Compact labeling scheme for XML ancestor queries*

Haim Kaplan † Tova Milo † Ronen Shabo †

Abstract

XML documents are often viewed as trees (basically the parse tree of the document), and queries over such documents typically test for ancestor relationships among tree nodes. Search engines process such queries using an index structure summarizing the ancestor relations. In the index, each document item (tree node), is identified using some logical id (node label), such that, given two labels, the engine can determine the ancestor relationship between the corresponding nodes. The length of the labels is a main factor of the index size. Therefore, reducing this length, even by a constant factor, is a critical issue.

In this work we will consider the following problem. Given a rooted XML tree T, label the nodes of T in the most compact way such that given the labels of two nodes, one can determine in constant time, by looking at the labels only, whether one node is an ancestor of the other.

Labelings currently being used are all variants of the following interval scheme. Number the leaves say from left to right and label each node with a pair consisting of the numbers of its smallest and largest leaf descendants. An ancestor query then amounts to an interval containment test on the labels. The maximum label length using this scheme is $2 \log n$, where n is the number of nodes in the tree.[1]

The focus of this work is finding a scheme that works best *in practice* on real XML data. We suggest an orthogonal prefix-based approach, where the labeling is such that an ancestor query roughly amounts to testing whether one label is a prefix of the other. We present several new labeling schemes based on this approach and analyze their performance both theoretically and empirically.

Keywords trees, ancestor queries, graph labeling, XML indexing, structural queries

---

*A preliminary version of this paper appeared the proceeding of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2002.

†School of Computer Science, Faculty of exact sciences, Tel-Aviv University, Tel Aviv 69978, Israel. E-mail: {haimk,milo,shabo}@post.tau.ac.il.

[1]All logarithms in this paper are to the base 2.

1 Introduction

The Web is constantly growing and contains a huge amount of useful information. To retrieve such data, people typically use search engines like Altavista [7] or Google [11] which provide full-text indexing services (the user gives a few words and the engine returns documents containing those words). The new emerging XML Web-standard [31] allows for more sophisticated querying of documents. XML allows to describe the semantic nature of the document components, enabling users not only to ask full-text queries (e.g. find documents containing the word "Fielding") but also utilize the document structure to ask for more specific data (e.g. find all book items containing "Fielding" as an author and a price less than 12$) [1, 3, 12, 17, 32, 33]. XML is being widely adopted as Web standard and it is believed that although a large portion of the Web will remain unstructured or hidden behind interactive graphics and forms, a large and content-wise essential portion of the Web will soon be available in XML [14, 36].

The problem studied in this work is the design of a compact index mechanism to support XML structural queries. More precisely, we develop compact labeling schemes of XML documents to be used by such an index. To understand this, consider the following rather typical implementation of an XML search engine [35, 36].

The key observation guiding the design of such an engine is that an XML document can be viewed as a tree (essentially the parse tree of the document) whose nodes are the document items and whose edges correspond to the "component-of" relationship among data items. With this view, a structural query amounts to finding nodes with particular tags (book, price, author, etc.) having certain ancestor relationship between them (e.g. book nodes that are ancestors of qualifying author and price nodes). To see an example consider the following query that searches for catalog books whose title contains the word "'database":

```
SELECT   x
FROM     *.catalog.*.book   x,
         x.title   y
WHERE    y contains "database"
```

Looking at the XML document (or more precisely, its corresponding parse tree) in Figure 1, the query retrieve all the `book` nodes that are *descendants* of a `catalog` node, and that are *parents* of a `title` node whose associated text (which is omitted from the figure) contains the word "databases".

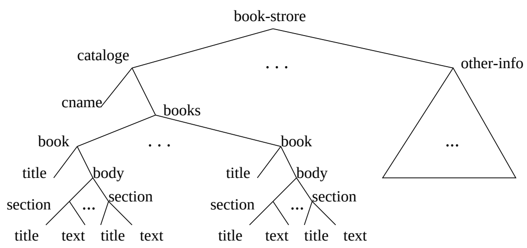


Figure 1: An XML tree

The heart of the search engine is an index structure summarizing ancestor relations. This is typically a big hash table [2] whose entries are the item tag names and the words in the indexed documents. To allow structural queries each item (node) in the XML documents (trees) is given a unique label, and every entry (item name or word) in the hash table is associated with the

[2]Or some alternative full-text indexing mechanism

list of labels of all the corresponding items (nodes). Furthermore, if items are given meaningful labels that reflect the hierarchical structure of the documents (namely, given labels of two items we can determine if one is a descendant, i.e. component, of the other), structural queries can be answered by using the index only, without access to the actual document. We first use the hash table to retrieve the items (in fact, their labels) of the relevant types/values (books, authors, price,...) and then iterate over the retrieved label sets and select those having the appropriate ancestor relationships.

To allow for good performance it is essential that the index structure (or at least a large part of it) reside in main memory. Observe that we are talking here about an extremely large memory size. Just to give a rough measure, it is estimated that the current number of Web pages is around 50 billion. Search engines typically index 15% of these pages, consuming around 0.3 of the original document size, (even when only the text with the document ids is indexed, not including separation to items or ancestor information). With 10KB text per page on the average, this already leads to about 25000 giga bytes. Since the length of the node labels is a main factor of the index size, reducing this, even by a constant factor, is a critical issue, contributing directly to hardware cost reduction and performance improvement. Our goal in this research is thus to design such a compact labeling scheme.

We assume in the sequel that all trees are such that each internal node has at least two children.[3] We denote by n the number of nodes in a tree T and by ℓ the number of leaves, (unless explicitly stated otherwise). Since we assume that each node in T has at least two outgoing edges, $n \leq 2\ell$.

Before presenting our results let us consider the labeling currently being used by actual systems. These are typically variants of the following *interval* scheme. We first label the leaves of T, say from left to right by consecutive integers starting from 1. We call these labels *leaf-labels*. Then we label each internal node v by a pair (interval) consisting of the label of v's leftmost leaf descendant and v's rightmost leaf descendant. Ancestor test for two nodes then amounts to an interval containment test for the corresponding labels. It is easy to see that the length of any label in this scheme is bounded by $2\log(\ell) \leq 2\log(n)$.

A considerable amount of work has been recently devoted to develop labeling schemes with worst case bounds smaller than $2\log n$. The best such labeling schemes use a recursive decomposition of the tree, and generates labels of length $\log n + O(\sqrt{\log n})$ [5]. This is in contrast with the best known lower bound which is $\log n + \Omega(\log\log n)$ [6]. Although theoretically best, these recursive schemes are not expected to improve on the simple interval scheme in practice. This is because the constant multiplying additive $O(\sqrt{\log n})$ is large and makes this term be the dominant when n is the current average size of XML file.

