Generalized Labeled LCA Queries

Jérémy Barbay, Ehsan Chiniforooshan, Alexander Golynski,
Jui-Yi Kao, Aleh Veraskouski

David R. Cheriton School of Computer Science,
University of Waterloo, Canada
Technical Report CS-2006-31,
revised in July 2007.

Abstract

Schema-free queries permit to search XML documents without knowing their schema. Among them, lowest common ancestor (LCA) queries were introduced in several variants on labeled trees. We define threshold LCA queries to generalize all those variants, and to extend them to the case where weights are assigned to each term of the query. We study how to solve those in the context where access to the document is streamed, and in the context where the document is accessed through a precomputed index. We propose space-efficient algorithms for both contexts, using space $\mathcal{O}(h+\sigma)$ independant from the size of the document, where h is the height of the input document and σ is the number of different labels. We describe two distinct algorithms in the streamed model, both of which read the input stream exactly once and run in linear time, and one algorithm in the indexed model, which provably runs in sublinear time for many instances, characterized by a difficulty measure.

Keywords: Streamed XML, Schema-free queries, LCA, SLCA, Threshold Set.

1 Introduction

XML [12] standardizes document tree structures, so that general tools can be developed and used for the many distinct applications adopting this standard. Each document is mainly composed of text, which forms its *content*, and tags, which form its *hierarchical structure*. It is assumed that documents of different content but similar semantics will share a similar hierarchical structure, expressed as a *schema* which specifies which tags can be used, and constrains the structures that they can form. Knowing the schema allows to write relevant queries on the structure of the document, while its content can be searched using traditional techniques. In many cases the schema is not known or is voluntarily ignored, for instance because documents following many different schemes must be searched [1]: then the documents are searched using *Schema-Free* [5, 8] queries. We consider *streamed* documents, on which one query must be answered in one single pass over the document and in linear time; and *indexed* documents, for which an index has been computed in advance in order to allow faster computations.

Schema-Free [5, 8] queries, which consist only of a set of keywords, are studied for applications where the queries must be written without any knowledge of the schema of the document, for instance because many documents with many different schemes must be searched [1], or simply because the schema is not communicated (as often in the streamed model). In particular, Schmidt,

Kersten and Windhouwer [8] defined *Lowest Common Ancestor* (LCA) queries, which are answered with the list of nodes which descendants match at least two keywords. Li, Yu and Jagadish [5], observing that the answer to those queries is not always relevant, proposed to exclude the nodes whose descendants also answer the query. Xu and Papakonstantinou [11] went further in restricting the answer set of the queries, by requesting that the descendants match *all the keywords*, while still excluding the nodes with a descendant also answering the query.

We introduce in Section 2 the concept of *Threshold LCA* queries, which generalizes the previous definitions of LCA queries into a single one through an additional parameter t and optional weights associated to the labels of the query. In Section 3 we propose two distinct one-pass algorithms to search *Streamed Documents* in linear time: one slightly more general (allowing negative weights) and the other returning more information (size of corresponding interval in the document). In Section 4 we describe an algorithm to search *Indexed Documents* in sublinear time for many instances, proved through an adaptive analysis in function of a measure of the difficulty of the instances. Throughout the paper we assume that the tree-structured document is accessed as a *multi-labeled tree* such that each of the n nodes is associated with some of the labels from an alphabet of size σ. The algorithms proposed are *space-efficient*: their space usage depends on the height h of the document and on the size of the alphabet σ, but is independant of the size of the document, assuming that a word of the machine can index a position in the document.

2 Previous Work and Definitions

While XML documents have a great variety of features, only a few are of interest in the context of schema-free queries. For instance, the distinction between elements and attributes, based on the semantic meaning of order, is ludicrous in the context of schema-free queries, where this order is not used at all. As a consequence we focus only on a few features of XML. The original definition of XML documents [12] is text-based, where each node of the structure of the document is associated with a string. We propose a view based on a relation between the structure of the document and a finite set of labels. More precisely, in this work, an *XML tree* is composed of a set $[\sigma] = \{1, \dots \sigma]$ of labels, a tree of n nodes such that each leaf is the only child of its parent, and a relation composed of t pairs between nodes and labels such that each internal node is assigned a label and each leaf is assigned a set of labels. The distinction is secondary, as any dictionary structure (such as a trie) permits to translate back and forth between labels and strings, but it makes the definition of the queries and algorithms easier. More formally, we say that a node x *matches* k labels $\alpha_1, \dots, \alpha_k$ if there is a k-tuple of nodes $(x_1, \dots, x_k)$ matching $(\alpha_1, \dots, \alpha_k)$ in the subtree rooted in x.

