2012b). Efficient LCA-based Algorithms for Processing Multiple Keyword Queries over XML Streams. *Submitted for publication.*

[Bigonha et al., 2012] Bigonha, C., Cardoso, T., Moro, M., Gonçalves, M., and Almeida, V. (2012). Sentiment-based Influence Detection on Twitter. *Journal of the Brazilian Computer Society*, 18:169–183.

[Brin and Page, 1998] Brin, S. and Page, L. (1998). The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, 30(1-7):107--117.

[Carney et al., 2002] Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. (2002). Monitoring Streams: a New Class of Data Management Applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 215--226, Hong Kong, China.

[Chan et al., 2002] Chan, C.-Y., Felber, P., Garofalakis, M., and Rastogi, R. (2002). Efficient Filtering of XML Documents with XPath Expressions. *International Journal on Very Large Data Bases*, 11(4):354–379.

[Chen et al., 2006] Chen, Y., Davidson, S. B., and Zheng, Y. (2006). An Efficient XPath Query Processor for XML Streams. In *Proceedings of the 22nd International Conference on Data Engineering*, page 79, Atlanta, USA.

[Chen et. al, 2008] Chen et. al, S. (2008). Scalable Filtering of Multiple Generalized-Tree-Pattern Queries over XML Streams. *IEEE Transactions on Knowledge and Data Engineering*, 20(12):1627–1640.

[Cohen et al., 2003] Cohen, S., Mamou, J., Kanza, Y., and Sagiv, Y. (2003). XSearch: A Semantic Search Engine for XML. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 45--56, Berlin, Germany.

[Cortes et al., 2000] Cortes, C., Fisher, K., Pregibon, D., Rogers, A., and Smith, F. (2000). Hancock: A Language for Extracting Signatures from Data Streams. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 9--17, Boston, USA.

[Diao et al., 2002] Diao, Y., Fischer, P., Franklin, M. J., and To, R. (2002). YFilter: Efficient and Scalable Filtering of XML Documents. In *Proceedings of the 18th International Conference on Data Engineering*, page 341, San Jose, California.

[Diao et al., 2004] Diao, Y., Rizvi, S., and Franklin, M. J. (2004). Towards an Internet-Scale XML Dissemination Service. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 612--623, Toronto, Canada.

[Duffield and Grossglauser, 2001] Duffield, N. G. and Grossglauser, M. (2001). Trajectory sampling for direct traffic observation. *IEEE/ACM Transactions on Networking*, 9(3):280--292.

[Franceschet, 2005] Franceschet, M. (2005). XPathMark: An XPath Benchmark for the XMark Generated Data. In *Proceedings of the 3th International Conference on Database and XML Technologies*, pages 129–143, Trondheim, Norway.

[Gedik et al., 2008] Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M. (2008). SPADE: The System S Declarative Stream Processing Engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1123--1134, Vancouver, Canada.

[Gou and Chirkova, 2007a] Gou, G. and Chirkova, R. (2007a). Efficient Algorithms for Evaluating XPath over Streams. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 269--280, Beijing, China.

[Gou and Chirkova, 2007b] Gou, G. and Chirkova, R. (2007b). Efficiently Querying Large XML Data Repositories: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1381--1403.

[Green et al., 2004] Green, T. J., Gupta, A., Miklau, G., Onizuka, M., and Suciu, D. (2004). Processing XML Streams with Deterministic Automata and Stream Indexes. *ACM Transactions Database Systems.*, 29(4):752--788.

[Guo et al., 2003] Guo, L., Shao, F., Botev, C., and Shanmugasundaram, J. (2003). XRANK: Ranked Keyword Search over XML Documents. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 16--27, California, USA.

[Gupta and Suciu, 2003] Gupta, A. K. and Suciu, D. (2003). Stream Processing of XPath Queries with Predicates. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 419--430, California, USA.

[Hu and Chou, 2009] Hu, C.-L. and Chou, C.-K. (2009). RSS Watchdog: an Instant Event Monitor on Real Online News Streams. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, pages 2097--2098, Hong Kong, China.

