Keyword Proximity Search in XML Trees

Vagelis Hristidis
Florida International University
vagelis@cs.fiu.edu

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

Yannis Papakonstantinou
UC, San Diego
yannis@cs.ucsd.edu

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Abstract

Recent works have shown the benefits of keyword proximity search in querying XML documents in addition to text documents. For example, given query keywords over Shakespeare's plays in XML, the user might be interested in knowing *how* the keywords co-occur. In this paper, we focus on XML trees and define XML keyword proximity queries to return the (possibly heterogeneous) set of *minimum connecting trees* (MCTs) of the matches to the individual keywords in the query. We consider efficiently executing keyword proximity queries on labeled trees (XML) in various settings: (i) when the XML database has been pre-processed, and (ii) when no indices are available on the XML database. We perform a detailed experimental evaluation to study the benefits of our approach, and show that our algorithms considerably outperform prior algorithms and other applicable approaches.

1 Introduction

Keyword search is a user-friendly information discovery technique that has been extensively studied for text documents. Keyword proximity search is well-suited to XML documents as well, which are often modeled as labeled trees [3]. For example, consider a document consisting of (marked up) Shakespeare's plays in XML. A user might be interested in matching the query keywords "mother, king, brother", and determine where they co-occur and within what context. For example, they may all appear within the same line or it may be that "king" and "brother" appear in a line of a speech and "mother" appears in another line of the same speech, and so on.

In the case of XML trees, the problem of keyword proximity search reduces to the problem of finding the subtrees rooted at the lowest common ancestors (LCAs) of the XML nodes that contain the keywords. Recently, a large corpus of work [18, 14, 19, 20] has been conducted on efficiently finding the LCAs of the query keyword nodes in XML trees.

However, these works focus on computing the LCA nodes and not the whole XML subtrees rooted at the LCA nodes. These subtrees are needed in order to rank the results and display them to the user, since ranking typically depends on the types of the connections. Furthermore, Xu and Papakonstantinou [20] and Li et al. [18] provide efficient algorithms for locating only the *Smallest LCAs* (see Section 6).

This paper presents algorithms to compute the *Minimum Connecting Trees (MCTs)* of the nodes that contain the keywords, that is, the subtrees rooted at the LCAs of the nodes that contain the keywords. We make the following technical contributions:

- We formulate two main problems: (i) identifying and presenting in a compact manner all MCTs, which explain how the keywords are connected, and (ii) identifying only MCTs whose root is not an ancestor of the root of another MCT.

- We design and analyze efficient algorithms to compute MCTs, in two cases: (i) when the XML data has been pre-processed and relevant indices have been constructed, and (ii) when the XML data has not been pre-processed, i.e., the XML data can only be processed sequentially.

- We perform a detailed experimental evaluation to study the benefits of our approach, and show that our algorithms considerably outperform both prior algorithms for keyword proximity on labeled graphs [7, 17, 13] as well as other applicable approaches.

Notice that this work only focuses on how to efficiently return the connections between the nodes that contain the keywords. However, similarly to previous LCA works [20, 18], it does not solve the problem of how to rank these connections. Intuitively, the MCT is the basic connecting component between objects of a tree although the specific strength of this connection has its own merit. The ranking problem has been studied in previous works [14, 7, 12]. The combination of our execution framework with these ranking techniques is left as future work.

The rest of this paper is organized as follows. We describe the notation we use and formulate the problems in Section 2. Our algorithms for the case of indexed XML data are presented in Section 3, and for unindexed data in Section 4. We present a detailed experimental evaluation of our algorithms in Section 5. Related work is discussed in Section 6, and we conclude with directions for further work in Section 7.

2 Framework

2.1 Notation

We use the conventional labeled directed tree notation to represent XML documents. Each node v of the tree corresponds to an XML element and is labeled with a tag $\lambda(v)$. If v is a leaf node it also has a string value $val(v)$ that contains a list of keywords. We assume that each node v has a unique id $id(v)$. Figure 1 illustrates a tree that will be used in the examples. $id(v)$ is the first component of the 4-tuple associated with each node v. The other three components will be explained in Section 3.2, where we first make use of these components.

A *keyword query* is simply a set of keywords $k_1, \ldots, k_m$. It returns a compact representation of the set of trees that connect the nodes that contain the keywords in their value or their tag. The following discussion formally defines and motivates the semantics.

Definition 2.1 (MCT and LCA) *The* minimum connecting tree (MCT) *of nodes* $v_1, \ldots, v_m$ *of the input labeled tree* T *is the minimum size subtree* T_M *of* T *that connects* $v_1, \ldots, v_m$.

The root of the tree is called the lowest common ancestor *(LCA) of the nodes* $v_1, \ldots, v_m$. ■

An MCT of keywords $k_1, \ldots, k_m$ is an MCT of nodes $v_1, \ldots, v_m$ that contain the keywords. For example, the MCTs (1) and (2) are two of the MCTs of the query “Tom, Harry”, and the MCTs (3), (4) and (5) correspond to the query “Tom, Dick, Harry”.

$$a_1 \leftarrow p_1 \rightarrow a_2 \tag{1}$$

$$a_8 \leftarrow p_4 \leftarrow s_3 \rightarrow p_5 \rightarrow a_9 \tag{2}$$

- $[r, 1, 42, 0]$ `root`
  - $[c_1, 2, 41, 1]$ `conference`
    - $[s_1, 3, 16, 2]$ `session`
      - $[p_1, 4, 9, 3]$ `paper`
        - $[a_1, 5, 6, 4]$ `author (Harry Smith)`
        - $[a_2, 7, 8, 4]$ `author (Tom Jones)`
      - $[p_2, 10, 15, 3]$ `paper`
        - $[a_3, 11, 12, 4]$ `author (Tom Brown)`
        - $[a_4, 13, 14, 4]$ `author (Dick Smith)`
    - $[s_2, 17, 26, 2]$ `session`
      - $[p_3, 18, 25, 3]$ `paper`
        - $[a_5, 19, 20, 4]$ `author (Tom Green)`
        - $[a_6, 21, 22, 4]$ `author (Harry Brown)`
        - $[a_7, 23, 24, 4]$ `author (Dick Jones)`
    - $[s_3, 27, 40, 2]$ `session`
      - $[p_4, 28, 31, 3]$ `paper`
        - $[a_8, 29, 30, 4]$ `author (Harry Jones)`
      - $[p_5, 32, 35, 3]$ `paper`
        - $[a_9, 33, 34, 4]$ `author (Tom Smith)`
      - $[p_6, 36, 39, 3]$ `paper`
        - $[a_{10}, 37, 38, 4]$ `author (Dick Brown)`

Figure 1: Input Labeled Tree Used in Examples

$$\begin{tikzcd}
 & & s_1 \arrow[dl] \arrow[dr] & & \\
 & p_1 \arrow[dl] \arrow[dr] & & p_2 \arrow[dr] & \\
a_1 & & a_2 & & a_4
\end{tikzcd} \tag{3}$$

$$\begin{tikzcd}
 & p_3 \arrow[dl] \arrow[d] \arrow[dr] & \\
a_5 & a_6 & a_7
\end{tikzcd} \tag{4}$$

$$\begin{tikzcd}
 & s_3 \arrow[dl] \arrow[d] \arrow[dr] & \\
p_4 \arrow[d] & p_5 \arrow[d] & p_6 \arrow[d] \\
a_8 & a_9 & a_{10}
\end{tikzcd} \tag{5}$$

According to the typical assumption of keyword proximity systems [7, 13, 17, 16, 4], smaller MCTs are considered better solutions since they provide a closer connection between the keywords. However, our framework and algorithms are not tied to a particular ranking function, since we focus on efficiently generating all the MCTs. In our running example, MCT (1) is better than MCT (2) since MCT (1) shows that Tom and Harry are co-authors while MCT (2) merely shows that they both had papers in the same session of the conference. Similarly, MCT (3) is better than MCT (5), since MCT (5) shows that the 3 authors are linked through 3 different papers in the same session, while MCT (3) shows that they are linked through only 2 different papers in the same session. Indeed, we will later augment our keyword queries to bound the size of the MCTs, since beyond a size the result is often uninteresting.

The set of MCTs is often overwhelmingly large since it may contain the following form of data redundancy, which leads to a number of MCTs that is exponential in the number of keywords in the query. Consider a list l_1 of nodes that contain k_1, a list l_2 of nodes that contain k_2, and so on, up to a list l_m of nodes containing k_m. Suppose node n is the pair-wise LCA of the nodes of the m lists and all nodes are at equal distances from n. In our running example, there is such a list $[a_2, a_3]$ of "Tom" nodes ($|l_1| = 2$) and a list $[a_6, a_8]$ of "Harry" nodes ($|l_2| = 2$), such that their common LCA is c_1 (conference). Then there are $|l_1| \times |l_2| \times \ldots \times |l_m|$ MCTs. Notice that if there are i, j such that $|l_i| > 1$ and $|l_j| > 1$, then each MCT can be implied (inferred) by the other MCTs and the set of MCTs is redundant. For example, the MCTs

$$a_2 \leftarrow p_1 \leftarrow s_1 \leftarrow c_1 \rightarrow s_2 \rightarrow p_3 \rightarrow a_6 \tag{6}$$

$$a_3 \leftarrow p_2 \leftarrow s_1 \leftarrow c_1 \rightarrow s_3 \rightarrow p_4 \rightarrow a_8 \tag{7}$$

of query "Tom, Harry" together imply the MCTs

$$a_2 \leftarrow p_1 \leftarrow s_1 \leftarrow c_1 \rightarrow s_3 \rightarrow p_4 \rightarrow a_8 \tag{8}$$

$$a_3 \leftarrow p_2 \leftarrow s_1 \leftarrow c_1 \rightarrow s_2 \rightarrow p_3 \rightarrow a_6 \tag{9}$$

The encoding of the set of MCTs in *grouped distance trees* resolves this problem. We first define distance MCTs.