2.1 Ordinal LCA

Considering only the structure of the tree, the *ordinal Lowest Common Ancestor* (LCA) of k nodes $x_1, \dots, x_k$ is the lowest common node $\texttt{LCA}(x_1, \dots, x_k)$ in common between the paths from each node x_i to the root. LCA queries on labeled trees have been studied by the database community in the context of schema-free queries on XML documents. The techniques used consist in choosing the encoding of the tree so that the LCA can be computed quickly. One technique often used is based on Dewey numbers, which represent the path to each node, reduce the LCA Problem to finding the common prefix of two paths, which takes time linear in the length of the smallest path.

LCA queries on ordinal trees were first studied by Harel and Tarjan [4]., and later on by Schieber and Vishkin [7], Wen [10], and Bender *et al.* [2, 3]. They defined the *ordinal Lowest Common*

Ancestor (LCA) of k nodes $x_1, \ldots, x_k$ as the lowest common node $\texttt{LCA}(x_1, \ldots, x_k)$ between the paths from each node x_i to the root.

It is a fundamental algorithmic problem on trees and has been extensively studied [3, 4, 7, 10].

The technique allowing to compute it quickly consists in choosing an adequate encoding of the tree: for instance Dewey numbers, which represent the path to each node, reduce the LCA Problem to finding the common prefix of two paths. Using some more sophisticated off-line precomputation, the LCA of two nodes can be computed in constant time [3, 10].

The technique introduced by Harel and Tarjan [4] and later refined consists in precomputing the answer to some queries (an index), and to use those precomputed answers to answer the future queries in constant time. One of the latest results about LCA is due to Bender, Farach-Colton, Pemmasani, Skiena, and Sumazin [3], who precompute in linear time a structure of $O(n)$ words (i.e. $O(n \lg n)$ bits) using dynamic programing, so that they support in constant time LCA on a tree of n nodes. Sadakane [6] improves this to precompute the largest common prefix length of suffixes in $6n + o(n)$ bits and implicitly defines a tree encoding using $2n + o(n)$ to support LCA in constant time.

2.2 Previous Work

Schmidt, Kersten and Windhouwer [8] observe that requiring the user to know the schema is impractical and unrealistic, and that content queries are insufficient. They suggest instead unstructured queries interpreted in regard to the structure of the document, and define the *Lowest Common Ancestor set* of k labels $\alpha_1, \ldots, \alpha_k$ as the set $\texttt{LLCA}(\alpha_1, \ldots, \alpha_k)$ such that the subtrees rooted at each node of this set partition all nodes matching at least two labels without intersecting each other.

Li, Yu and Jagadish [5] observe that the answer to LCA queries is sometime meaningless, because the level of relevance of the nodes matching some labels vary too much. They propose to return only the most relevant nodes matching the query, through the *Meaningful Lowest Common Ancestor* set of k labels $\alpha_1, \ldots, \alpha_k$, defined as the set $\texttt{MLCA}(\alpha_1, \ldots, \alpha_k)$ of nodes matching at least two labels which corresponding subtree does not contain any other node matching at least two labels.

The distinction with the previous queries is that the nodes returned by a `LLCA` query cover all nodes matching the labels, while the `MLCA` query rejects nodes which match the labels but are judged too general because some of their descendants are already matching those labels.

Xu and Papakonstantinou [11] go further in restricting the answer set of the queries, by requesting that the nodes match *all labels*. As Li *et al.*, they don't consider all such nodes but reduce the answer to the most meaningful nodes, forming the *Smallest Lowest Common Ancestor* set of k labels $\alpha_1, \ldots, \alpha_k$: the set $\texttt{SLCA}(\alpha_1, \ldots, \alpha_k)$ of nodes such that the subtrees rooted at each node of this set partition all nodes matching $\alpha_1, \ldots, \alpha_k$ without intersecting each other.

They also extended their algorithm to compute the list of nodes matching k distinct labels, hence introducing a new query.

Definition 1 (`ASLCA` queries [11]) *The set of* All Lowest Common Ancestors *corresponding to k labels $\alpha_1, \ldots, \alpha_k$ is the set* $\texttt{ASLCA}(\alpha_1, \ldots, \alpha_k)$ *of nodes matching $\alpha_1, \ldots, \alpha_k$.*

2.3 Threshold Labeled LCA

We propose a fourth type of query, generalizing both `MLCA` and `SLCA` query-types. Whereas `MLCA` queries require two labels to be matched by each node of the answer, and `SLCA` queries require all

of the labels to be matched, we *parametrize* the amount of labels that a node of the answer should match, and consider *weights* associated to each label to measure the relevance of each label to the answer.