[Hummel et al., 2011] Hummel, F. C., Silva, A. S., Moro, M. M., and Laender, A. H. F. (2011). Multiple keyword-based queries over XML streams. In *Proceedings of the*

20th ACM International Conference on Information and Knowledge Management, pages 1577–1582, Glasgow, UK.

[Järvelin and Kekäläinen, 2002] Järvelin, K. and Kekäläinen, J. (2002). Cumulated Gain-based Evaluation of IR Techniques. *ACM Transactions on Information Systems*, 20(4):422--446.

[Lee and Lee, 2009] Lee, H.-H. and Lee, W.-S. (2009). Selectivity-sensitive Shared Evaluation of Multiple Continuous XPath Queries over XML Streams. *Journal Information Sciences*, 179:1984--2001.

[Lenkov, 2003] Lenkov, D. (2003). Binary XML - Position paper for The W3C Workshop on Binary Interchange of XML Information Item Sets. Website. `http://www.w3.org/2003/08/binary-interchange-workshop/31-oracle-BinaryXML_pos.htm`.

[Li et al., 2007] Li, G., Cheung, A., Hou, S., Hu, S., Muthusamy, V., Sherafat, R., Wun, A., Jacobsen, H.-A., and Manovski, S. (2007). Historic Data Access in Publish/Subscribe. In *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*, pages 80--84, Ontario, Canada.

[Li et al., 2008] Li, G., Hou, S., and Jacobsen, H.-A. (2008). Routing of XML and XPath Queries in Data Dissemination Networks. In *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, pages 627--638, Beijing, China.

[Li et al., 2010] Li, J., Liu, C., Zhou, R., and Wang, W. (2010). Suggestion of Promising Result Types for XML Keyword Search. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 561--572, Lausanne, Switzerland.

[Liu and Chen, 2008] Liu, Z. and Chen, Y. (2008). Reasoning and Identifying Relevant Matches for XML Keyword Search. *Proceedings of the VLDB Endowment*, 1(1):921--932.

[Liu and Chen, 2011] Liu, Z. and Chen, Y. (2011). Processing Keyword Search on XML: a Survey. *World Wide Web*, 14(5-6):671–707.

[Madden and Franklin, 2002] Madden, S. and Franklin, M. (2002). Fjording the Stream: an Architecture for Queries over Streaming Sensor Data. In *Proceedings of the 18th International Conference on Data Engineering*, pages 555 –566, California, USA.

[McCafferty, 2011] McCafferty, D. (2011). Brave, New Social World. *Communications of the ACM*, 54(7):19--21.

[Min et al., 2007] Min, J.-K., Park, M.-J., and Chung, C.-W. (2007). XTREAM: An Efficient Multi-query Evaluation on Streaming XML Data. *Information Sciences*, 177(17):3519--3538.

[Moro et al., 2007] Moro, M. M., Bakalov, P., and Tsotras, V. J. (2007). Early Profile Pruning on XML-aware Publish-Subscribe Systems. In *Proceedings of the 33rd International Conference on Ver Large Data Bases*, pages 866–877, Vienna, Austria.

[Olteanu, 2007] Olteanu, D. (2007). SPEX: Streamed and Progressive Evaluation of XPath. *IEEE Transactions on Knowledge and Data Engineering*, 19(7):934--949.

[Onizuka, 2010] Onizuka, M. (2010). Processing XPath Queries with Forward and Downward Axes over XML Streams. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 27--38, Lausanne, Switzerland.

[Park et al., 2009] Park, H. K., Shin, S. J., Na, S. H., and Lee, W. S. (2009). M-COPE: A Multiple Continuous Query Processing Engine. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, pages 2065--2066.

[Peng and Chawathe, 2005] Peng, F. and Chawathe, S. S. (2005). XSQ: A Streaming XPath Engine. *ACM Transactions Database Systems*, 30(2):577–623.

[Ramanan, 2009] Ramanan, P. (2009). Worst-case Optimal Algorithm for XPath Evaluation over XML Streams. *Journal of Computer and System Sciences*, 75(8):465 – 485.