Definition 2.2 (DMCT) *Consider nodes $v_1, \ldots, v_m$ of the input tree T. The* Distance MCT (DMCT) *$T_D = d(T_M)$ of the MCT T_M of nodes $v_1, \ldots, v_m$ is the minimum node-labeled and edge-labeled tree such that:*

1. *T_D contains the nodes $v_1, \ldots, v_m$,*

2. *T_D contains the LCAs $u_1, \ldots, u_k$ of any pair of nodes (v_i, v_j) where $v_i, v_j \in [v_1, \ldots, v_m]$, $i \neq j$,*

3. *there is an edge labeled with the number ℓ between any two distinct nodes $n, n' \in \{v_1, \ldots, v_m, u_1, \ldots, u_k\}$ if there is a path of length ℓ from n' to n in T_M, and the path does not contain any node $n'' \in \{u_1, \ldots, u_m\}$ other than n and n'.* ■

The DMCT (10) corresponds to the MCT (1) and the DMCTs (11-14) correspond to the MCTs (6-9).

$$a_1 \xleftarrow{1} p_1 \xrightarrow{1} a_2 \tag{10}$$

$$a_2 \xleftarrow{3} c_1 \xrightarrow{3} a_6 \tag{11}$$

$$a_3 \xleftarrow{3} c_1 \xrightarrow{3} a_8 \tag{12}$$

$$a_2 \xleftarrow{3} c_1 \xrightarrow{3} a_8 \tag{13}$$

$$a_3 \xleftarrow{3} c_1 \xrightarrow{3} a_6 \tag{14}$$

Notice that the exponential explosion in the number of keywords is still present. *Grouped DMCTs* resolve the problem (if possible) by grouping together DMCTs of the same structure.

Definition 2.3 (GDMCT) *A* Grouped DMCT *of a tree T is a labeled tree where edges are labeled with numbers and nodes are labeled with lists of node id's from T.*

A DMCT D belongs *to a GDMCT G if D and G are isomorphic. Assuming that f is the mapping of the nodes of D to the nodes of G, which induces a corresponding mapping, also called f, of the edges of D to the edges of G, the following must hold:*

1. *if n_D is a node of D, n_G is a node of G and $f(n_D) = n_G$ then the label of n_G contains the id of n_D,*

2. *if e_D is an edge of D, e_G is an edge of G and $f(e_D) = e_G$ then the label of e_D and the label of e_G are the same number.* ■

The GDMCT (15) captures DMCTs (11-14). The notation $u_1[a_2, a_3]$ indicates that the label of the node u_1 is $[a_2, a_3]$.

$$u_1[a_2, a_3] \xleftarrow{3} u_0[c_1] \xrightarrow{3} u_2[a_6, a_8] \tag{15}$$

Note that each tree that is an instance of a GDMCT and is also a subtree of the XML data tree T is a DMCT of an MCT of T.

We define the size of a GDMCT (or DMCT) to be the sum of the weights of its edges. We often eliminate from the solution those trees whose sizes exceed a user-provided size threshold K.

2.2 Problems

We consider two closely related keyword search problems in this paper.

Problem 1 (All GDMCTs Problem) *Given an input labeled tree T, keywords $k_1, \ldots, k_m$, and an integer K, find the minimal set of tuples (n, G), where G is a GDMCT whose root has list label $[n]$ such that:*

1. *n is an LCA of $k_1, \ldots, k_m$,*
2. *each DMCT D of size up to K rooted at node n that is an LCA of $k_1, \ldots, k_m$ belongs to at least one GDMCT G, such that (n, G) is a tuple,*
3. *if any node id n_i is removed from the label $[n_1, \ldots, n_i, \ldots, n_m]$ of a node $n' \in G$ of a tuple (n, G) then there is at least one DMCT D of size up to K that does not belong to any tuple though it is rooted at the LCA n of $k_1, \ldots, k_m$.*
4. *every node n_i of the label $[n_1, \ldots, n_i, \ldots, n_m]$ of a node n' contains the same subset S of keywords from $k_1, \ldots, k_m$.*[1]
5. *the size of G is no more than K.* ■

[1]This condition ensures that each DMCT D contained in the GDMCT (that is, D is also contained in T) contains all keywords $k_1, \ldots, k_m$.

The query "Tom, Harry", with $K = 5$ returns the relation (16), while the same query with $K = 3$ returns (17).

$$\begin{aligned}
\{\quad & (p_1, \quad u_1^1[a_1] \xleftarrow{1} u_0^1[p_1] \xrightarrow{1} u_2^1[a_2]) \\
& (s_1, \quad u_1^2[a_1] \xleftarrow{2} u_0^2[s_1] \xrightarrow{2} u_2^2[a_3]) \\
& (p_3, \quad u_1^3[a_5] \xleftarrow{1} u_0^3[p_3] \xrightarrow{1} u_2^3[a_6]) \\
& (s_3, \quad u_1^4[a_8] \xleftarrow{2} u_0^4[s_3] \xrightarrow{2} u_2^4[a_9]) \quad \}
\end{aligned} \tag{16}$$

$$\begin{aligned}
\{\quad & (p_1, \quad u_1^1[a_1] \xleftarrow{1} u_0^1[p_1] \xrightarrow{1} u_2^1[a_2]) \\
& (p_3, \quad u_1^3[a_5] \xleftarrow{1} u_0^3[p_3] \xrightarrow{1} u_2^3[a_6]) \quad \}
\end{aligned} \tag{17}$$

A closely related problem to Problem 1, discussed next, is one which returns only GDMCTs whose roots (i.e., the LCAs) are not themselves ancestors of roots of other returned GMDCTs.

Problem 2 (Lowest GDMCTs Problem) *Given an input labeled tree T, keywords $k_1, \ldots, k_m$, and an integer K, find the minimal set of tuples (n, G), such that:*

1. *(n, G) is a tuple for Problem 1, i.e., the All GDMCTs Problem, and*
2. *if (n', G') is also a tuple for Problem 1, then n is not an ancestor of n'.* ■

For Problem 2, the query "Tom, Harry", with $K = 3$ still returns (17), while the same query with $K = 5$ returns the relation (18). Note that the tuple with $n = s_1$ from the relation (16) is no longer a solution for the Lowest GDMCTs Problem since it is an ancestor of node p_1 which is part of a solution.

$$\begin{aligned}
\{\quad & (p_1, \quad u_1^1[a_1] \xleftarrow{1} u_0^1[p_1] \xrightarrow{1} u_2^1[a_2]) \\
& (p_3, \quad u_1^3[a_5] \xleftarrow{1} u_0^3[p_3] \xrightarrow{1} u_2^3[a_6]) \\
& (s_3, \quad u_1^4[a_8] \xleftarrow{2} u_0^4[s_3] \xrightarrow{2} u_2^4[a_9]) \quad \}
\end{aligned} \tag{18}$$

In this paper, we focus our attention on these two problems. We also consider variants of Problems 1 and 2, where we are interested in returning only the LCAs (not the complete GDMCTs), provided there is at least one DMCT rooted at the LCA with size no more than K. We refer to these variants as the "All LCAs Problem" and the "Lowest LCAs Problem" in the paper.

Notice that, in practice, one may augment GDMCTs with additional information about their nodes. For example, one may ask that the title of the paper is always displayed along with the paper. [17] has introduced the “target objects” concept to handle this requirement. For simplicity, we will neglect such augmentations since they do not affect the performance issues that are the focus of this paper.

In the sequel, we design efficient algorithms for these problems, and experimentally evaluate them, under two cases: (i) when the XML data has been pre-processed and relevant indices have been constructed before the keyword query is evaluated (Section 3), and (ii) when the XML data has not been pre-processed, i.e., the XML data can only be processed sequentially (Section 4).

3 Algorithms: Indexed XML Data

In this section, we first focus on Problem 1 (All GDMCTs), and design two competitive algorithms to solve it: a straightforward, nested-loops algorithm, and a more sophisticated stack-based algorithm that is tailored to the XML tree structure in identifying LCAs and GDMCTs. We then discuss the modifications to our stack-based algorithm that are needed to solve the variants (Lowest GDMCTs, All LCAs and Lowest LCAs) of our core problem. These algorithms are compared experimentally in Section 5.

3.1 All GDMCTs: Nested Loops Algorithm

Intuitively, the nested loops algorithm (`NL`) for the case of indexed XML data operates over separate lists of nodes, $L(k)$, one for each query keyword, k, to identify the GDMCTs whose sizes are no more than the user-provided threshold, K. The master index for the nested loops algorithm is organized as an inverted index, as follows. A hash table (the keywords are the keys) of all the keywords in the XML data tree T is created and for each keyword k we keep a list $L(k)$ (value of hash table) of the nodes n of T that contain k, where each node n is stored with its *path-id*: the list of node ids along the path from the root of T to n. This choice facilitates the easy identification of the LCA and the GDMCT of a set of nodes, which can be determined by simply examining the path-ids of the respective nodes. This index is built in one pass over T before any query arrives. For example, some entries in the master index for the XML tree of Figure 1 are shown below.

```
Nested-Loops Algorithm( $k_1, \ldots, k_m, K$ ) {
$R$: array of result GDMCTs;
For $i = 1, \ldots, m$ do
    Get $L(k_i)$ from master index;
For each combination $(u_1, \ldots, u_m)$ with $u_i \in L(k_i)$ do
    $mct$ = getMCT($u_1, \ldots, u_m$);
    if (size($mct$) $\leq K$)
      Add $mct$ to its corresponding GDMCT in $R$ if such
      a GDMCT exists, else create a new GDMCT in $R$;
Return R;
}

getMCT( $u_1, \ldots, u_m$ ) {
Let $p_i$ denote the path-id of $u_i$;
$depth = 0$;
While $p_1[depth] = \ldots = p_m[depth]$ do
    $depth++$;
LCA $= p_1[depth - 1]$;
Return the MCT rooted at LCA;
}
```

Figure 2: Nested Loops Algorithm

`Tom:` $[[r, c_1, s_1, p_1, a_2], [r, c_1, s_1, p_2, a_3], [r, c_1, s_2, p_3, a_5], [r, c_1, s_3, p_5, a_9]]$

`Dick:` $[[r, c_1, s_1, p_2, a_4], [r, c_1, s_2, p_3, a_7], [r, c_1, s_3, p_6, a_{10}]]$

`Harry:` $[[r, c_1, s_1, p_1, a_1], [r, c_1, s_2, p_3, a_6], [r, c_1, s_3, p_4, a_8]]$

The execution stage of the Nested Loops Algorithm, using this index, is presented in Figure 2. Essentially, it checks all combinations of nodes from the keyword lists, computes an MCT (minimum connecting tree) for each combination, and then merges the resulting MCT into the list of result GDMCTs, provided its size is within the user-specified threshold.