Definition 2 *Consider a tree T labeled by a binary relation $R : [n] \times [\sigma] \to \{0,1\}$, a query $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$, a node x of the tree, and a positive number t.*

- *The* score *of x is the sum of the weights of the labels associated to x or at least to one of its descendants:*
$$\texttt{score}(x) = \sum_{\alpha \in [\sigma]} Q[\alpha] \max_{y \text{ descendant of } x} R(y, \alpha)$$
- *the answer to a* Threshold Labeled LCA query *(*`TLLCA`*) is the set of nodes x such that x's score is at least t and no descendant of x matches the previous condition.*

By definition, when the weights are all equal to zero or one, the answer to such a query corresponds to the answer of a `MLCA` query for $t = 2$, and to the answer of a `SLCA` query for $t = k$. This extension of the query-type to its weighted variant can be used to automatically personalize user queries: given a set of labels input by the user, assign them a normal weight and add to them several labels of small weights defining the profile of the user.

3 Streamed Documents

In the context of streamed documents, instead of accessing the input document T directly, we are allowed to read $\pi(T)$ only in one pass, where $\pi(T)$ denotes the preorder sequence of the nodes of T with parentheses around all the nodes (see Figure 1 for an example). We seek for sub-linear space algorithms. In the following two sections, we present two linear time algorithms for the weighted query problem.

3.1 Recursive Algorithm

Let $Q = (\alpha_1, \alpha_2, \ldots, \alpha_k)$ be the weighted query with weights $W = (w_1, w_2, \ldots, w_k)$. A naive algorithm can be stated as follows. We crawl the tree recursively starting from the root. For each node, we check whether it can be an answer to the weighted query, namely, if the following two conditions are satisfied:

- none of its descendants is an answer, and
- the weight of the union of the labels of all its descendants is above the given threshold.

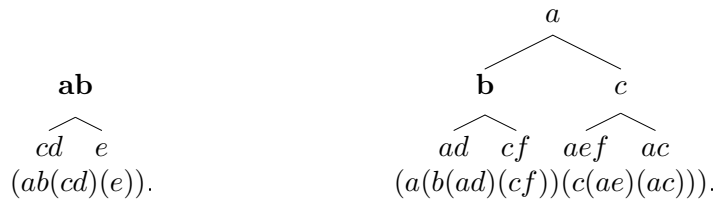

Figure 1: Examples of streamed trees. Nodes in $LCA(a, b, c)$ are denoted in bold.

Once these conditions are checked for a node v, we return the union of the labels of all the descendants of v to its parent u (u performed a recursive call to v). However, the

Algorithm 1 Weighted_query(v)
returns: $L \cap Q$, where L is the set of all labels occurring in the v's subtree and a Boolean value found_v that indicates whether v or any of its descendants is an answer to the weighted query

```
1:  S ← ∅
2:  found_v ← false
3:  for all u is a child of v do
4:     (S^t, found_u) ← Weighted_query(u)
5:     S ← S ∪ S^t { S holds the set of labels up to and including the current child u in the loop }
6:     found_v ← found_v ∨ found_u
7:  end for
8:  S ← S ∪ (labels(v) ∩ Q)
9:  if (¬found_u) ∧ (weight(S) ≥ w) then
10:    output(v)
11: end if
12: return (S, found_v)
```

straightforward implementation of Algorithm 1 is not very efficient, since we need to maintain a set of labels for each level of recursion. The goal is to give a more efficient implementation. We start by modifying the algorithm slightly so that any two sets of labels corresponding to different recursion levels will have empty intersection. We achieve this by modifying line 8 so that whenever we add a label to the set of the current level, we also remove it from all the ancestor's sets. Such an operation can be implemented efficiently by a union-find data structure [9]. A union-find data structure maintains a family of sets of elements subject to the following two operations: union(u, v) creates a new set that is union of the sets u and v (the side effect of this operation is that the sets u and v are destroyed), and find(v) finds the set that contains element v. We allow some sloppiness of the notation by denoting an element and a set by letters u and v. In fact, the implementation by Tarjan [9] picks an elements from each set, and this elements becomes the representative of this set. In our implementation, the levels of recursion will play the role of the representatives of the sets, and the labels in the query will play the role of the elements in the sets, see Algorithm 2. This algorithm is a streamed and improved version of Algorithm 1. Each level of recursion stores four local variables: found, p, $r[d]$ (weight of the current node), and $S[d]$ (the pointer to the corresponding set in the union-find data structure). So that the total space is O(depth of the tree). Lines 22-28 process the labels of the current node one by one: for label a, we first find the index i in the query so that $a = \alpha_i$ then find the level where the label α_i is stored, and perform the path compression in the union-find data structure as necessary [9]; line 25 sets α_i's pointer to the current set $S[d]$. Thus, the modified algorithm stores each label only at the deepest possible level to which the label belongs. To implement the check weight$(S) \geq w$ on the line 9 of Algorithm 1, we store a weight $r[d]$ for all the recursion levels d and update it as necessary. We have an invariant that when the node at the level d is being processed, $r[d]$ equals to weight(S), the total weight of the union of the labels of all the descendants of v. These weights can be updated as follows: in line 11 we can sum the two corresponding weights since all the sets in the union-find data structure are disjoint; line 26 subtracts the weight from the corresponding set where the label a is stored, and line 27 adds it