In contrast, we suggest in this work an orthogonal *prefix-based* approach, where the labeling is such that an ancestor query roughly amounts to testing whether one label is a prefix of the other. We present several new labeling schemes based on this approach and analyzed their performance both theoretically and empirically. Our study shows that these schemes are likely to be winners in practice.

We consider two possible physical representations of the data:

- In a fixed-length representation we allocate to each label the same fixed amount of space. Therefore, the longest possible label determines the length of every label, and the size of the index. Using a fixed-length representation, the lower the worst-case guarantee on the length of a label, the smaller the space we need to allocate for the index.

[3] If this is not the case to begin with we add a child to each internal node having only one child. This transformation at most doubles the number of nodes.

- In a variable-length representation each label may have a different length (consisting of a fixed prefix stating the actual length of the label, followed by the label itself). Here the size of the index is essentially determined by the average length of a label (rather then the maximum length of a label). However, the maximum length of a label also has some importance (even if smaller). This is because the length of the fixed prefix of every label is the logarithm of the maximum length of a label.

We tested our algorithms on a real XML data and on two families of random trees.

We first present a basic prefix labeling scheme that assigns to the outgoing edges of each node in the tree a set of prefix-free binary strings. The label of a node v is then the concatenation of the strings along the path from the root to v. This scheme already produce an index smaller by 10%-30% (depending on the physical representation) compared to the interval scheme on our XML data.

A possible problem with this simple scheme is that the length of the labels may become large for deep trees, e.g. if the tree is a long chain we can get labels of length $\Theta(n)$. Although such trees are rare in real data, some may occur. So in particular when a fixed length representation is used, a single skewed tree may increase the size of the index substantially.

To remedy this we use a standard *compression* mechanism which compresses (portions of) long paths into single virtual nodes. We then use alphabetic prefix free labeling of the compressed tree to label the original nodes. The ancestor test is still a simple prefix test together with an additional comparison. These labeling schemes are typically the best performers in practice. On our XML data the additional advantage which we gain from compression was not large since our trees are quite balanced. Nevertheless on a larger data set when fixed length representation is used, and few skewed trees are present, the gain could be substantial.

For these prefix labeling schemes with compression we are able to prove worst case upper bounds on the maximum length of a label which are comparable with the $2\log n$ upper bound on the length of a label of the interval scheme. We believe that this worst case guarantee on a practically good scheme is important to have since occationally skew trees may appear.

We also tested a two-level labeling schemes which arise in the recent theoretical work on labeling algorithms [2] and [5] (See section 2). In these schemes the tree is partitioned to a main top piece $\overline{T}$ and forest of pruned subtrees. We label $\overline{T}$ and the pruned subtrees separately and then combine these labels to label the nodes in T. Abiteboul *et al* [2] and Alstrup and Rauhe [5] used the interval scheme to label both $\overline{T}$ and the pruned subtrees.

Motivate by the good performance of the prefix labeling schemes we introduce a new two-level scheme which labels $\overline{T}$ with the interval scheme and the pruned subtrees with the compressed prefix labeling scheme. Then we give each node its final label by concatenating its own label in its subtree and the label of its lowest (leaf) ancestor in $\overline{T}$. On the theoretical side we can prove a worst case guarantee of $\frac{3}{2}\log n + O(\log\log n)$ on the maximum length of a label of the new mixed two-level scheme. Practically however both the new two-level scheme and the original one are not competitive on real XML files. Their hidden constant factors are too high to make them practical on typical XML file which is relatively small.

The purpose of our experiments with random trees was two fold: One goal was to estimate the additive constant factors of the two-level schemes so we can have a better idea of when they could be competitive. Another goal was to check how robust is the performance of our algorithms to the particular data set. Recall that for XML data, our study shows that in general prefix labeling schemes are effective in reducing the size of the index for both variable length and fixed length representations. Compared to the interval scheme our new schemes produce labels of total length smaller by about 30%. On random trees for fixed length representation prefix schemes were less efficient. But using variable length representation the compressed prefix schemes were better than the interval scheme. Our experiments with random trees also show that two-level schemes become competitive only for tree with at least 20000 nodes.

The structure of the work is as follows:

We overview work related to ours in Section 2. Sections 3, 4, and 5 describe the proposed prefix based approach. Section 3 describes the basic prefix labeling scheme, Section 4 describes the compressed prefix labeling scheme, and Section 5 describes the two-level approach mixing the interval scheme with the compressed prefix labeling scheme. Finally, the experimental results are presented in Section 6. We conclude in Section 7. Theorems which state upper bounds on the length of the labels produced by our algorithms are stated in the text, but the technical proofs are given in an Appendix.

2 Related work

A variant of the interval scheme was first described by Santoro and Khatib [29]. Instead of numbering the leaves from left to right, they use a depth first numbering of the tree. The label of a vertex v is its depth first number together with the depth first number of its last descendant in the traversal. Implementations of variants of the interval scheme are integrated in the Xyleme XML warehouse system [36].

The first to introduce a labeling scheme with a bound better than $2\log n$ were Abiteboul, Kaplan, and Milo [2], who suggested to use a two-level partition of the tree. Their best bound of $\frac{3}{2}\log n$ on the label length is obtained by applying this partition recursively $\log\log n$ times. Alstrup and Rauhe [5] building upon the work of Abiteboul *at el* describe a two-level labeling scheme producing labels of at most $\frac{3}{2}\log n$ bits, and a recursive scheme producing labels of length at most $\log n + O(\sqrt{\log n})$ bits. The main new ingredient that Alstrup and Rauhe added to the data structure of Abiteboul *at el* is alphabetic codes. The best known lower bound on the length of the shortest possible labels is $\log n + \Omega(\log\log n)$, proved by Alstrup, Bille, and Rauhe [6]

Peleg [28] considered informative labeling schemes of trees and graphs. In his paper Peleg formalizes the following general problem. Given a graph G and function f on subset of vertices, label the vertices of G such for every subset $W \subseteq V$ one can compute, or at least approximate, the value of $f(W)$ given only the labels of the vertices in W. Notice that our problem falls within this framework if we restrict G to be a tree and f is a boolean function of pairs of vertices such that $f(v,w)$ is true if and only if v is ancestor of w. Another particularly important and well studied problem within this framework is where $f(v,w)$ is the length of the shortest path from v to w. For such f the labeling scheme is called *distance labeling scheme*. Peleg computes upper and lower bounds on the maximum label length for various graph classes and functions f.