[Ramos et al., 2011] Ramos, T. L. A. S., Oliveira, R. S., Carvalho, A. P., Ferreira, R. A. C., and Meira, W. (2011). Watershed: A High Performance Distributed Stream Processing System. In *Proceedings of the 2011 23rd International Symposium on Computer Architecture and High Performance Computing*, pages 191--198, Washington, USA.

[Salton and Buckley, 1988] Salton, G. and Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513 – 523.

[Schmidt et al., 2001] Schmidt, A., Kersten, M., and Windhouwer, M. (2001). Querying XML Documents Made Easy: Nearest Concept Queries. In *Proceedings of the*

17th International Conference on Data Engineering, pages 321--329, Heidelberg, Germany.

[Singh et al., 2008] Singh, J., Eyers, D. M., and Bacon, J. (2008). Controlling Historical Information Dissemination in Publish/Subscribe. In *Proceedings of the 2008 Workshop on Middleware Security*, pages 34--39, Leuven, Belgium.

[Sodan et al., 2010] Sodan, A., Machina, J., Deshmeh, A., Macnaughton, K., and Esbaugh, B. (2010). Parallelism via Multithreaded and Multicore CPUs. *Computer*, 43(3):24 –32.

[Sourlas et al., 2009] Sourlas, V., Paschos, G. S., Flegkas, P., and Tassiulas, L. (2009). Caching in Content-based Publish/Subscribe Systems. In *Proceedings of the 28th IEEE Conference on Global Telecommunications*, pages 1401--1406, Hawaii, USA.

[Sun et al., 2007] Sun, C., Chan, C.-Y., and Goenka, A. K. (2007). Multiway SLCA-based Keyword Search in XML Data. In *Proceedings of the 16th International Conference on World Wide Web*, pages 1043--1052, Alberta, Canada.

[Termehchy and Winslett, 2009] Termehchy, A. and Winslett, M. (2009). Effective, Design-Independent XML Keyword Search. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, pages 107--116, Hong Kong, China.

[Tian et al., 2011] Tian, Z., Lu, J., and Li, D. (2011). A Survey on XML Keyword Search. In *Web Technologies and Applications*, volume 6612 of *Lecture Notes in Computer Science*, pages 460–471. Springer.

[Vagena et al., 2007a] Vagena, Z., Colby, L. S., Özcan, F., Balmin, A., and Li, Q. (2007a). On the Effectiveness of Flexible Querying Heuristics for XML Data. In *Procedings of the 5th International XML Database Symposium*, pages 77–91, Viena, Austria.

[Vagena and Moro, 2008] Vagena, Z. and Moro, M. M. (2008). Semantic Search over XML Document Streams. In *Proceedings of 3rd International Workshop on Database Technologies for Handling XML Information on the Web*, Nantes, France.

[Vagena et al., 2007b] Vagena, Z., Moro, M. M., and Tsotras, V. J. (2007b). RoXSum: Leveraging Data Aggregation and Batch Processing for XML Routing. In *Proceedings of the 23rd International Conference on Data Engineering*, pages 1466–1470, Istanbul, Turkey.

[Wilde and Glushko, 2008] Wilde, E. and Glushko, R. J. (2008). XML Fever. *Commun. ACM*, 51(7):40–46.

[Wu and Theodoratos, 2012] Wu, X. and Theodoratos, D. (2012). A Survey on XML Streaming Evaluation Techniques. *The VLDB Journal*, pages 1--26. Published online: 10 June, 2012.

[Xu and Papakonstantinou, 2005] Xu, Y. and Papakonstantinou, Y. (2005). Efficient Keyword Search for Smallest LCAs in XML Databases. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 537–538, Maryland, USA.

[Xu and Papakonstantinou, 2008] Xu, Y. and Papakonstantinou, Y. (2008). Efficient LCA-based Keyword Search in XML Data. In *Proceedings of the 11th International Conference on Extending Database Technology*, pages 535--546, Nantes, France.

[Zhou et al., 2010] Zhou, R., Liu, C., and Li, J. (2010). Fast ELCA Computation for Keyword Queries on XML Data. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 549--560, Lausanne, Switzerland.