Algorithm 2 Procedure Weighted_query()
returns true, if the current node in the stream or one of its descendants match the query

```
1: d ← d + 1 { d is a global variable indicating the level of recursion}
2: p ← current position in the stream { p is local variable indicating the first position of the current
   node }
3: r[d] ← 0 {Create a new root with weight 0}
4: found ← false
5: while true do
6:    read a { Read the next element in the input stream }
7:    if a ='(' then
8:       {Child of the current node, recurse}
9:       found ← found ∨ Weighted_query() {The recursive call to Weighted_query created a new
         root at the level d + 1}
10:      Union (S[d], S[d + 1]) {Union the two roots: the newly created one and the current}
11:      r[d] ← r[d] + r[d + 1] {Correspondingly update the weight of the current root}
12:   end if
13:   if a =')' then
14:      {Finished processing the current node, exit the recursion}
15:      if (¬found) ∧ (r[d] ≥ w) then
16:         q ← current position in the stream { the last position of the current node }
17:         output (p, q) {None of the descendants is an answer and sufficient weight}
18:         found ← true
19:      end if
20:      d ← d − 1
21:      return found
22:   end if
23:   if a ∈ {α_1, ..., α_k} then
24:      i ← index such that a = α_i
25:      z ← Find(i) { The label a is stored in the set at the level z }
26:      set a's pointer to S[d] { The union-find implementation in [9] is pointer based }
27:      r[z] ← r[z] − w_i
28:      r[d] ← r[d] + w_i { Move the label a down to the current level and update the root weights
         correspondingly}
29:   end if
30: end while
```

to the weight of the current set. Initially, all k query labels point to a dummy root at the level 0 with the weight $\sum_i w_i$, and the level variable is set to 0.

Although the running time for the $n-1$ union operations and m find operations is $O((n+m)\alpha(m))$ [9], where α denotes the inverse Ackerman function; we can show that in our specific case these operations only take $O(n+m)$ time. We first simplify the original union-find data structure, so that when it performs union of two roots, it always link the deepest level of the tree to the second deepest level (this is the only type of unions we perform in the algorithm), the find operation performs the standard path compression as usual. Clearly, the total cost of the union operations is $O(m)$. Note that every edge in the union-find data structure, except for the edges that are adjacent to a label, corresponds to an edge in the original tree. For each find operation, we charge the cost of compressing all the edges except the first and the last one on the path to the corresponding edges in the tree. Note that no edge in the tree gets charged twice. So that the total number of edge compressions is at most $n-1$ plus $2m$. And hence we have $O(n+m)$ run-time.

3.2 Sequential Algorithm

The idea of the algorithm is as follows: while the algorithm reads the input stream, $\pi(T)$, it computes the minimal intervals of the stream that contain enough keywords. For each interval, the algorithm computes the closest pair of parentheses that contain the interval; this shows the LCA of the nodes of the interval. Finally, the algorithm filters out the LCA's that have another LCA as a descendant. The algorithm is shown in Algorithm 3.

Algorithm 3 uses an auxiliary data-structure to find the minimal intervals with enough keywords as described in the following Lemma.

Lemma 1 *There exists a linear time data-structure that, for any set of keywords $\alpha_1, \ldots, \alpha_k$ and any threshold weight w, supports the following operations:*

1. *insert(a, d): a is a keyword in $\alpha_1, \ldots, \alpha_k$ and d is some data associated with a.*
2. *last: If (a_1, d_1), ..., (a_k, d_k) is the sequence of inserted pairs and the weight of keywords that appear at least once in $a_1, \ldots, a_k$ is at least w, then this operation returns d_h such that*
   (a) *the weight of keywords that appear at least once in $a_h, a_{h+1}, \ldots, a_k$ is at least w, and*
   (b) *for any $h' > h$ the weight of keywords that appear at least once in $a_{h'}, a_{h'+1}, \ldots, a_k$ is less than w.*

Moreover, this data-structure uses $O(k)$ space.