Maybe the problem closest to ours that Peleg studies is the *separation level* problem. In this problem $G = T$ is a tree and $f(v,w)$ is the depth of the lowest common ancestor of v and w. Peleg shows that the length of the labels needed for the separation problem is close (within an additive $\log n$ factor) to the length of the labels needed for a *distance labeling scheme* of T. From the results on distance labeling schemes [18, 13] of trees it follows that there is a labeling scheme for the separation level problem with labels of length $O(\log^2 n)$ bits, and that this bound is tight up to a constant factor. Peleg also provides a $O(\log^2 n)$ labeling scheme for the related problem where $f(v,w)$ is an identifier[4] of the lowest common ancestor of v and w in T, and proves that this bound

[4]The identifiers are given together with T as input.

is tight up to constant factor.

Several algorithms have been developed to preprocess a given tree so that subsequent lowest common ancestor queries[5] can be answered efficiently [9, 10, 15, 19, 22, 30]. The preprocessing part of these algorithms build data structures that allow to find lowest common ancestors and in particular, given v and w determine if v is an ancestor of w. These data structures, however, use global tables and are not organized in a distributed fashion as a labeling of the nodes. More recently, Alstrup, Gavoille, Kaplan, and Rauhe [4] presented an algorithm that labels the nodes of a rooted tree such that from the labels of two nodes, one can compute in constant time the label of their lowest common ancestor. The labels assigned by their algorithm are of length $O(\log n)$ and the algorithm runs in $O(n)$ time.

3 Simple prefix labeling

Our first simple prefix labeling scheme labels each node with a binary string such that node v is an ancestor of node u only if and only if the label of v is a prefix of the label of u.

Observe that when T is a binary tree, we can tag the two outgoing edges of each node by zero and one respectively and label a node v by the concatenation of the tags of the edges on the path from the root to v. Clearly then a node v is an ancestor of a node w if the label of v is a prefix of the label of w. Now, if T is not a binary tree, we can still play the same trick if we assign a binary string to each edge, such that the collection of strings assigned to the outgoing edges of any given node is *prefix-free.*

Definition 3.1 *We call an assignment of binary strings to the edges of the tree, such that the collection of strings associated with the outgoing edges from any node is prefix free, a* prefix free assignment.

Once we have such an assignment the label of a node v is now the concatenation of the strings assigned to the edges on the path to v, and, as before, ancestor testing amounts to prefix checking.

Minimizing the labels length. Naturally, we are interested in finding assignments that minimize the space consumption of the resulting node labels. The exact minimization function, however, depends on the physical representation of the labels:

- When a variable-length physical representation is used, namely different labels have different lengths, our goal is to find an assignment that minimizes the sum of the lengths of the labels (or equivalently the average label length). We call this optimization problem the *Minimum Sum of Labels* (MSL) problem.
- When a fixed-length physical representation is used, namely all labels have the same length (determined by the longest label), our goal is to find an assignment that minimizes the maximum length of a label. We call this optimization problem the *Minimum Maximum Label* (MML) problem.

To see the difference between the two problems consider for example the tree shown in Figure 2. To minimize the sum of the lengths of the labels we assign to the edges outgoing from the root the strings $00, 01, 10, 11$ respectively. All other nodes have exactly two outgoing edges to which we assign zero and one respectively (It will follow from Theorem 3.2 below that this is indeed the

[5]The *lowest common ancestor* of two nodes x, y is the common ancestor of both x and y whose distance to x (and to y) is smaller than the distance to x of any other common ancestor of x and y.

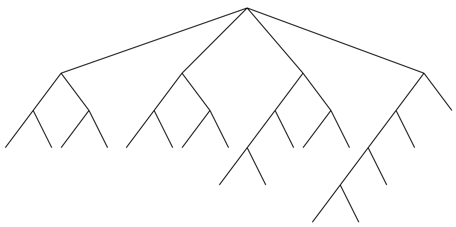


Figure 2: MSL vs. MML

optimum labeling). On the other hand, to minimize the maximum length of a label we will assign to the edges outgoing from the root the strings $000, 001, 01, 1$ respectively, which will make the maximum label length 5 rather than the 6 with the previous assignment. (Again, it will follow from Theorem 3.3 below that this is indeed the optimum labeling).

In the following we show how to solve both the MSL and the MML problems using variants of Huffman's algorithm [25] for constructing optimal prefix labels. We first give an overview of Huffman's algorithm and then explain how to adapt it to our case.

In the hereafter we denote the subtree of T rooted at a node v by T_v, and the number of nodes in T_v by $size(v)$.

3.1 The Huffman algorithm

Given a set of weights $w_1, \ldots, w_n$, Huffman's algorithm finds a set $b_1, \ldots, b_n$ of prefix-free binary strings that minimizes $\Sigma_{i=1\ldots n} w_i |b_i|$, where we use $|b_i|$ to denote the length of the string b_i. To find the strings, the algorithm constructs a binary tree, whose leaves are the given weights, that minimizes $\Sigma_{i=1\ldots n} w_i l_i$, where l_i is the number of edges on the path from the root to w_i. The corresponding strings are then obtained from the tree by tagging the outgoing edges of each node by zero and one and concatenating the tags along the path from the root to each w_i.

The binary tree is constructed iteratively as follows: We start with a forest of n singleton nodes of weights $w_1, \ldots, w_n$. At each step the algorithm combines the two lightest trees into one tree (by adding a new node as a parent of the two roots) whose weight is the sum of weights of the two lightest trees. The algorithm ends when there is only one tree in the forest. Huffman [25] proved that the final tree indeed minimizes $\Sigma_{i=1\ldots n} w_i l_i$ among all possible binary trees with $w_1, \ldots, w_n$ as their leaves.

3.2 Solving the MSL problem

The problem of finding a prefix free assignment that minimizes the sum of the lengths of the labels breaks into a collection of independent subproblems one for each internal node of the tree. Let v be an internal node, and let $u_1, \ldots, u_k$ be the children of v. We apply Huffman's algorithm to the (multi) set of weights $w_1, \ldots, w_k$ where $w_i = size(u_i)$, $1 \leq i \leq k$. (Recall that $size(u_i)$ is the size of the subtree rooted at u_i). Then we take the string b_i which Huffman's algorithm associated with w_i and assign it to the edge (v, u_i).

The following theorem shows that by applying the Huffman's algorithm, as explained above, independently to each node of the tree we obtain a prefix free assignment that solves the MSL problem.