For example, given the keyword query “Tom, Harry”, and a threshold $K = 3$, the Nested Loops algorithm would examine the 12 node-pairs in the cross-product of the index entries for Tom and Harry, compute 12 MCTs, determine that only 2 of them meet the threshold, and finally return two GDMCTs (see relation (17)).

There are two main sources of inefficiency in the Nested Loops algorithm. First, as illustrated in the above example, it has to check all the combinations of nodes from the keyword lists, i.e., getMCT(.) is called $|L(k_1)| \times \cdots \times |L(k_m)|$ times. Second (not illustrated in the above example), the grouping of the results into GDMCTs is not tightly integrated with the algorithm and a lookup to the array R is required for each relevant MCT found.

We next present a stack-based algorithm that overcomes both these sources of inefficiency, is tailored to the XML tree structure in identifying GDMCTs, and delivers performance that is considerably better than the Nested Loops Algorithm.

3.2 All GDMCTs: Stack-Based Algorithm

Our stack-based algorithm, which we refer to as SA, makes use of a node numbering system, which associates `(start, end, depth)` numbers with each node in the XML tree, where `start` and `end` correspond to the first and the final times the node is visited in a depth-first traversal of the XML tree, and `depth` is the depth of the node from the root of the tree. In Figure 1, we depict the `(start, end, depth)` numbering with each node, as the last three components of the 4-tuple. For example, the numbering associated with s_1 is $(3, 16, 2)$. Such a numbering has been repeatedly utilized (see, e.g., [21, 5]), in a variety of XML related algorithms.

This numbering permits efficient checking of ancestor-descendant (or containment) relationships (by comparing containment of the corresponding `(start, end)` intervals), and can also be used to determine the *distance* between an ancestor and a descendant node in the XML tree (by computing the difference between corresponding `depth`s). This latter fact (only exploited in [21, 5] to check parent-child relationships) will be very useful for us to efficiently compute sizes of MCTs. For example, one can determine that s_1 is an ancestor of a_4 (since the interval $(3, 16)$ contains the interval $(13, 14)$) and also determine that the distance between them is 2 (i.e., $4 - 2$), without knowing the intermediate node between s_1 and a_4.

3.2.1 Index Structure and Algorithm

Intuitively, the stack-based algorithm for computing GDMCTs, on indexed XML data, operates over lists of nodes, two for each query keyword (these lists are described below). It

- maintains candidate LCA nodes on a stack,
- computes and maintains partial GDMCTs at each candidate LCA, for subsets of query keywords, and

- computes and outputs result GDMCTs when all descendant nodes of a candidate LCA are known to have been examined.

In order to do so, the lists associated with each keyword k need to contain, in addition to the nodes of T that contain k, ancestors of these nodes as well. This is because, while the `(start, end, depth)` numbers suffice to check ancestor-descendant relationships, they are insufficient to identify the lowest common ancestors. For example, one would not be able to determine that the lowest common ancestor of a_1 (with node numbering $(5, 6, 4)$) and a_3 (with node numbering $(11, 12, 4)$) is s_1 (with node numbering $(3, 16, 2)$).

Indexing by keyword is provided by the master index, which is organized as an inverted index, as follows. A hash table of all the keywords in the XML data tree T is created and for each keyword k we keep two lists:

- $L(k)$ of the nodes of T that contain k in T, and
- $L^a(k)$ of the ancestors of nodes in $L(k)$.

That is, the (master) index consists of two lists ($L(k)$ and $L^a(k)$) for each keyword. Each node is stored as $(id, start, end, depth)$, and $L(k)$ and $L^a(k)$ are sorted in ascending `start` order. This index is also built in one pass over T before any query arrives. For example, the entries for keywords Tom, Dick and Harry, in the index for the XML tree of Figure 1, are shown below.

`Tom:` $L = [(a_2, 7, 8, 4), (a_3, 11, 12, 4), (a_5, 19, 20, 4), (a_9, 33, 34, 4)]$
$L^a = [(r, 1, 42, 0), (c_1, 2, 41, 1), (s_1, 3, 16, 2), (p_1, 4, 9, 3), (p_2, 10, 15, 3), (s_2, 17, 26, 2), (p_3, 18, 25, 3), (s_3, 27, 40, 2), (p_5, 32, 35, 3)]$

`Dick:` $L = [(a_4, 13, 14, 4), (a_7, 23, 24, 4), (a_{10}, 37, 38, 4)]$
$L^a = [(r, 1, 42, 0), (c_1, 2, 41, 1), (s_1, 3, 16, 2), (p_2, 10, 15, 3), (s_2, 17, 26, 2), (p_3, 18, 25, 3), (s_3, 27, 40, 2), (p_6, 36, 39, 3)]$

`Harry:` $L = [(a_1, 5, 6, 4), (a_6, 21, 22, 4), (a_8, 29, 30, 4)]$
$L^a = [(r, 1, 42, 0), (c_1, 2, 41, 1), (s_1, 3, 16, 2), (p_1, 4, 9, 3), (s_2, 17, 26, 2), (p_3, 18, 25, 3), (s_3, 27, 40, 2), (p_4, 28, 31, 3)]$

While the L^a lists in this index are not present in the index for the nested loops algorithm, each entry in the L and L^a lists is small and of fixed size, unlike in the nested loops index (where the

```
Stack Algorithm SA( $k_1, \ldots, k_m, K$ ) {
    $S$: stack, where stack entry $s$ consists of ($s$.nodeID, $s$.GDMCTs), where $s$.GDMCTs is a list of GDMCTs;
1.  $L \leftarrow$ GetList($k_1, \ldots, k_m$);
2.  While $S$ not empty OR more nodes in $L$ do {
3.      $n \leftarrow$ Find node with smallest start value in $L$;
4.      While $S$ is not empty AND top($S$).end $<$ $n$.start do
5.          POP($S$);
6.      PUSH($n, S$);
} }
```

Figure 3: High Level Description of Stack Algorithm for All GDMCTs Problem

entry size depends on the length of the path from the root of the XML tree). The asymptotic size complexity of the master index for the Stack Algorithm is better than that of the master index for the Nested Loops Algorithm. This is because each ancestor of a node containing keyword k is represented only once in the Stack Algorithm's master index, whereas each ancestor is represented in the path-ids of the Nested Loops Algorithm's master index as many times as it has descendants that contain keyword k. Hence, generally deeper (resp. more shallow) trees require less (resp. more) storage for the SA master index, compared to the Nested Loops Algorithm index. We shall also show empirically, in Section 5, that the sizes of the master indices for the two algorithms are not substantially different.

We next describe the execution stage of the Stack Algorithm in more detail. To clarify the description and point out the novel contributions of the algorithm, we split it into two parts. The first part (Figure 3) describes how the selected list of nodes is traversed in a depth-first manner and the nodes are pushed and popped from the stack. This type of stack-based traversal has been successfully applied in previous works [5, 10] to efficiently answer XML join queries as we explain in Section 6. The second and novel part (Figure 4) of the SA algorithm is the processing and bookkeeping performed at each stack operation (i.e., push and pop) in order to maintain a minimum amount of information that allows the efficient and timely output of the GDMCTs.