Proof: The proof is simple: we use a doubly linked list L, together with an integer w_{current} representing the weight of the keywords that are in L. Initially L is empty. For each keyword α_i, We will keep at most one pair (a_j, d_j) in L such that $a_j = \alpha_i$. Also, we keep a pointer for each keyword α_i to the pair (a_j, d_j) in L such that $a_j = \alpha_i$. If there is no such pair in L, the pointer is NULL. We denote this pointer by $p[\alpha_i]$.

Whenever a pair (a_i, d_i) is inserted into L, we can check if L already has a pair with label a_i by checking whether $p[a_i]$ is NULL or not. If $p[a_i]$ is NULL, we add (a_i, d_i) at the head of L, update $p[a_i]$ to point to the location of (a_i, d_i) in L, and increase w by the weight of a_i. Otherwise, we remove the pair that $p[a_i]$ points to from L, insert (a_i, d_i) at the head of L, and update $p[a_i]$ to point to the location of (a_i, d_i) in L.

After each insertion, we also do the following: while $w_{\text{current}} - w$ is at least the weight of a_ℓ, where (a_ℓ, d_ℓ) is the element at the end of L, we remove (a_ℓ, d_ℓ) and decrease w_{current} by the weight of a_ℓ.

Whenever operation "last" is called, we simply return d_ℓ, where (a_ℓ, d_ℓ) is the pair at the end of L.

It is simple to check that the running time is linear in terms of the number of insertions and the data-structure takes $O(k)$ space. □

Algorithm 3 Sequential Algorithm

```
 1: STACK: a stack.
 2: MY LLIST: the doubly linked list described in Lemma 1.
 3: loc ← 0.
 4: last-output ← 0.
 5: while the input stream is not finished do
 6:    read a.
 7:    loc ← loc +1.
 8:    if a ='(' then
 9:       push loc into STACK.
10:    else if a ∈ {α_1, ..., α_k} then
11:       insert (a, loc) into MY LLIST.
12:    else if a =')' then
13:       open ← pop one element from STACK.
14:       if the weight of MY LLIST is at least w then
15:          tail ← the preorder number of the last element of MY LLIST.
16:          if (open < tail) and (open > last-output) then
17:             last-output ← loc.
18:             output (open, loc).
19:          end if
20:       end if
21:    end if
22: end while
```

In the following theorem, we prove that Algorithm 3 works correctly.

Theorem 1 *For an input stream $\pi(T)$, any set of keywords $\alpha_1, \ldots, \alpha_k$, and any integer w, Algorithm 3 returns a sequence of pairs $(\ell_1, r_1), (\ell_2, r_2) \ldots, (\ell_k, r_k)$ such that*

1. *ℓ_i and r_i are locations of an open parenthesis and its corresponding closed parenthesis in $\pi(T)$,*
2. *the total weight of keywords that appear at least once in $\pi(T)$ between ℓ_i and r_i is at least w,*
3. *there is no set of parentheses (ℓ', r') (that correspond to each other) inside (ℓ_i, r_i) such that the total weight of keywords that appear at least once between ℓ' and r' is at least w, and*
4. *there is no pair with the above three properties that is not returned by the algorithm.*

Note that the pairs that are reported to the output correspond to the nodes of T that are in the answer of the input `TLLCA` *query.*

Proof: The proof has two parts: we first prove that any pair (ℓ, r) that has properties 1–3 will be returned, and, second, we prove that if a pair (ℓ, r) is returned, it has properties 1–3.

1. Assume that (ℓ, r) is a pair with properties 1–3. Then, when Algorithm 3 reads the rth element of the input stream ($\texttt{loc} = r$), which is ')' due to property 1, it goes to lines 13–20. So, `open` will be set to ℓ. Because of property 2, the condition of line 14 holds and, again due to property 2, `tail` will be set to a number greater than ℓ. Therefore, `tail` will be greater than `open`. Also, because of property 3, `last-output` will be less than ℓ and thus `open`. So, (`open`, `loc`), which is equal to (ℓ, r), will be reported to the output.

2. Assume that (ℓ, r) is returned. Then, similarly to the above case, it is simple to verify that all properties 1–3 hold for (ℓ, r).

□

Note that, by a slight modification, Algorithm 3 can output more information like the length of the interval that contains enough keywords. This information cannot be easily extracted from the recursive algorithm.

4 Indexed Documents

In contrast to the context of streamed documents, in the context of indexed documents an algorithm can access the input data in an arbitrary order.