Theorem 3.2 *For a given tree T, the above algorithm computes a prefix free assignment that minimizes the sum of the lengths of the resulting labels.*

Proof. Let s be a prefix free assignment that solves the MSL problem, and Let $s(e)$ be the string assigned by s to the edge e. By the definition of the MSL problem, s minimizes the sum $\Sigma_{v \in T} \Sigma_{e \in P(r,v)} s(e)$, where $P(r,v)$ stands for the set of edges on the path from the root, r, to v. Since every edge $(p(v), v)$ occurs on the paths to nodes in the subtree rooted at v we can rewrite the summation above as $\Sigma_{v \neq r} size(v) * s((p(v), v))$. It is clear that this sum is minimized if and only if the assignment to the edges between any node v and its children $u_1, \ldots, u_k$ is prefix free and minimizes $\Sigma_v size(u_i) * s(v, u_i)$. Since our algorithm produces such an assignment the theorem follows. □

3.3 Solving the MML problem

Our algorithm for solving the MML problem uses a variant of Huffman's algorithm which compute prefix-free collection of binary strings $b_1, \ldots, b_n$ that minimizes $max_{i=1 \ldots n}(w_i + |b_i|)$ rather than the sum $\Sigma_{i=1 \ldots n} w_i |b_i|$. We call this variant the H-max algorithm.

Given a set of weights $w_1, \ldots, w_n$ the H-max algorithm works as Huffman's algorithm merging at each iteration the two lightest trees. The difference is in the weight which H-max assigns to the new tree: Rather than taking the sum of the weights of the two merged trees, the H-max algorithm sets the weight of the new tree to be 1 + the largest weight among the two trees being merged. It is not hard to prove (see [23]) that indeed the H-max algorithm finds a string assignment which minimizes $max_{i=1 \ldots n}(w_i + |b_i|)$

Next we describe how to use the H-max algorithm to solve the MML problem. As in the solution of the MSL problem we apply the H-max algorithm for each internal node v in order to assign strings to the edges outgoing from v. But, while in the MSL algorithm each node could be treated independently of the others, here we have to solve the subproblems of all the descendants of v before we can apply the H-max algorithm to v: Let v be an internal node, and let $u_1, \ldots, u_k$ be the children of v. Recall that T_{u_i} denotes the subtree of T rooted at a node u_i. Suppose we have already computed a string assignment to all the edges in T_{u_i}, for every $1 \leq i \leq k$, and let w_i be the length of the longest concatenation of strings on the edges of a root to leaf path in T_{u_i}. To assign strings to the edges (v, u_i), $1 \leq i \leq k$, we apply the H-max algorithm to the weights $w_1, \ldots, w_k$ and assign the string which corresponds to w_i to the edge (v, u_i).

The following theorem establishes the correctness of the algorithm above.

Theorem 3.3 *Given a tree T, the above algorithm computes a prefix free assignment that minimizes the maximum length of a resulting label.*

Proof. By induction on the depth of T. If the depth of T is one then the theorem follows from the correctness of the H-max algorithm.

Assume that the theorem holds for all trees of depth ℓ and let T be a tree of depth $\ell + 1$. Let v be the root of T, let $u_1, \ldots, u_k$ be the children of v, and let L be the labeling produced by our algorithm.

It is easy to see that there exists an optimal labeling L^{opt} of T which induces an optimal labeling of T_{u_i} for every $1 \leq i \leq k$. (Otherwise, we can replace the labeling of L^{opt} inside T_{u_i} with an optimal labeling for T_{u_i} without increasing the maximum length of a label.) Therefore by induction the maximum length of a label of L restricted to T_{u_i}[6] equals to the maximum length of a label of L^{opt} restricted to T_{u_i}.

[6]A restriction of a labeling of T to T_{u_i} is the labeling of T_{u_i} which assigns to the edges of T_{u_i} the same strings that T assigns.

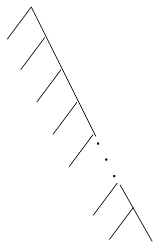


Figure 3: A tree with long prefix labels

Consider the labels assigned by L^{opt} to the edges (v, u_i) for every $1 \leq i \leq k$. Let a_i be the length of the label assigned by L^{opt} to the edge (v, u_i). Let b_i be the length of the label assigned by L to the edge (v, u_i). From the correctness of H-max follows that $max_{i=1\ldots n}(w_i + |b_i|) \leq max_{i=1\ldots n}(w_i + |a_i|)$, where w_i is maximum length of a label of L and L^{opt} restricted to T_{u_i}. Therefore L is also an optimal labeling. $\Box$

In contrast with the interval scheme of Section 1, the prefix scheme presented in this section does not have a logarithmic worst-case upper bound on the maximum label length. The labels derived from the prefix-free assignment solving the MML problem (and from the one solving the MSL problem) highly depend on the structure of the tree. For skewed trees the maximum label length can be $\Omega(n)$. As a simple example consider the tree in Figure 3. The label of the deepest leaf will have $n/2$ bits.[7] It turns out however that by a rather simple preprocessing of the tree, which compresses such long paths, one can significantly reduce the worst-case upper bound on the label length. In the next section we will present a prefix based labeling scheme that for all trees (i.e. even for trees as in Figure 3) the maximum label length is $2 \log n$.

3.4 Labeling a weighted tree

A typical XML index contains entries not only for the XML items appearing in the documents (i.e. the nodes in the document tree) but also for the words appearing in the document text. Looking at the document parse tree, this text appears at the leaves. It follows that while an internal node of an XML tree appear in the index just once, a leaf may appear several times, ones per word occurring in the text at that leaf.

When the physical representation of labels is variable-length, hence labels have different lengths, we may want to take the number of words occurring in each leaf into account - we would like the labels of leaves containing large number of words to be shorter than those having a smaller number of words (and perhaps even shorter than labels of some internal nodes). As we explain above, in contrast with the interval labeling scheme, it is easy to adapt our prefix scheme to a setup where nodes have different weights. The core of our simple prefix scheme tailored for variable length representation was the algorithm for solving the MSL problem. If weights of nodes vary we modify this algorithm such that it minimizes the weighted sum of lengths of the labels rather than their unweighted sum. As result nodes with high weights will get smaller labels.

The required modification is as follows. We define a weight $w(u)$ for each node $u \in T$ to be equal to the number of words that u contains. We generalize the definition of $size(u)$ to be the sum of the weights of the nodes in the subtree rooted by u. (Note that when all weights are one then $size(u)$ is the size of the subtree as we defined it before.) We solve the MSL problem on T with the

[7]The maximum label length is $O(n)$. To see that one can binarize the tree by replacing the outgoing edges of each internal node v by a arbitrary binary tree with the children of v at its leaves, and label the resulting binary tree.

same algorithm as in Section 3.2 but using the modified sizes when we solve the Huffman problem associated with each node. A proof analogous to that of Theorem 3.2 shows that this algorithm indeed minimizes the weighted sum of the lengths of the labels, $\sum_{u \in T} w(u)|label(u)|$.