The stack S consists of entries of the form (s.nodeID, s.GDMCTs), where s.GDMCTs is a list of GDMCTs found so far, rooted at the node with id s.nodeID. These GDMCTs may be *partial*, i.e., contain a subset of the query keywords, and are annotated with the keywords their nodes contain.

```
GetList($k_1, \ldots, k_m$) {
1.  For $i = 1, \ldots, m$ do;
2.      Get $L(k_i)$ and $L^a(k_i)$ from master index;
3.  return $(\cap_i^2 L^a(k_i)) \bigcup (\cup_i L(k_i))$;
        /* $\cap_i^2 L^a(k_i)$ computes, in a single scan, nodes that appear in at least two distinct lists */ }
POP($S$){
4.          $h \leftarrow$ pop($S$);
5.          Output and remove from $h$.GDMCTs those GDMCTs that contain all keywords;
                /* results whose LCA is $h$ */
6.          $h' \leftarrow$ top($S$);
7.          For each GDMCT $G$ in $h$.GDMCTs do {
                /* migrate the remaining GDMCTs of $h$ to $h'$ */
8.              $d = h$.depth $-h'$.depth;
9.              $r \leftarrow$ root($G$); /* $r$.nodeID = $h$.nodeID */
10.             If $degree(r) = 1$ AND $e$ is the edge of $G$ incident on $r$ then {
                    /* $h$ does not have to be in the GDMCT, and $h'$ can replace $h$ as the new root */
11.                 $label(e) \leftarrow label(e) + d$; /* GDMCT edge label */
12.                 $label(r) \leftarrow h'$.nodeID; /* GDMCT node label */ }
13.             else { /* $h$ still needs to be in the GDMCT, and a new root must be created */
14.                 Create a new root $r'$ for $G$, and $label(r') \leftarrow h'$.nodeID;
15.                 Add edge $e'$ from $r'$ to $r$, and $label(e') \leftarrow d$; }
16.             If $size(G) > K$ then drop $G$; /* pruning condition, size = sum of edge labels */ }
            /* the next two steps combine and merge the GDMCTs of $h$ with those of $h'$ */
17.         $h'$.GDMCTs $\leftarrow h'$.GDMCTs $\cup$ CreateNewGDMCTs( $h$.GDMCTs, $h'$.GDMCTs );
18.         $h'$.GDMCTs $\leftarrow$ Merge( $h$.GDMCTs, $h'$.GDMCTs ); }
PUSH($n, S$){
19.     Push $s(n, \emptyset)$ onto $S$; /* new stack entry $s$ */
20.     For $i = 1, \ldots, m$ do
21.         If $n$ contains $k_i$ then $s$.GDMCTs $\leftarrow s$.GDMCTs $\cup\{n^i\}$;
                /* superscript $i$ identifies the keyword contained in the new single-node, partial GDMCT */ }
CreateNewGDMCTs( $h$.GDMCTs, $h'$.GDMCTs ) {
   /* Generate new GDMCTs that contain multiple keywords. Notice that
      even GDMCTs with all keywords will not be output at this point. */
22.     NewGDMCTs $\leftarrow \emptyset$;
23.     For each $G \in h$.GDMCTs do
24.         For each $G' \in h'$.GDMCTs do
                /* check if $G', G$ are disjoint */
25.             If keywords($G$) $\cap$ keywords($G'$) $= \emptyset$ AND $size(G) + size(G') \leq K$ then {
                    /* glue the disjoint trees on their common root */
26.                 NewGDMCTs $\leftarrow$ NewGDMCTs $\cup$ mergeGDMCTs($G, G'$); }
27.     Return NewGDMCTs; }
Merge( $h$.GDMCTs, $h'$.GDMCTs ) {
   /* combine isomorphic trees */
28.     NewGDMCTs $\leftarrow h$.GDMCTs $\cup h'$.GDMCTs;
29.     For each $G \in h$.GDMCTs do
30.         For each $G' \in h'$.GDMCTs do
31.             If keywords($G$) = keywords($G'$) AND $G, G'$ are isomorphic with mapping $\mu$ from $G$ to $G'$
32.                     AND for every node $u^i \in G, u'^i \in G'$, $\mu(u^i) = u'^i$ then {
                    /* merge node lists with the same keyword match.*/
33.                 Replace $G, G'$ in NewGDMCTs by mergeGDMCTs($G, G', \mu$); }
        /*mergeGDMCTs(.) differs from Merge(.) because (a) it inputs 2 single GDMCTs, and */
        /* (b) it does not check any condition before merging.*/
34.     Return NewGDMCTs; }
```

Figure 4: Operations of Stack Algorithm for All GDMCTs Problem

The algorithm scans the list L consisting of nodes that either contain at least one keyword or are ancestors of at least two nodes that contain the query keywords; these are the only nodes that have the chance of being an LCA or participating in a GDMCT. Nodes of L are being pushed and popped from the stack S as the scanning proceeds. In particular, at the end of each iteration of the main loop (i.e., of the loop of lines 2–6 of Figure 3) the top entry of S contains the node n with the highest `start` value seen so far. The other entries of the stack correspond to the ancestors of n. Before n is pushed onto the stack, all the stack entries that do not correspond to ancestors of n are popped from S. This is accomplished by the loop of lines 4–5 of Figure 3. When an entry h is popped from S, any complete GDMCTs from h.GDMCTs are output (line 5 of Figure 4). The remaining GDMCTs are partial. Since there is a possibility that the parent of h may have descendants that have the keywords that the partial GDMCTs miss, the partial GDMCTs of h become partial (or complete) GDMCTs of its parent h'. Notice that the entry h' may already have partial GDMCTs that reflect the keywords found in descendants of h' that were inspected before h. The transfer of each partial GDMCT G of h to the set of GDMCTs of h' follows the following steps:

- modify G to reflect the new root (lines 10–15) of Figure 4,
- check to see if G satisfies the pruning condition (line 16 of Figure 4)

Once we have the modified and pruned set of partial GDMCTs of h we compare them against the GDMCTs of its parent h' and create new GDMCTs as is appropriate (line 17 of Figure 4), which we merge with the GDMCTs of h'. In particular, we create a new GDMCT for each pair of GDMCTs from h and h' that can be "glued" together to contain a larger subset of the keywords (lines 23–26 of Figure 4). Finally, we merge (line 18 of Figure 4) into the same GDMCT every pair of GDMCTs from h and h' that are isomorphic, to ensure the minimality of the number of produced GDMCTs.

Notice that the reason that the result GDMCTs rooted at node h are output when h is popped from the stack (line 5 of Figure 4) and not when they are initially produced (lines 17, 22–27 of Figure 4) is because there could be more GDMCTs that are "mergeable" with the GDMCTs already produced (lines 18, 28–33 of Figure 4).

3.2.2 Illustrative Example

We illustrate the execution of our Stack Algorithm, using an example, with two query keywords "Tom, Harry", and a threshold of 3. The master index lists L and L^a are shown above for these query keywords. In line (3), the intersection of L^a(Tom) and L^a(Harry) would produce the list $[(r, 1, 42, 0), (c_1, 2, 41, 1), (s_1, 3, 16, 2), (p_1, 4, 9, 3), (s_2, 17, 26, 2), (p_3, 18, 25, 3), (s_3, 27, 40, 2)]$. Notice that the entries $(p_2, 10, 15, 3), (p_4, 28, 31, 3)$ and $(p_5, 32, 35, 3)$ are not present in this list, since they are ancestors of only one of the query keywords, and hence can neither be an LCA, nor be part of any GDMCT.

The Stack Algorithm then iteratively chooses entries from (the conceptual union of) L(Tom), L(Harry), and this intersection. Some of the initial stack states in the execution are depicted below.

1. $$\begin{array}{|l|}(p_1, \emptyset)\\(s_1, \emptyset)\\(c_1, \emptyset)\\(r, \emptyset)\\ \hline\end{array}$$
   The first four entries in the intersection of the L^as are pushed on S.

2. $$\begin{array}{|l|}(a_1, \{a_1^2\})\\(p_1, \emptyset)\\(s_1, \emptyset)\\(c_1, \emptyset)\\(r, \emptyset)\\ \hline\end{array}$$
   The first entry a_1 from L(Harry) is pushed on S, and a partial GDMCT is created; the superscript of 2 in the GDMCT of a_1 indicates a match for the second query keyword "Harry".

3. $$\begin{array}{|l|}(p_1, \{p_1 \xrightarrow{1} a_1^2\})\\(s_1, \emptyset)\\(c_1, \emptyset)\\(r, \emptyset)\\ \hline\end{array}$$
   When examining the first entry a_2 from L(Tom), the top of stack a_1 is popped, and a new GDMCT is created at p_1.

4. $$\begin{array}{|l|}(a_2, \{a_2^1\})\\(p_1, \{p_1 \xrightarrow{1} a_1^2\})\\(s_1, \emptyset)\\(c_1, \emptyset)\\(r, \emptyset)\\ \hline\end{array}$$
   The first entry a_2 from L(Tom) is pushed on S, and a partial GDMCT is created; the superscript of 1 in the GDMCT of a_2 indicates a match for the first query keyword "Tom".

5. $$\begin{array}{|l|}(p_1, \{p_1 \xrightarrow{1} a_1^2, a_2^1 \xleftarrow{1} p_1 \xrightarrow{1} a_1^2, p_1 \xrightarrow{1} a_2^1\}) \\ (s_1, \emptyset) \\ (c_1, \emptyset) \\ (r, \emptyset) \\ \hline \end{array}$$
When examining the second entry a_3 from L(Tom), the top of stack a_2 is popped, and new (combined) GDMCTs are created at p_1. Note that a solution has been found, but it is not output yet.

6. $$\begin{array}{|l|}(s_1, \{s_1 \xrightarrow{2} a_1^2, s_1 \xrightarrow{2} a_2^1\}) \\ (c_1, \emptyset) \\ (r, \emptyset) \\ \hline \end{array}$$
When examining the second entry a_3 from L(Tom), the top of stack p_1 is also popped, and the answer $(p_1, a_2^1 \xleftarrow{1} p_1 \xrightarrow{1} a_1^2)$ is output. Additional GDMCTs are also associated with the (new) top of stack s_1.

7. $$\begin{array}{|l|}(a_3, \{a_3^1\}) \\ (s_1, \{s_1 \xrightarrow{2} a_1^2, s_1 \xrightarrow{2} a_2^1\}) \\ (c_1, \emptyset) \\ (r, \emptyset) \\ \hline \end{array}$$
The entry a_3 from L(Tom) is then pushed on the stack, and a partial GDMCT is created.

8. $$\begin{array}{|l|}(s_1, \{s_1 \xrightarrow{2} a_1^2, s_1 \xrightarrow{2} [a_2^1, a_3^1]\}) \\ (c_1, \emptyset) \\ (r, \emptyset) \\ \hline \end{array}$$
When examining the next entry s_2 from the intersection of L^a(Tom) and L^a(Harry), the top of stack a_3 is popped, new GDMCTs are created, and merged with the GDMCTs associated with s_1. In particular, the GDMCT $s_1 \xrightarrow{2} a_3^1$ is created (since a_3 is at distance 2 from s_1), and merged with $s_1 \xrightarrow{2} a_2^1$, resulting in $s_1 \xrightarrow{2} [a_2^1, a_3^1]$. The GDMCT $a_3^1 \xleftarrow{2} s_1 \xrightarrow{2} a_1^2$ is not created, since its size (of 4) exceeds the user-defined threshold of 3.