Our algorithm works in a similar way as the sequential one, but takes advantage of the index. It scans through the nodes in *postorder* to find minimal intervals whose labels contribute at least t to the score of the corresponding subtree. We call the set of roots of those subtrees the *relaxed answer* to the query. Only those nodes that do not have any descendants in the input pass the filtering of the relaxed answer, which follows.

Algorithm 4 starts from the first node in postorder. For each node x under consideration, it finds the smallest node x_{right} larger than x, such that all the labels associated with the nodes $(x, \ldots, x_{right})$ contribute at least t to the score of their common ancestors. Then the algorithm finds the largest node x_{left} with the same property but starting from x_{right} and progressing backward: this ensures that $(x_{left}, \ldots, x_{right})$ is the *minimal t-interval* that starts from x_{left} and contributes enough weight. The algorithm outputs $y = \texttt{lca}(x_{left}, x_{right})$ and proceeds further, by setting the current node x to the smallest node larger than x_{left} that matches at least one label α of positive weight $Q(\alpha) > 0$. The algorithm terminates when it cannot find the next node x_{right}.

The auxiliary function weight(x_{left}, x_{right}) used in Algorithm 4 shows what weight all the nodes in the interval $(x_{left}, \ldots, x_{right})$ contribute to the the score of the root node $y = \texttt{lca}(x_{left}, x_{right})$.

$$\text{weight}(x_{left}, x_{right}) = \sum_{\alpha \in [\sigma]} Q(\alpha) \texttt{I}(\exists x \in (x_{left}, \ldots, x_{right}), \text{s.t. } \alpha \text{ is associated to } x),$$

where $I()$ is the indicator function.

To find node x_{right}, the algorithm computes the node $x_\alpha = \texttt{label\_successor}(\alpha, x)$ for each label $\alpha, Q(\alpha) > 0$, which is the smallest node larger than x associated with the label α. It builds a min-priority queue from all such nodes and retrieves the smallest set of the nodes, such that the sum $\sum Q(\alpha)$ of all associated with them labels is at least t. The algorithm sets x_{right} to the last node retrieved from the priority queue.

Algorithm 4 Main Algorithm for Answering TLLCA Queries (T, Q, t)

$x = x_1$;
loop
  $x_{right} \leftarrow$ the smallest node larger than x, s.t. weight$(x, x_{right}) \geq t$;
  if $x_{right} = +\infty$ then EXIT;
  $x_{left} \leftarrow$ the largest node smaller than x_{right}, s.t. weight$(x_{left}, x_{right}) \geq t$;
  output $\texttt{lca}(x_{left}, x_{right})$;
  $x \leftarrow$ the smallest node larger than x_{left} associated with any label α, s.t. $Q(\alpha) > 0$;
end loop

Algorithm 4 finds x_{left} in a similar way, except that it finds the set from largest nodes $x_\alpha = \texttt{label\_predecessor}(\alpha, x_{right})$ smaller than x and associated with a specific label α; and builds a max-priority queue.

Our on-line filtering algorithm is inspired by the one provided implicitly by Xu and Papakonstantinou [11]: it receives nodes from the main algorithm one by one and leave only those that do not have descendants among inputted ones. The algorithm stores the most recent received node as $\texttt{x}_\texttt{r}$ and check the relationship between $\texttt{x}_\texttt{r}$ and the new node x coming to the input. If x is an ancestor of $\texttt{x}_\texttt{r}$, x is discarded, because ancestors are not in the answer. If x is a descendant of $\texttt{x}_\texttt{r}$, $\texttt{x}_\texttt{r}$ is discarded and replaced by x. Otherwise, our filtering algorithm sends $\texttt{x}_\texttt{r}$ to the output (i.e. confirms that it is in the answer) and sets $\texttt{x}_\texttt{r}$ to the newcomer node x. When there is no more nodes coming to the input, the algorithm outputs $\texttt{x}_\texttt{r}$ to the answer and finishes.

Algorithm 5 Filtering Algorithm for Answering TLLCA Queries

$\texttt{x}_\texttt{r} \leftarrow$ the first node coming to the input;
for each x coming to the input **do**
  $\texttt{x}_\texttt{lca} \leftarrow \texttt{lca}(\texttt{x}_\texttt{r}, x)$;
  if $\texttt{x}_\texttt{lca} = \texttt{x}_\texttt{r}$ then discard x;
  elseif $\texttt{x}_\texttt{lca} = x$ then $\texttt{x}_\texttt{r} \leftarrow x$;
  else output $\texttt{x}_\texttt{r}$; $\texttt{x}_\texttt{r} \leftarrow x$;
end for
output $\texttt{x}_\texttt{r}$;

Each node in the answer corresponds to the minimal t-interval (x_{left}, x_{right}) that has weight of at least t and which was used by the algorithm to find this node; the inverse is not always true. As the number of such intervals of minimal size increases the number of nodes in the answer increases as well. We take the number of minimal t-intervals as our adaptive *measure of difficulty*.