4 Compressed prefix labeling

In this section we enhance the prefix labeling with a preprocessing step that first compresses the tree. As a result labels of nodes even in a highly skewed trees cannot get too large. The penalty is a slightly more complicated ancestor test which incorporates another comparison in addition to the prefix decision. We will use the following notation. We denote the concatenation of two string x and y by $x.y$. Given strings b_1, b_2, b_3 such that $b_2 = b_1.b_3$ we denote b_3 by $suffix(b_2, b_1)$. We say that s_1 is *smaller* than s_2, and denote it by $s_1 < s_2$, when s_1 precedes s_2 in the lexicographic order of binary strings.

At a high level the algorithm works as follows. We decompose the tree T into disjoint paths, and construct a (partially ordered) compressed tree $\hat{T}$ by contracting each path into a big virtual node (and defining a partial order on the children of the node). We first label $\hat{T}$ using a scheme similar to the simple prefix labeling scheme of Section 3. Once $\hat{T}$ is labeled we label each original node based on the label of the virtual node (path) to which it belongs. The label of a node v in T will be composed of two substrings $code(v)$ and $separator(v)$. All nodes on the same path will have the same *code* (which is essentially the label of the node corresponding to the path in $\hat{T}$), and will be distinguished by their *separator*. Furthermore, we assign codes and separators such that for every two nodes $v, u \in T$, v is an ancestor of u iff the following two conditions hold.

(i) $code(v)$ is a prefix of $code(u)$, and

(ii) $separator(v) \leq suffix(code(u), code(v)).separator(u)$. ($\leq$ here stands for a lexicographically comparison between the two strings.)

We show how to choose the labels such that the length of the code concatenated with the separator of each node is bounded by $2\log n$ bits. One can then encode each label using $2\log n + \log\log n + 2$ bits: The encoding consists of two fixed length fields one of length $2\log n + 1$ bits and the other of length $\log\log n + 1$ bits. The first field contains the concatenation $code(v).separator(v)$, padded at the beginning with a sequence of '0's ending with a single '1'. (The '1' denotes where in the field the actual data starts). The second field tells us how to decode the first field, namely, in which position the *code* ends and the *separator* starts.

The rest of this section is organized as follows. Section 4.1 describes how to decompose the tree into paths and construct the (partially ordered) compressed tree. Section 4.2 describes how to use an appropriate labeling of $\hat{T}$ to label T and how to answer ancestor queries using these labels. Finally, Section 4.3 discusses various ways to label $\hat{T}$.

4.1 Compressing the tree via path decomposition

As explained above, $\hat{T}$ is obtained in three steps. First the tree T is decomposed into paths, next each path is compressed into one virtual node and, finally, a partial order is defined on the tree nodes, based on the above construction.

Path decomposition To explain how the tree is decomposed into paths, we first need to define the paths that we are interested in. For a node v we define the *path of* v, denoted by $path(v)$, as the set of all the nodes in T_v with weight $\geq size(v)/2$. Note that this set of nodes indeed form a path since each of v's descendants can have at most one child with weight $\geq size(v)/2$. We partition

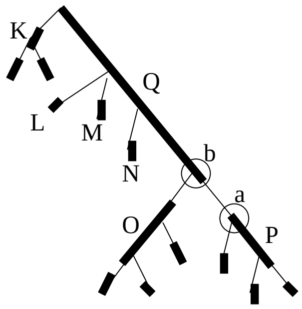


Figure 4: Path decomposition

the tree into paths recursively as follows. We start from the root v of T and work top down. We first find $path(v)$ and mark it as a new path. Then we continue recursively from every node not on $path(v)$ that is a child of a node on $path(v)$ and decompose all the subtrees rooted at these nodes in the same way.

For a path p, we denote by $head(p)$ the node on p closest to the root, and by $parent(p)$ the parent of $head(p)$ in T. Our decomposition algorithm is illustrated in Figure 4. The decomposed paths are marked with thick dark lines. The head and parent of the path P are marked by a and b respectively.

Constructing the compressed tree Once the tree T is decomposed into paths, we use the path decomposition to construct a new compressed tree $\hat{T}$. One can visualize $\hat{T}$ as the tree obtained from T by contracting each path into a single node. The nodes of $\hat{T}$ are the paths of the path decomposition and a node $p \in \hat{T}$ is a child of a node $q \in \hat{T}$ if and only if $parent(p) \in q$. It is easy to see that the depth of $\hat{T}$ is at most $\log(n)$. (Since if q is a parent of p in $\hat{T}$ then $size(head(p)) < size(head(q)/2)$

For a node $v \in T$, the *representative* of v in $\hat{T}$, denoted by $rep(v)$, is the node p corresponding to the path that contains v.

Defining the partial order on the tree nodes Finally, we define a partial order on the nodes of $\hat{T}$, as follows. For every internal node $p \in \hat{T}$ we define a partial order $\prec_p$ on the children of p such that $q \prec_p q'$ if and only if $parent(q)$ is a proper ancestor of $parent(q')$. We will omit the subscript p from $\prec_p$ without confusion since every node in $\hat{T}$ participates in exactly one such partial order. To see an example, in the tree of Figure 4, $\hat{T}$ is obtained by contracting each thick dark line into a single node. In the resulting compressed tree K, L, M, N, O, P are children of Q and the partial order on them is $K \prec \{L, M\} \prec N \prec \{O, P\}$.

Let p be a node which corresponds to a contracted path $v_1, \ldots, v_m$ of length m in T. We can partition the children of p into m sets, $S_1, \ldots S_m$, where S_i contains all children q of p such that $parent(q) = v_i$. The partial order $\prec_p$ determines the order between any pair of children of p that are in different sets. Pairs that are in the same set are unrelated.

4.2 Assigning labels

In the previous section we introduced the notion of a *prefix free assignment*, and used it to define the labels of the tree nodes. To label $\hat{T}$ we need to refine this notion so as to take the partial order $\prec$ on the tree nodes into consideration.