9. Entries from the lists continue being examined, new GDMCTs created, and pruned until all the answers are output.

3.3 Lowest GDMCTs: Stack-Based Algorithm

We now present a simple modification of the Stack Algorithm of Figures 3, 4 to efficiently answer Problem 2 (the Lowest GDMCTs Problem). This is the case when the user is interested only in the lowest GDMCTs, i.e., those GDMCTs whose roots are not ancestors of other returned GDMCT roots. The key observation is that once we output the GDMCTs of a node u (in line 5 of Figure 4), none of the ancestors of u in the stack can be LCAs of returned GDMCTs; hence, we can remove all of them from the stack! Specifically, we can add the following lines after line 5 of the Stack Algorithm in Figure 4.

```
5a.   If a GDMCT was output then {
5b.       $S \leftarrow \emptyset$;
5c.       return; }
```

As an example, consider again the query keywords “Tom, Harry”, but with a threshold of 5. Once the first solution $(p_1, a_2^1 \xleftarrow{1} p_1 \xrightarrow{1} a_1^2)$ is output in Step 6 (in the illustrative example of Section 3.2.2), the stack is emptied. Thus, no GDMCT with an LCA of c_1 or s_1 would be returned. (Note that, in the All GDMCTs Problem for this example, the solution $(s_1, a_3^1 \xleftarrow{2} s_1 \xrightarrow{2} a_1^2)$ would also be returned.) We refer to this algorithm as `SALowAll`.

3.4 LCAs: Stack-Based Algorithms

The Stack Algorithm can also be easily modified to solve the All LCAs Problem and the Lowest LCAs Problem, where the user is not interested in the GDMCTs, but only in the LCA nodes. Essentially, the algorithms, which modify `SA` and `SALowAll` and we refer to as `SAOne` and `SALowOne`, respectively, would still need to maintain GDMCTs with stack nodes, with two simplifications:

- Procedure Merge(.) in Figure 4 could be simplified, no merging of GDMCTs would need to be done, and line 33 could be replaced by:

```
33.        Remove one of $G, G'$ in NewGDMCTs;
```

- It is possible to output an LCA early when the first GDMCT (with all keywords) is computed for that node (in Procedure CreateNewGDMCTs(.) in Figure 4), instead of waiting until the node is popped from the stack.

An important point to note is that, while tempting, it does not suffice to simply (i) maintain, with each stack node u, the distance d_i to the closest descendant u_i of u found so far containing keyword k_i, and (ii) produce an output when each distance has been filled in, and the sum of the distances is $\leq K$. This is because, except for the special case of two query keywords, the size of a GDMCT is not simply the sum of the distances from the LCA to each of the nodes containing the m keywords.

3.5 Complexity Analysis

This section presents (Section 3.5.3) a worst-case complexity analysis for SA. Before doing so, we perform an analysis of the maximum number of the resulting GDMCTs (Section 3.5.1) and we discuss how individual operations of SA can be performed in linear time on the size of the GDMCTs (Section 3.5.2).

3.5.1 Total Number of GDMCTs

We show that in the worst case the numbers of DMCTs and of GDMCTs are exponential on the number of keywords. However, under reasonable assumptions explained below, the worst case number of GDMCTs is smaller than that of DMCTs. Also notice that in practice the number of GDMCTs is typically much smaller than the number of DMCTs, due to the grouping.

Consider a query with m keywords $k_1, k_2, \ldots, k_m$. Let $L(k_i)$ be the list of the nodes of tree T that contain keyword k_i. A DMCT can be obtained by combining one node from each of the m lists $L(k_i), 1 \leq i \leq m$. Thus, in the worst case, the total number of DMCTs is given by $\Pi_{i=1}^{m}|L(k_i)|$, which is exponential in m. GDMCTs group isomorphic DMCTs to provide a more compact result. But what is the worst case total number of GDMCTs? We show that this can also be exponential in m.

In particular, consider a node n that has each of the m keywords k_i in its subtree, and each keyword k_i occurs at h different depths $d = 1, \ldots, h$ in the subtree rooted at n. It is easy to see that there has to be a different GDMCT for each combination of (keyword, depth). In this case, there are $\Pi_{i=1}^{m} h = h^m$ GDMCTs, which is exponential in m.

However, under reasonable assumptions, the number of GDMCTs is asymptotically smaller that that of DMCTs. Consider the simple case where GDMCTs have no internal nodes, no node contains more than one keyword, and the XML tree has height H. Then the maximum possible number of DMCTs is $\Pi_{i=1}^{m}|L(k_i)|$ as above, but the maximum number of GDMCTs is H^m (each of the m keywords can be in depth $1,\ldots,H$). Hence, if H is viewed as a constant, the number of GDMCTs is asymptotically smaller than that of the DMCTs.

3.5.2 Complexity of Finding Isomorphic GDMCTs

Deciding when two GDMCTs can be merged in SA is expensive, unless we refine the representation of GDMCTs. In this section we describe a canonical representation of a GDMCT that allows (a) a rapid determination of whether GDMCTs can be glued together in CreateNewGDMCTs (lines 23-25 of Figure 4), and (b) checking whether two GDMCTs are isomorphic, permitting them to be merged (lines 31-33 of Figure 4). In this canonical representation:

- Each node in the GDMCT is annotated with the keywords in its subtree, in lexicographic ordering, and the size of its subtree.
- The children sub-trees (rooted at nodes $n_1,\ldots n_j$) of node n are ordered according to lexicographic ordering of the annotations of the roots of these children subtrees.

Given this canonical representation, one can linearize the GDMCTs in an XML-like nested representation with start and end tags, obtained from the node annotations. Given this linearized representation:

- Checking whether two GDMCTs can be glued together requires checking if their keyword sets are disjoint, and if their combined size does not exceed K, which can be checked using their annotations in the canonical representations; this can be done in a single pass of the GDMCTs, that is, in linear time on the size of the GDMCTs.
- Checking whether two GDMCTs are isomorphic can be done by equating the canonical representations; this can be done in linear time on the size of the GDMCTs as well.

3.5.3 Time Complexity of SA

In the SA algorithm, each node in L (which is computed in GetList) is pushed on to the stack, and popped from the stack, at most once. When a node is popped from the stack, its GDMCTs need to be compared (and possibly merged) with the GDMCTs of its parent node in the stack. Since each operation on a pair of GDMCTs can be done in linear time on the size of the GDMCTs, the total time complexity of SA is a function of the total number of GDMCT comparisons, which is quadratic in the total number of GDMCTs. As a result, in the worst-case, we have:

Theorem 1 *The time complexity of SA is* $O(|L| + K \cdot (\Pi_{i=1}^{m}|L(k_i)|)^2))$.

4 Processing Unindexed XML Data

In this section, we consider the case when no master index is available on the XML data tree, and the goal is to efficiently solve the All GDMCTs Problem for a specific keyword query (with a threshold). Both the Nested Loops Algorithm and the Stack Algorithm have straightforward adaptations to work without index lists, by doing a single pass over the data tree. In particular, `NLStream`, which is the streaming version of `NL`, first traverses in one pass the data tree to create the index lists of the query keywords and then executes the NL algorithm [2]. The streaming version of the Stack Algorithm, which we refer to as `SAStream`, is realized by making the following changes to the Stack Algorithm of Figures 3, 4. Notice that `NLStream` makes an additional pass over the data tree, unlike `SAStream` which just makes a single pass.

```
   Remove line 1 of Figure 3.
   Replace lines 2–3 of Figure 3 with:
2. While $S$ not empty OR more nodes in $T$ do
3.     $n \leftarrow$ Get next node in depth-first order from $T$;
   /*Note that $d$ in line 8 of of Figure 4 will always be 1.*/
```

[2] The main drawback of this approach is that the indexing and the execution stages are separated, which means that the entire inverted index entries would have to be stored and then processed. This factor becomes more important when the index entries are too long to fit in memory, and are moved to and from secondary storage during the indexing and processing stages.

Frequency	*XMark 10MB*	*XMark 100MB*	*DBLP*
1-10	41176	161048	401181
11-200	10924	15883	19213
$>$200	1294	9210	956

Table 1: Number of keywords in each frequency range in the data sets used

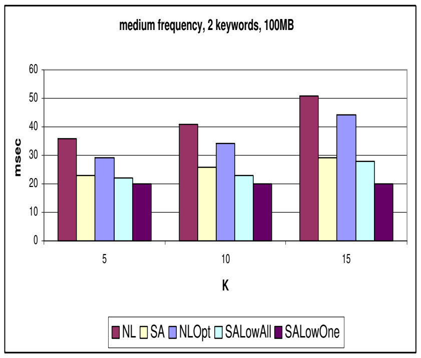
5 Experimental Evaluation

We have designed and performed a comprehensive set of experiments to understand the performance of the proposed algorithms. We used both real and synthetic data sets. The synthetic datasets were generated using the XMark benchmark [2] for various database sizes. We also used the DBLP database [1] to explore the performance of our algorithms using more realistic data distributions. The experiments were conducted on a Xeon 2.2GHz computer with 1GB of RAM running Windows 2000 Professional. The algorithms were implemented in Java and the parsing of the XML files is performed using the SAX API of the Xerces Java Parser [3]. The master index is implemented as a Java Hashtable persistent object.

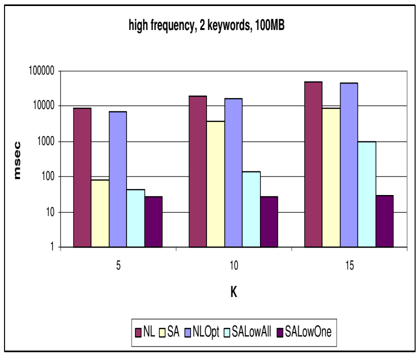
There are three main parameters affecting the performance of our algorithms, namely (i) the value of K denoting the threshold, (ii) the number m of keywords, and (iii) the size of the dataset. To understand better the performance of our algorithms for keywords of different selectivities, we perform experiments using sets of keywords having different frequencies, namely *low*, corresponding to keywords with frequency between 1 and 10 in each data collection, *medium*, corresponding to keywords with frequency 11-200, and *high*, corresponding to keywords with frequency above 200. The number of keywords in each frequency range, in the different data sets used, is shown in Table 1.