Definition 3 *Consider a labeled tree T, a query Q, and a positive number t.* The difficulty δ *of the problem instance is the number of distinct minimal t-intervals.*

In the following lemmas we prove dependencies between nodes in the answer and relaxed answer, minimal t-intervals, and iterations of Algorithm 4. We represent them graphically in Figure 2.

Lemma 2 *Consider a labeled tree T, a query Q, and a positive number t. There is a bijection f_1 between the set of all minimal t-intervals and each main loop iteration of Algorithm 4.*

Proof: Consider a function f_1 from the set of all minimal t-intervals to the set of all iterations of Algorithm 4 that processes these intervals. We prove that f_1 is injective and surjective.

Function f_1 is injective because for each two different minimal t-intervals I_1 and I_2 the corresponding iterations $f_1(I_1)$ and $f_1(I_2)$ of Algorithm 4 are different: the intervals start from different nodes (otherwise they cannot be different, because they are minimal), and each iteration of the algorithm starts from one of the successors of the node the previous iteration started.

Moreover, as any interval the algorithm comes up with is a minimal t-interval, because it will have weight less than t without the first or the last node, f_1 is surjective. □

Lemma 3 *Consider a labeled tree T, a query Q, and a positive number t. There is an injection f_2 from the answer to the set of all minimal t-intervals.*

Proof: Consider a function f_2 that for each node y in the answer gives the leftmost minimal t-interval $(x_{left}, \ldots, x_{right})$, s.t. $\texttt{lca}(x_{left}, x_{right}) = y$. We proof this lemma by showing that for each pair of different nodes x_1, x_2 in the answer, their images $f_2(x_1)$, $f_2(x_2)$ are different.

As nodes x_1 and x_2 cannot have ancestor-descendant dependency, because an ancestor is not in the answer in this case, they have totally disjoint subtrees. While the minimal t-interval that gives the required weight to the node is inside the subtree rooted by this node, the intervals $f_2(x_1)$, $f_2(x_2)$ cannot be the same. The injectivity of f_2 follows. □

Lemma 4 *Consider a labeled tree T, a query Q, and a positive number t. Assuming that Algorithm 4 outputs all nodes that are in the answer and some or no nodes that are not in the answer but are in the relaxed answer, Filtering Algorithm 5 outputs all and only nodes from the answer.*

Proof: We show that all the nodes discarded by Filtering Algorithm 5 are not in the answer and all the nodes it outputs are in the answer.

The filtering algorithm discards nodes only in two cases: first, it discards $\texttt{x}_\texttt{r}$ when it finds that the next coming node is its descendant, which means that node $\texttt{x}_\texttt{r}$ cannot be in the answer. Second, it discards the coming node x when x is an ancestor of node $\texttt{x}_\texttt{r}$ and cannot be in the answer.

All the time the filtering algorithm is working, it ensures node $\texttt{x}_\texttt{r}$ to be not an ancestor of any of the nodes in the input (here we use the order the Algorithm 4 outputs the nodes to the filtering algorithm). The filtering algorithm outputs $\texttt{x}_\texttt{r}$ if the coming node belongs to the next "branch" of the tree and cannot ban $\texttt{x}_\texttt{r}$ from being in the answer as well as all the subsequent nodes that will follow it. □

Theorem 2 *Consider a labeled tree T, a query Q, and a positive number t. There is an algorithm that answers TLLCA query performing $\mathcal{O}(k\delta)$ priority queue, label-successor, label-predecessor operations and $\mathcal{O}(\delta)$ lca operations.*

Proof: The correctness of the algorithm follows from Lemma 2, Lemma 3, and Lemma 4: the minimal t-interval of any node from the answer will be presented in the set of all minimal t-intervals (Lemma 3), will be found by Algorithm 4 and passed to the input of the Filtering Algorithm 5 (Lemma 2) that will output it successfully to the answer (Lemma 4).