Definition 4.1 *Consider a tree $\hat{T}$ and a partial order $\prec_p$ for every internal node $p \in \hat{T}$ relating its children. An* ordered prefix free assignment *for $\hat{T}$ (w.r.t. the partial orders $\prec_p$) is a prefix free assignment to the edges of $\hat{T}$ that respects $\prec_p$. Namely, an assignment such that whenever q and q' are children of p and $q \prec_p q'$, then the string assigned to the edge (p, q) is smaller than that assigned (p, q').*

Now, given a tree T to label, we first find an ordered prefix free assignment to the compressed tree $\hat{T}$. (We describe how to do that in the next section.) Once having such an assignment to $\hat{T}$, the label of a node p in $\hat{T}$ is simply the concatenation of the strings assigned to the edges of $\hat{T}$ along the path from the root to p. Finally, we define the labels of the nodes in T based on the labels of the compressed tree $\hat{T}$. Recall that the label of a node v consists of two parts, *code*(v) and *separator*(v). We define *code*(v) to be the label of *rep*$(v) \in \hat{T}$, and *separator*(v) to be the smallest (lexicographically) string assigned to the outgoing edges of *rep*(v) which point to some representative of v's children, or the empty string if *rep*(v) is a leaf node.

Our ancestor test is based on the following theorem. The proof of this theorem follows from the fact that the codes and separators were generated based on an ordered prefix free assignment.

Theorem 4.2 *If v_1, v_2 are on the same path (hence represented by the same node in $\hat{T}$) then v_1 is an ancestor of v_2 iff* $\text{separator}(v_1) < \text{separator}(v_2)$.
If v_1, v_2 are on distinct paths (hence represented by distinct nodes in $\hat{T}$), v_1 is an ancestor of v_2 iff
(i) $\text{rep}(v_1)$ *is an ancestor of* $\text{rep}(v_2)$ *in $\hat{T}$ (hence* $\text{code}(v_1)$ *is a proper prefix of* $\text{code}(v_2)$*), and*
(ii) $\text{separator}(v_1) \leq \text{suffix}(\text{code}(v_2), \text{code}(v_1))$.

Proof. The first part of theorem follows immediately from the definitions of the labels and the definition of an ordered prefix free assignment. So let v_1 and v_2 be two nodes of T on distinct paths.

Assume first that v_1 is an ancestor of v_2. It follows from the definition of $\hat{T}$ that *rep*(v_1) is an ancestor of *rep*(v_2). Let $x \in T$ be the deepest ancestor of v_2 on the path which corresponds to *rep*(v_1). Since v_1 is an ancestor of v_2 in T then x must be a descendant of v_1. Let y be the child of x on the path from x to v_2 in T. By the definition of the lexicographic prefix free assignment to the edges outgoing from *rep*(v_1) the string assigned to the edge $(\textit{rep}(v_1), \textit{rep}(y))$ is at least as large lexicographically as the separator of v_1. Since the string assigned to the edge $(\textit{rep}(v_1), \textit{rep}(y))$ is a prefix of $\textit{suffix}(\textit{code}(v_2), \textit{code}(v_1))$ the statement follows.

The proof of the other direction where we assume that (i) *rep*(v_1) is a proper ancestor of *rep*(v_2) in $\hat{T}$ and (ii) $\textit{separator}(v_1) \leq \textit{suffix}(\textit{code}(v_2), \textit{code}(v_1))$, and have to show that v_1 is an ancestor of v_2 is similar. □

The following corollary which immediately follows from Theorem 4.2 specifies our ancestor test precisely.

Corollary 4.3 *For two nodes v_1, v_2 in T, v_1 is an ancestor of v_2 iff the following hold.*
(i) $\text{code}(v_1)$ *is a prefix of* $\text{code}(v_2)$*, and*
(ii) $\text{separator}(v_1) \leq \text{suffix}(\text{code}(v_2), \text{code}(v_1)).\text{separator}(v_2)$.

4.3 Finding an ordered prefix free assignment of $\hat{T}$

To conclude we need to explain how to find an appropriate ordered prefix-free assignment for $\hat{T}$. For fixed length representation we find an ordered prefix-free assignment of $\hat{T}$ that minimizes the maximum length of a label of a node in $\hat{T}$. Note that by the definition of our path decomposition there is bijection between the leaves of $\hat{T}$ and the leaves of T. Therefore among all ordered prefix-free assignments of $\hat{T}$ the one that minimizes the maximum length of a label in $\hat{T}$ also minimizes the maximum length of a label of a node in T.

For variable length representation we associate a weight $w(p)$ with each node of $p \in \hat{T}$. We define the weight $w(p)$ to be equal to the length of the path represented by p. We compute an ordered prefix free assignment of $\hat{T}$ that minimizes the weighted sum of the labels of the nodes of $\hat{T}$, that is minimizes $\sum_{p \in \hat{T}} w(p)|label(p)|$. Note that since the label of a node $p \in \hat{T}$ is used as the code part of exactly $w(p)$ nodes in T, an ordered prefix free assignment of $\hat{T}$ that minimizes $\sum_{p \in \hat{T}} w(p)|label(p)|$ also minimizes the sum of the code parts of the resulting labels of nodes of T.

Finding an ordered prefix free assignment of $\hat{T}$ that minimizes the resulting labels of nodes of T (the codes and the separators) is a harder problem that we do not know how to solve. The difficulty is the following. Let p be a node in $\hat{T}$ and let v be a node in T which belongs to the path represented by p. Note that the edges outgoing of v in T correspond to edges outgoing of p in $\hat{T}$, but the children of p corresponding to these edges are unrelated in the partial order associated with p. Among the labels assigned to these edges we use the lexicographic smallest as a separator in the label of v. If we would have known in advance which edge will get the lexicographic smallest label we could modify the weights so we would get an optimal labeling. The problem is that we do not know in advance which edge would get the smallest label.

Remark: If nodes in T have different weights because of different amount of text that they contain as described in 3.4, then we can adapt the compressed prefix scheme to consider these weights and produce smaller labels for heavy nodes. We do that simply by defining the weight of a node $p \in \hat{T}$ to be the sum of the weights of the nodes of T that are on the path represented by p (rather than the length of this path). Our algorithm would then minimize the sum of the code parts of the labels of T assuming that the number of occurrences of each label in the index is equal to the weight of the node.

To find labelings of $\hat{T}$ as described above we define optimization problems OMSL (Ordered MSL) and OMML (Ordered MML) analogous to the MSL and the MML problems of Section 3. To both problems the input is a tree T, together with a partial order $\prec_v$ for every internal node v, and the output is an ordered prefix free assignment. In the OMML problem we look for an assignment that minimizes the length of the longest label. In the OMSL problem the tree is weighted, each node p has a weight $w(p)$, and the goal is to find an ordered prefix free assignment that minimizes the weighted sum of the lengths of the resulting labels.