The experiments are divided into three classes. First we evaluate the proposed algorithm `SA`, and its variants `SALowAll`, `SALowOne`. As a baseline for comparison, we use the algorithm `NL`, which computes LCAs and GDMCTs using a nested loops approach. We also evaluate an improvement of this basic strategy that uses the optimal algorithm for identifying the LCA of a pair of keywords [15]. This algorithm, `NLOpt`, still considers all pairs of keywords in a nested loops fashion, but it identifies the LCA of a pair very efficiently, namely in $O(1)$ time. Next, in Section 5.2, we evaluate our algorithms for the case when no indices are available on the XML data.

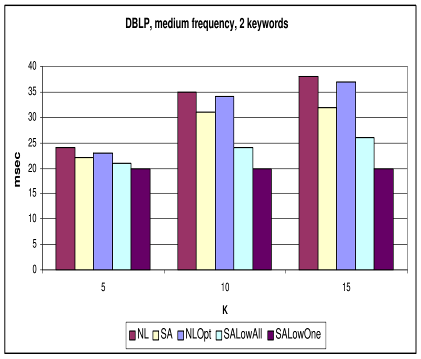
[3]http://xml.apache.org/xerces-j/



(a) XMark 100MB, medium frequency



(b) XMark 100MB, high frequency



(c) DBLP, medium frequency

Figure 5: Varying K

Dataset (MBs)	SA *index (MBs)*	NL *index (MBs)*
1	5.1	3.7
10	49	37
100	500	377

Table 2: Index Size Requirements of SA

Each value reported in our graphs is an average collected from 50 repetitions of the experiment. Finally, we compare the SA algorithm against algorithms for keyword proximity search on labeled graphs [16, 17, 4]. However, since the algorithms of the prior work operate on data stored in relational database systems we also built a version of the SA for XML data stored in a relational database, so that the comparison is straightforward.

5.1 Evaluating SA and its Variants

Our first experiment evaluates the index size requirements of the proposed SA algorithm, for different sizes of XML data collections of the XMark benchmark. First we compare the size of the index required by the Stack Algorithm (SA) compared to the Nested Loops Algorithm (NL) for various XMark dataset sizes. We allocate 4 bytes for each node identifier and each `start`, `end` value in the depth-first numbering, and 1 byte for the `depth` number. Since the `start` value serves as a unique node identifier as well, we take this into account in our space computation for the SA index. Table 2 presents the index size of SA compared with that of NL, for various database sizes generated using the generation tools available in the XMark benchmark. Considering the entries of the table, it is evident that the index size requirements of SA are about 33% higher than that of NL. As we will soon demonstrate, SA introduces this small space overhead in order to provide orders of magnitude performance improvements.

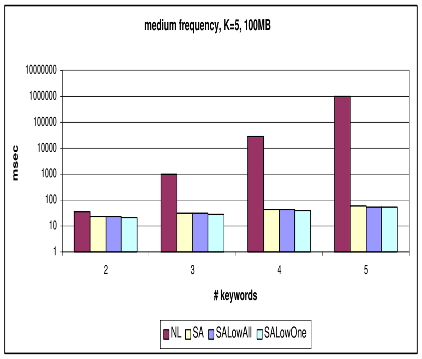
Figure 5 presents the performance of the algorithms as K (the distance threshold) increases for a fixed number of keywords (equal to two), for the XMark 100MB and the DBLP datasets. In the rest of the section, due to space considerations, we do not present the graphs for low frequency keywords since we have found that they take constant time (up to 20 msec which is the disk access time) to execute. For the same reason, we only present results for (the most common in practice) medium frequency keywords for DBLP, because we use the larger XMark dataset to show how the time scales for frequent keywords (we have found that DBLP scales following the same patterns).

It is evident that `SA` is considerably superior to both `NL` and `NLOpt`. `SA`'s performance benefits are pronounced when high frequency keywords are involved, since the number of nodes from the underlying XML tree involved in the operation increases considerably. `NL` incurs high overhead because it considers all possible pairs of nodes containing the query keywords and groups the results in GDMCTs. `NLOpt` also considers all pairs, although each pair requires much less time to process (compared to `NL`) and thus its performance is somewhat improved. Disk access appears to be the dominating factor in Figure 5(a) and Figure 5(c) (because relatively smaller lists of nodes are involved due to medium frequency query keywords), whereas processing time is the dominating performance factor in Figure 5(b). Table 3 presents the average number of GDMCTs for the various keyword frequencies in the 100MB XMark dataset, for different threshold values. It is evident that the number of GDMCTs produced in the case of high frequency keywords is much higher, contributing considerably to the increased overhead of `NL` and `NLOpt`, in addition to their inherent overhead of considering all node pairs. The trend for all algorithms is to experience a degradation in their performance as K increases, for a specific data size and keyword frequency, because the expected size of the stack nodes involved in the operation increases. Notice that for algorithms `SALowAll` and `SALowOne` this degradation in performance is not significant, even compared to algorithm `SA`, since the output produced by these algorithms is much smaller. In particular it is interesting to observe that for the Algorithm `SALowOne`, which produces the least output, its performance appears almost insensitive to the range of K values tested. In contrast, it only depends on the specific dataset and subsequently on the corresponding query keyword frequency.

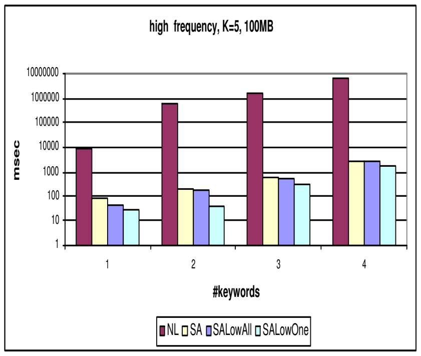
Figure 6 presents the results of an experiment exploring the performance impact of an increasing number of keywords, for a fixed threshold $K = 5$. Notice that for clarity of display, `NLOpt`

K	*medium frequency*	*high frequency*
5	0.38	33.9
10	28	186
15	37	385

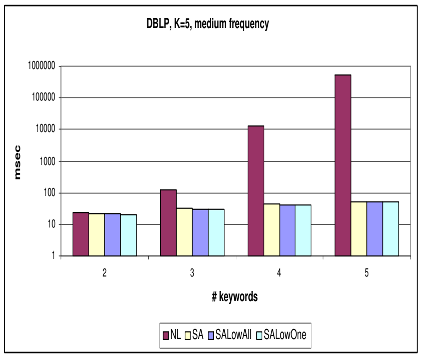
Table 3: Average number of GDMCTs for the 100MB XMark dataset, for medium and high frequency keywords



(a) XMark 100MB, medium frequency



(b) XMark 100MB, high frequency



(c) DBLP, medium frequency

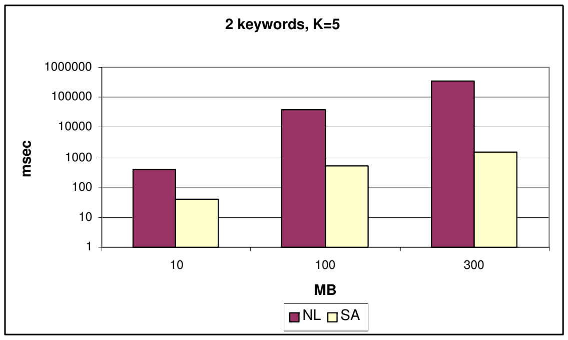
Figure 6: Varying number of keywords m

is not plotted since its performance is very close to NL. Since NL considers all combinations of keywords, one from each keyword list, its performance deteriorates exponentially to the number of keyword lists. Algorithm SA and its variants are capable of scaling gracefully to an increasing number of keywords, since they perform a single pass over the keyword lists and their performance benefits are substantial.

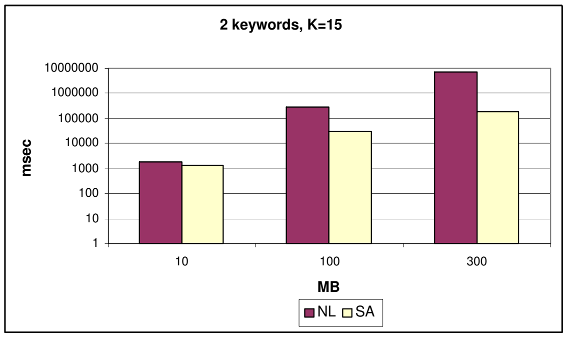
Figure 7 presents the performance of the algorithms for increasing database size, for various values of the distance threshold K; notice the log scale on the Y axis. To isolate the effects of increasing data size, we present the results for keywords selected uniformly at random among the 1000 keywords with the highest frequency in each data set respectively. The results, which are shown in Figure 7, indicate that the proposed algorithms scale gracefully with increasing database size, exhibiting almost linear increase in performance with database size. The scalability limitations of algorithm NL are evident in the figure. Increasing the database size is expected to increase in effect the absolute frequencies of the most 1000 frequent keywords, which is the keyword collection from which our queries are derived. As a result, by increasing the database size, the keyword lists provided as input to each algorithm respectively are much larger in size. Table 4 presents some statistics of the distribution of frequencies of the 1000 most frequent keywords, as the size

Size	*max freq*	*min freq*	*avg. freq*
10MB	6663	218	296
100MB	66247	2176	2945
300MB	1999274	6557	8820