Each iteration of Algorithm 4 costs $\mathcal{O}(k)$ label-successor and priority queue operations for finding x_{right}, $\mathcal{O}(k)$ label-predecessor and priority queue operations for finding x_{left}, and one $\texttt{lca}(x_{left}, x_{right})$ operation. Filtering Algorithm 5 performs one more $\texttt{lca}(x_{left}, x_{right})$ operation for every node from the input. As the number of iterations of Algorithm 4 is δ the total worst-case

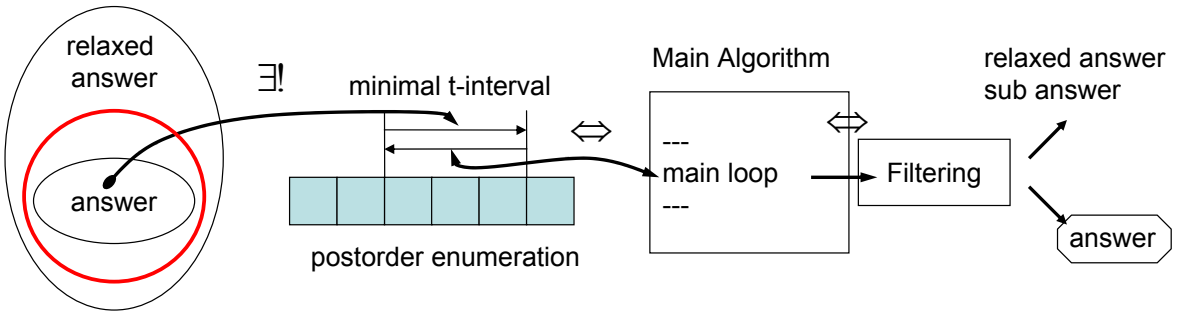

Figure 2: The set of all nodes that correspond to all minimal t-intervals contains answer entirely but is only a subset of the relaxed answer. The algorithm finds the corresponding node for each minimal t-interval and determines whether it is in the answer or not via filtering.

complexity is bounded by $\mathcal{O}(k\delta)$ of priority queue, label-successor, label-predecessor operations and $\mathcal{O}(\delta)$ lca operations. □

The high-level operations `label_successor`(α, x) and `label_predecessor`(α, x_{right}) can be supported in logarithmic time by posting lists of the nodes associated with each label. The heap implementation of a priority queue has logarithmic time complexity. The operator `lca`(x_{left}, x_{right}) can be easily supported in constant time [3].

5 Discussion of Results

We consider a family of schema-free queries based on the search for lowest ancestor descendants (LCA queries). Through the addition of a parameter and of a system of weights on the terms of the query, we generalize two existing query-types by threshold labeled LCA queries. We provide two distinct one-pass space-efficient algorithms to search *Streamed Documents* in linear time. One is slightly more general, and allows negative weights in the query, while the other can return more information, such as the size of the interval corresponding to the labels matched in the document. We also provide a space-efficient algorithm to search *Indexed Documents*. We prove that its complexity is sublinear for many instances, through an adaptive analysis in function of a measure of the difficulty of the instances.

Beside generalizing existing query-types, threshold labeled LCA queries have applications on their own: the addition of weights to the terms of the query can be used to automatically personalize user queries, so that given a set of labels input by the user with a normal weight, several labels of small weights defining the profile of the user are automatically added to the query.

Lastly, weights can also be assigned to the labels associated to the nodes of the tree, in order to indicate the degree of relevance of each distinct label of a given node (e.g. the size of files in the multi-labeled tree representing the file system, or the number of occurrence of a label in a paragraph). The extension of the algorithms presented here to this model is more problematic, and it is an open question to know if such weighted threshold LCA queries on weighted multi-labeled trees can be answered efficiently, in the streamed or indexed model.

Acknowledgments: The authors would like to thank Tamer Oszu for his encouragements and suggestions, Naomi Nishimura, Prabakhar Ragde and Ian Munro for their support, and the

algorithm and complexity discussion group of the University of Waterloo were the whole project started. For any question, contact Ehsan Chiniforooshan as the corresponding author at `echinifo@uwaterloo.ca`.

References

[1] Sihem Amer-Yahia, SungRan Cho, and Divesh Srivastava. Tree pattern relaxation. In *Extending Database Technology*, pages 496–513, 2002.

[2] Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.

[3] Michael A. Bender, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Finding least common ancestors in directed acyclic graphs. In *Symposium on Discrete Algorithms*, pages 845–854, 2001.

[4] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[5] Yunyao Li, Cong Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, 2004.

[6] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete algorithms*, pages 225–232, 2002.

[7] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.

[8] Albrecht Schmidt, Martin L. Kersten, and Menzo Windhouwer. Querying XML documents made easy: Nearest concept queries. In *ICDE*, pages 321–329, 2001.

[9] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[10] Zhaofang Wen. New algorithms for the lca problem and the binary tree reconstruction problem. *Inf. Process. Lett.*, 51(1):11–16, 1994.

[11] Yu Xu and Yannis Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 527–538, New York, NY, USA, 2005. ACM Press.

[12] François Yergeau, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (third edition). Technical report, W3C Recommendation, February 2004.