Before we explain how to solve these problems, lets look again at the corresponding unordered versions of the problems and highlight the differences between the two versions. Recall that to solve the MSL and MML problems we considered each node separately and used Huffman's algorithm (or its variant H-max) to assign strings to the outgoing edges of the node. Observe that we cannot use these algorithms here since they do not provide any guarantee on the lexicographic order of the resulting strings. To illustrate this consider the tree in Figure 5, and assume that the three edges outgoing from the root are ordered from left to right. Let u_1, u_2, and u_3 be the children of the root. Using Huffman's algorithm on the weights $size(u_1) = 3$, $size(u_2) = 7$, and $size(u_3) = 3$, the edges outgoing from the root are assigned the strings $00, 1, 01$ (or $11, 0, 10$) respectively, since the two lighter subtrees will be merged first. (All other nodes have two outgoing edges, assigned zero and one respectively.) While this makes the sum of the lengths of the labels minimal[8], the order defined on the children of the root is not respected by the lexicographical order of the assigned

[8]In this example this assignment also minimizes the maximum length of a label.

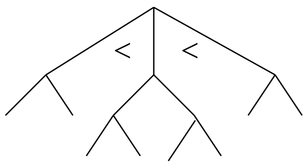


Figure 5: Ordered vs. unordered labeling.

strings. The shortest possible string assignment respecting this order is $00, 01, 1$ (or $0, 10, 11$). This increases the maximum length of a label to 4 (rather than 3 with the previous assignment) and it increases the sum of the lengths of the labels as well.

4.3.1 Solving the OMSL and OMML problems

Nevertheless, it turns out that we can still solve the OMSL and OMML problems using a general scheme similar to that used for MSL and MML. Namely, we solve a separate problem for each individual node to assign binary strings to its outgoing edges.

Let p be an internal node, and let $q_1, \ldots, q_k$ be the children of p. Let $w_i = size(q_i)$ and define $w_i \prec w_j$ iff $q_i \prec_p q_j$. To solve OMSL we need to find a set of prefix free binary strings $b_1, \ldots, b_k$ such that $b_i < b_j$ iff $w_i \prec w_j$ and $\Sigma w_i |b_i|$ is minimized. Equivalently we need to find a binary tree with leaves $w_1, \ldots, w_k$ that minimizes $\sum w_i \ell_i$, where ℓ_i is the length of the path to the leaf w_i, subject to the constraint that w_i must preceded w_j in an in-order traversal of the tree if $w_i \prec w_j$. Similarly for OMML we need to find a binary tree minimizing $\max_{i=1..k}(w_i + \ell_i)$ where w_i is now the depth of the recursive solution obtained for the subtree rooted by q_i. Again here the tree should be such that w_i must preceded w_j in an in-order traversal of the tree if $w_i \prec w_j$.

It is not hard to see that theorems analogous to Theorem 3.2 and 3.3 hold for these algorithms. So if we solve optimally the problem corresponding to each internal node we indeed obtain optimal solutions for the OMSL and the OMML problems.

How do we optimally solve the local problem defined above at each node ? Recently, Barkan and Kaplan [8] described algorithms that solve these problems in time $O(\left(\frac{W_{sum}}{W_{min}}\right)^{2\alpha} \log(\frac{W_{sum}}{W_{min}})k^2)$ time where $W_{sum} = \sum_{i=1}^{n} w_i$, $W_{min} = \min_i w_i$, and $\alpha = \frac{1}{\log \phi} \approx 1.44$ [9]. Since in our case the weights are linear in the size of the tree this running time is polynomial. Using the algorithm of Barkan and Kaplan for the local problem in each node we obtain an algorithm that finds the optimal solution for the OMML or the OMSL problem.

Barkan and Kaplan obtained their result after we carried out our experiments, therefore we have not used their algorithms to solve the OMSL and OMML problems. Instead we implemented different algorithms which find a good assignment that is not necessarily optimal. Note however, that the upper bounds which we establish below on the maximum length of a label produced by these algorithms for the OMML problem, hold for the optimal algorithm as well.

Method 1: Choose an arbitrary total order When the partial order on the set of weights $w_1, \ldots, w_k$ is in fact a total order, i.e. $w_1 \prec w_2 \prec \ldots \prec w_k$ then there are well known polynomial time algorithms for finding an optimal prefix free set of strings $b_1 < b_2 < \ldots < b_k$ that minimize either $\Sigma_{i=1\ldots k}(w_i|b_i|)$ or $max_{i=1\ldots k}(w_i + |b_i|)$. These algorithms in fact compute a binary tree with k leaves corresponding to $w_1, \ldots w_k$ from left to right in symmetric order, that minimizes $\sum w_i \ell_i$ where

[9] $\phi = \frac{\sqrt{5}+1}{2} \approx 1.618$ is the golden ratio

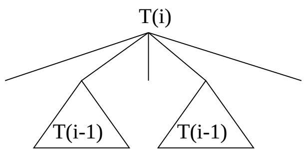


Figure 6: Tree with labels of length $3\log(n)$

ℓ_i is the depth of w_i. Such a tree is called an *alphabetic tree* for $w_1, \ldots, w_k$, and the corresponding strings b_i form an *alphabetic code* for $w_1, \ldots, w_k$. One algorithm to find alphabetic tree is based on dynamic programming and runs in $O(k^2)$ time [21, 26]. Algorithms with running time $O(k \log k)$ are also known [24, 20].

We can use an algorithm for a totally ordered set of weights to obtain a solution to the local problem in each node while solving the OMSL or the OMML problem. To label the outgoing edges from an internal node p we pick an arbitrary total order $\prec_p^t$ consistent with $\prec_p$ and find an optimal string assignment consistent with this total order using one of the algorithms mentioned above. The optimality criteria and the particular algorithm we use vary according to whether we solve the OMSL problem or the OMML problem.

For the OMML problem we can establish the following bound on the maximum label length generated by this algorithm. The proof is provided in the Appendix.