Table 4: Statistics on the frequency of 1000 most frequent keywords for increasing database size, for XMark data



(a) XMark, K=5



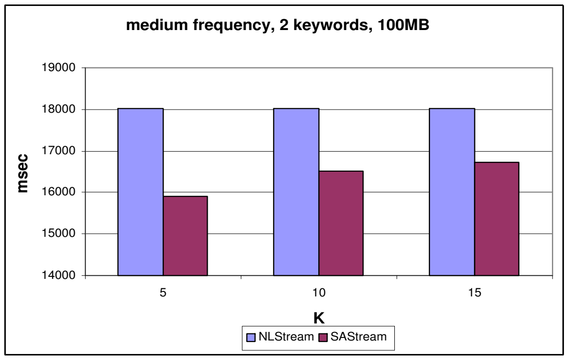
(b) XMark, K=15

Figure 7: Varying database size

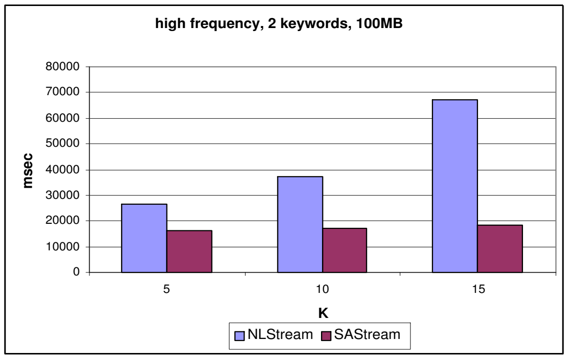
of the datasets increases. It is evident that the top 1000 keyword frequencies increase substantially with increasing database size.

5.2 Evaluating the `SAStream` Algorithm

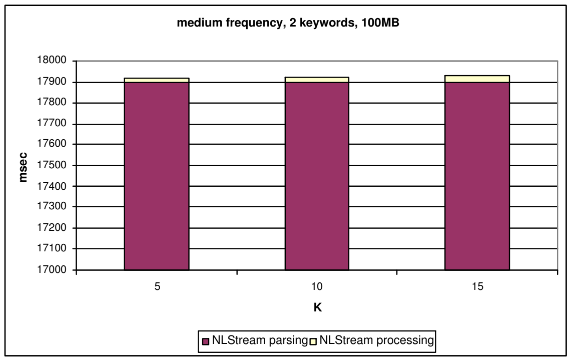
We now present the evaluation of the variants of our algorithms for non-indexed data, where the execution times increase dramatically due to the lack of indexing that leads to reading the whole XML file. Figure 8 compares algorithms `NLStream` and `SAStream` for increasing values of the distance threshold K, for two keywords, for medium and high frequency keywords. Notice that `NLStream` initially parses the XML document, constructing indices, and then operates on those indices. In contrast Algorithm `SAStream` can operate immediately in conjunction with document parsing. In Figure 8(a), since we are dealing with not so frequent keywords, `NLStream`'s performance is dominated by the time to read the document and create the keyword lists and thus its performance appears to increase only marginally with increasing values of K. Figure 8(c) presents a breakdown of the times spent at the two stages of `NLStream`'s execution. In effect, `SAStream` produces the desired result faster than the time required by `NLStream` to identify the relevant keywords and build indices. The performance advantages of `SAStream` are pronounced as the



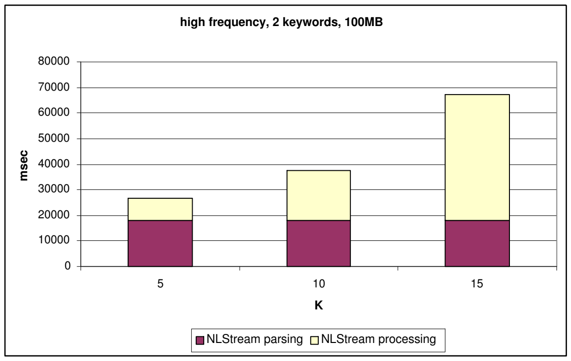
(a) medium frequency



(b) high frequency



(c) `NLStream` Components (medium freq)

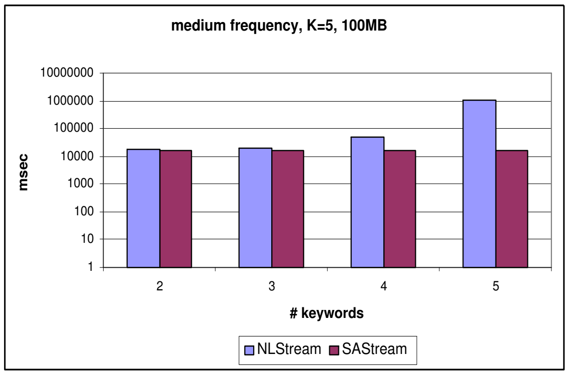


(d) `NLStream` Components (high freq)

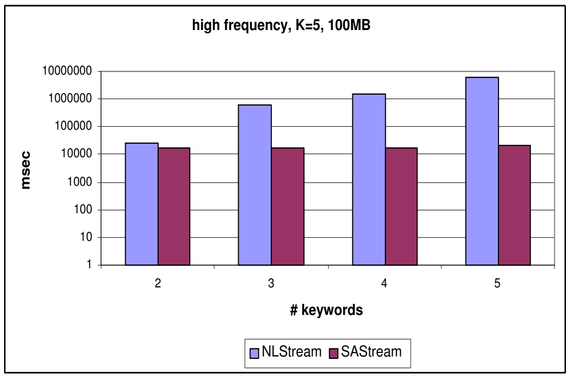
Figure 8: Varying K in the algorithms for non-indexed, 100 MB XMark data

frequency of the keywords involved in the operation increases, since its performance is linear in the size of the document. Contrasting Figure 8(b) and Figure 8(d), we observe that the time required by `NLStream` to produce the output increases, since larger lists of nodes are involved in the operation. The performance advantages `SAStream` offers in this case are substantial.

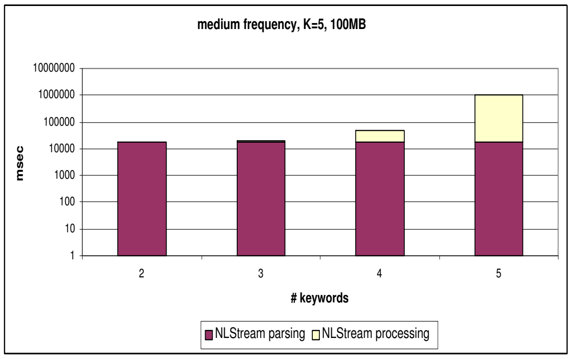
In Figures 9(a) and (b), we present the performance of `SAStream` and `NLStream` as the number of keywords increases, for a fixed distance threshold $K = 5$. In Figures 9(c) and (d), we present a breakdown of the times taken by algorithm `NLStream` at the various stages of its execution. `NLStream`'s execution time increases exponentially with m, in contrast to `SAStream`, whose times remain relatively stable, since document parsing and identification of relevant answers are interleaved. As observed in Figures 9(c) and (d), parsing time is the dominating factor in the performance of `NLStream` with processing time becoming significant as the number of keywords increases.



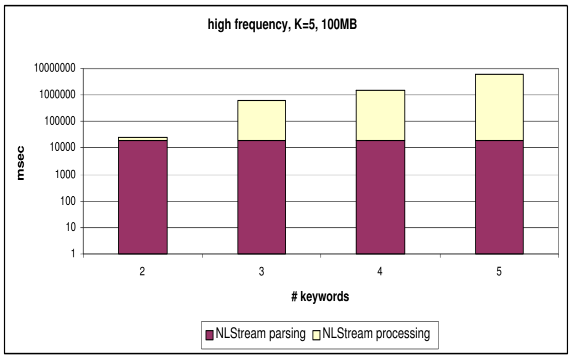
(a) medium frequency keywords



(b) high frequency keywords

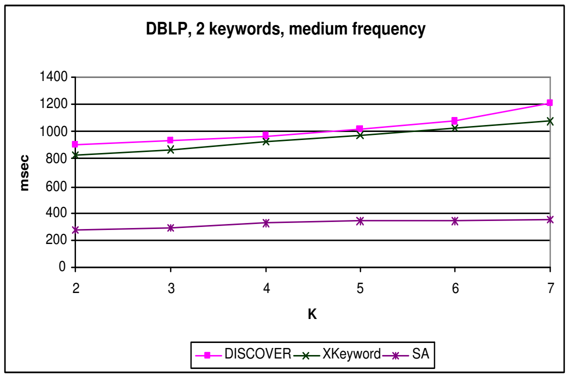


(c) `NLStream` Components (medium freq)

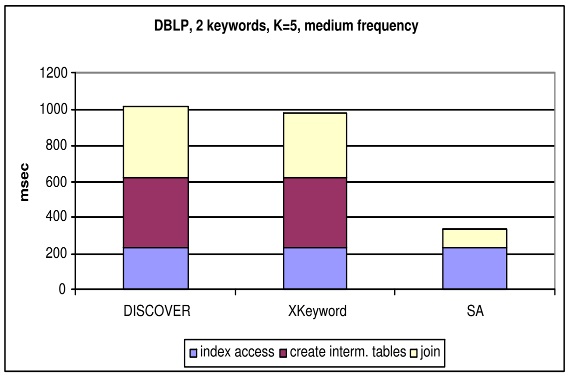


(d) `NLStream` Components (high freq)

Figure 9: Varying the number of keywords in the streaming algorithms, 100MB XMark Data



(a) DBLP, 2 keywords



(b) DBLP, 2 keywords, $K = 5$

Figure 10: Compare to DBMS based approaches

5.3 Adaptation of SA algorithm for DBMS

Next we compare SA against three systems that perform keyword proximity search on labeled graphs: DBXplorer [4], DISCOVER [16] and XKeyword [17] (see Section 6 for a short description of these works). Since all of them operate on data stored in a relational database, in order to have a fair comparison we implemented a version of SA which operates on data stored in a DBMS.

In particular, the exact same indexing method is used as in XKeyword and DISCOVER. That is, Oracle Intermedia Text Index [4] is used to find the nodes that contain the keywords. The nodes of the tree along with their (start,end,depth) triplet are stored in a relation, which we refer to as *Master* relation, whose text attributes are indexed by Oracle Intermedia. The runtime of the algorithm consists of two stages: reading the text index to get the nodes/tuples that contain the keywords and their ancestors, and executing the SA algorithm on these nodes. Given the nodes that contain the keywords, their ancestors are computed using the (start, end) information on which a B+ index has been built. The index reading stage to find the nodes with the keywords is identical to the one used in XKeyword and DISCOVER. However, these works continue by building a set of intermediate tables (tuple sets) and finally executing a set of join queries to produce the results. On the other hand, SA does not need to access the database any more to compute the results. Figure 10 compares the performance of these algorithms for the DBLP dataset. Figure 10 (b) analyzes the cost of each algorithm into the costs of the consisting stages. Notice that we do not include DBXplorer in the graphs, since it is slower than DISCOVER due to the lack of common subexpressions reuse.

Finally, notice that the performance of SA decreases considerably when building the master index as described above, since two steps are needed to get the keyword lists: first query the DBMS text index to get the node ids, and second get the corresponding (start,end,depth) triplets from the Master relation. On the other hand, these triplets are retrieved in a single step using the file-based master index described in Section 3.2.1.

6 Related Work

Lowest Common Ancestor The first area of research relevant to this work is the computation of the LCA of a set of nodes of a data tree. Kersten et al. [19] present an algorithm, which

[4]http://technet.oracle.com/products/text/content.html.

for two keywords, is the same as the Nested Loops algorithm (NL) we present. For more than two keywords, their semantics are different from the traditional proximity search semantics [13]. In particular, their algorithm inputs a set of relations (i.e., sets of nodes of different types) that contain the keywords and outputs all pairwise LCAs and not global LCAs. Notice that the nodes are grouped by type and not by keywords, so there could be pairwise LCAs that only contain the same keyword twice. Also notice that they use a schema, in contrast to our work.

Li et al. [18] and XKSearch [20] defined Smallest LCAs (SLCAs) to be LCAs that do not contain other LCAs. Li et al. [18] incorporated SLCA search in XQuery. The algorithms of XKSearch benefit from the observation that, in contrast to the general LCA problem, the number of smallest LCAs is bounded by the size of the smallest keyword list. Consequently, in [20] the keyword lists of the inverted index are themselves indexed and indexed lookup is used to find potential matches in the large keyword lists. The algorithm has a generalization to finding all LCAs but then its key observation does not apply and, more important, it has no efficient way to produce summaries (such as the GDMCTs) of why each result node is an LCA. The algorithm in [20] cannot be straightforwardly modified to support the general LCA problem.

XRANK [14] and XSEarch [12] return subtrees as answers to the keyword queries. However, the algorithm of XRANK does not return MCTs to explain how the keywords connect to each other. Furthermore, only the most specific result are output. They also present a ranking method which, given a tree t containing the keywords, assigns a score to t using an adaptation of PageRank [9] for XML databases. Their ranking techniques are orthogonal to the retrieval and hence can easily be incorporated in our work. XSEarch focuses on the semantics and the ranking of the results, and during execution, they use an all-pairs index to check the connectivity between the nodes.

Efficiently computing the *lowest common ancestor* (LCA) of a pair of nodes in a tree is a problem that has received a lot of attention in the theoretical community and efficient approaches in main memory are known for its solution [15, 6]. In particular, given a tree, after suitable preprocessing it is possible to construct data structures, to answer to LCA queries (given a pairs of nodes report the node which is the LCA of the pair in the tree) in $O(1)$ time. The construction is relatively involved (the interested reader could consult [15]) and efficient, provided that the data structures fit in memory. We adapt suitably modified algorithms proposed for main memory LCA of a pair of nodes, making them suitable for the problems we consider herein (algorithm `NLOpt`), and use them as a basis for comparison with our solutions.

Proximity search on labeled graphs Proximity search on labeled graphs [13, 7, 16, 17, 4] has been suggested as an effective information discovery method. In most works the labeled graph is derived by connecting the tuples of a relational database by primary key/foreign key links. [4, 16, 7] are particularly built for relational databases: SQL queries are used to derive the result. More recent works [17, 7] use XML data as the motivation for labeled graphs; the edges correspond to element/subelement connections or IDREF links.

The algorithms for keyword proximity search in labeled graphs are intrinsically expensive, heuristics-based and typically use various forms of precomputation in order to improve the performance. They do not significantly exploit the special case where the data structure is a tree.

Goldman et al. [13] retrieve and rank objects according to their proximity from other objects of interest in a labeled graph. They show how to speed up the computation of the pairwise distances between any two nodes of the graph by precomputing a hub structure. The choice of hubs is guided by heuristics. However, when calculating the distance between two sets S_1, S_2 of nodes, all combinations of nodes from S_1, S_2 are tested for results, leading to a quadratic (cubic for three keywords etc) cost similar to the Nested Loops algorithm (NL) of Figure 2. They propose a way to avoid this quadratic number of disk accesses by clustering objects of the same type (eg. movies or actors), which is a solution that can work for keywords appearing as tag names in an XML document but is not realistic for arbitrary keywords. And still their algorithm suffers from a quadratic (or more) number of comparisons.

The BANKS system [7] finds MCTs in a labeled graph by using an approximation to the Steiner tree problem, which is NP-hard. The key idea (we omit optimization details) is the following: BANKS progressively calculates the neighbor sets N_i of distance up to K of every node u_i that contains a keyword and outputs a spanning tree T when the root of T is found in the intersection of the N_i's. This leads, similarly to Goldman et al. [13], to a quadratic (for two keywords) number of comparisons, in contrast to our one pass algorithms. Their implementation is tuned for a graph that fits in main memory.

DISCOVER [16], XKeyword [17] and DBXplorer [4] are systems working on top of relational databases, facilitating keyword search for relational [16, 4] and XML databases [17]. DISCOVER and DBXplorer output trees of tuples connected through primary-to-foreign key relationships, that contain all the keywords of the query. They first get from the master index the tuples that contain the keywords and then generate a set of SQL queries corresponding to all different ways to connect the keywords based on the schema graph. XKeyword extends the work of DISCOVER by materializing path indices in a relational database, to reduce the number of joins in the generated SQL queries. These works rely on a schema, in contrast to this work. More importantly, since the data structure is a graph, it is impractical to store all the connections between all pairs of nodes in the inverted index of the keywords. Hence, they may need to read from the disk an unbounded number of connecting tuples, to discover the connections between the keyword nodes. In contrast, in our work, we index the nodes that contain the keywords along with their “coordinates” in the source tree, which leads to a single disk access per keyword in the typical case (when the set of nodes that contain each keyword fits in a disk page). In Section 5.3, we compare these works to an adaptation of our approach for a DBMS. This adaptation removes our advantage of tightly integrating the keyword index with the representation of the “coordinates” of the nodes. However, we show that we still perform considerably better than these works.

Finally, stack-based algorithms for processing XML queries have been proposed recently in the literature computing containment joins [5] as well as holistic joins [10]. Our algorithms differ from these algorithms in that we incrementally maintain and output LCAs and GDMCTs, which are considerably more complex than checking ancestor-descendant relationships.

7 Conclusions and Future Work

In this paper, we have investigated the problem of XML keyword queries, with the aim of identifying the most specific context elements (i.e., LCAs) that contain all the keywords, along with a compact description of their witnesses (i.e., GDMCTs). We have proposed and evaluated efficient algorithms for a number of variants of this problem, and have established that the context of XML keyword queries can indeed be efficiently determined as part of query evaluation.

Our work opens the door to a number of different avenues of research in XML keyword queries. What would Information Retrieval style approximate matching look like? Our stack-based algorithms maintain partial GDMCTs during query evaluation; are these the desired answers to approximate keyword queries? What is the analog of *tf∗idf* for ranking the results of XML keyword queries? What are appropriate linguistic mechanisms to incorporate our keyword querying primitives into XQuery? We are currently exploring some of these promising directions of research.

References

[1] DBLP computer science bibliography. *http://dblp.uni-trier.de/*.

[2] The XML benchmark project. *Available from http://www.xml-benchmark.org*.

[3] S. Abiteboul, P. Buneman, and D. Suciu. Data on the Web: From Relations to Semistructured Data and XML. *Morgan Kaufmann*, 1999

[4] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. *Proceedings of ICDE*, 2002.

[5] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. *Proceedings of ICDE*, 2002.

[6] M. Bender and M. F. Colton. The LCA problem revisited. *Latin American Theoretical Informatics*, 2000.

[7] G. Bhalotia, C. Nakhey, A. Hulgeri, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases Using BANKS. *Proceedings of ICDE*, 2002.

[8] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. *W3C Working Draft. Available from http://www.w3.org/TR/xquery*.

[9] S. Brin, and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Proceedings of WWW7*, 1998.

[10] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. *Proceedings of SIGMOD*, 2002.

[11] J. Clark and S. DeRose. XML path language XPath 1.0. *W3C Recommendation. Available from http://www.w3.org/TR/xpath*.

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

[13] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. *Proceedings of VLDB*, 1998.

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

[15] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing, 13 (2)*, pages 338–355, 1984.

[16] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. *Proceedings of VLDB*, 2002.

[17] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. *Proceedings of ICDE*, 2003.

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

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

[20] Y. Xu, and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. *Proceedings of SIGMOD*, 2005.

[21] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. *Proceedings of SIGMOD*, 2001.