Theorem 4.4 *Let $\hat{T}$ be a tree with a given partial order $\prec_p$ in each internal node p. Assume that we choose a total order, $\prec_p^t$ in each internal node p consistent with $\prec_p$, and then solve the OMML problem by labeling the edges outgoing from p using an optimal string assignment consistent with this total order. Then the length of the labels is bounded by $\log(\hat{n}) + 2d$, where d is the depth of $\hat{T}$, and $\hat{n}$ is the number of nodes in $\hat{T}$.*

Since the number of nodes in $\hat{T}$ is no larger than the number of nodes of T, and the depth of $\hat{T}$ is at most $\log(n)$, where n is the number of nodes in T, Theorem 4.4 implies a bound of $3\log(n)$ on the maximum label length of a node in T produced by our algorithm for the OMML problem. The following example shows that this bound is tight. Let $T(i)$, $i \geq 1$ be a tree with $2^{i+1} - 4$ nodes, having a recursive structure as described in Figure 6. (For $i = 1$ it is simply the empty tree). Note that since each subtree here weights less than half of its parent, $\hat{T}$ is equal to T. Now, assume that we run the above algorithm on the tree, such that the arbitrary total order chosen at each node v order the outgoing edges of v consecutively from left to right. It is easy to see that any order preserving prefix-free string assignment must assign a string of length 3 to at least one edge pointing to a $T(i-1)$. Since this happens at each level of the recursion, there must be at least one leaf with labels of length $3 \times$ (*the height of the tree*), which, up to a small additive constant equals $3\log(n)$.

Nevertheless, it turns out that we can improve the above algorithm and obtain a better upper bound on the maximum label length. To achieve that we avoid completing each partial order into an arbitrary total order (thereby creating unnecessary constraints) but, instead, take advantage of the unordered parts of the tree. Going back to the above example, since in this case $\hat{T}$ is completely unordered, we could have avoided ordering the tree altogether, run the MML algorithm on it, and obtain labels of length at most $2\log(n)$. This leads us to the second algorithm described below.

Method 2: Exploiting the structure of the partial order Let p be a node in $\hat{T}$ that corresponds to a path $v_1, \ldots, v_m$ of length m in T, and let S_i be the set of children of p whose parent is v_i. Our definitions of the path decomposition and the compressed tree imply that the total size of nodes in S_m is at least half the total size of all children of p, i.e. $\sum_{u \in S_i, i<m} size(u) \leq \sum_{u \in S_m} size(u)$. Our improved algorithm for computing an ordered prefix free assignment to $\hat{T}$ exploits this fact.

We split the children of p into two groups as follows. The first group consists of the children in the sets $S_1, \ldots, S_{m-1}$. (Recall that S_i is the set of children q such that $parent(q)$ is the i^{th} node on the path corresponding to p.) The partial order $<_p$ induces a partial order on this set of children. The second group is an unordered group consisting of all children in the last set S_m. Note that the total weight of the elements in the first group is no larger than the total weight of the elements in the second group.

Now we can treat each group separately. First we complete the partial order defined on the first group to an arbitrary total order and find an optimal string assignment consistent with this total order using one of the algorithms mentioned above. Second we run Huffman's algorithm to assign strings to nodes in the unordered second group. Finally we have to combine the string assignments of the two parts into one string assignment consistent with the original partial order $\prec_p$. We do that by adding a leftmost zero to the strings of the ordered part and a leading one to the strings of the unordered part. Intuitively we are able to get shorter labels since the weight of the unordered set is large and we avoid imposing unnecessary constraints on its elements.

In the Appendix we prove that a somewhat more involved version of the this algorithm produce labels of length at most $2\log n$. As a corollary we obtain that the algorithm which optimally solves the OMML problem in fact produce labels of length at most $2\log n$. Specifically we prove the following theorem.

Theorem 4.5 *Let T be a tree, and let $\hat{T}$ be the compressed tree of T. There is an algorithm that finds an ordered prefix free assignment of $\hat{T}$ such that the length of the resulting labels of nodes in T is bounded by $2\log n$.*

5 The two-level algorithm

The best guarantee we have so far (in terms of n) on the maximum length of a label is about $2\log n$. We get such an upper bound using either the interval scheme or the compressed prefix scheme of Section 4.3.1. In this section we describe two labeling schemes with a bound of roughly $\frac{3}{2}\log n$ on the maximum label length. We obtain these schemes by combining the ideas undelying previous schemes with a partition of the tree.

Recently, Abiteboul, Kaplan, and Milo [2] also introduced labeling schemes with bounds better than $2\log n$ on the maximum label length. The simple scheme suggested by Abiteboul *et al* [2] is based on a two-level partition of the tree, into a main top piece $\overline{T}$ and a forest of pruned subtrees. The tree $\overline{T}$ and each of the pruned subtrees is labeled independently using the interval scheme. The final label of a pruned vertex is obtained by a combination of its label from its own subtree and the label of its lowest (leaf) ancestor in $\overline{T}$. When we determine whether v is an ancestor of w there are three possible cases. If v and w are in different subtrees (hence their labels contain different ancestor leaf labels from $\overline{T}$) then they are clearly unrelated. If v and w belong to the same subtree we can determine whether v is an ancestor of w by comparing their interval labels within the subtree. Finally, if v is in $\overline{T}$ and w is in a pruned subtree we can decide by comparing the label of w's leaf ancestor in $\overline{T}$ (that is incorporated in the label of w) to the label of v.

Abiteboul *et al* [2] showed that the lengths of the labels generated by this two-level scheme are bounded by $\frac{5}{3}\log n$. Intuitively, the scheme takes advantage of the fact that, in the interval scheme, the length of a leaf-label is half the length of a label of an internal node. It obtains the better bound on the maximum label length by balancing out the length of a leaf-label with the length of the label of an internal node. Abiteboul *et al* also show how to apply their two-level scheme recursively $\log\log n$ times to obtain an even better bound on the maximum label length. Their best scheme has labels of length at most $\frac{3}{2}\log n$. More recently, Alstrup and Rauhe [5] improved the results of Abiteboul *et al* [2] using alphabetic trees [21, 26, 24, 20]. They describe a two-level scheme that produce labels of length $\frac{3}{2}\log n$ and a recursive scheme that produce labels of length $\log n + O(\sqrt{\log n})$.

In this section we suggest a two-level scheme similar to the one of Abiteboul *et al* [2] and Alstrup and Rauhe [5]. In contrast with [2, 5] our new scheme uses the interval scheme only to label the top level tree $\overline{T}$. It labels the pruned subtrees using the compressed prefix scheme of Section 4.3.1. We call this scheme the *mixed two-level scheme*. We prove that the lengths of the labels of the mixed two-level scheme are bounded by $\frac{3}{2}\log n + O(\log\log n)$.

Our motivation to consider these recursive algorithms is their good theoretical guarantees. However, by inspecting the algorithms it is obvious that as the recursion gets deep the constants hidden behind the asymptotic terms gets high. To estimate how bad these constants are, and what is their effect in practice we focus on two-level schemes that use only one level of recursion. The motivation for suggesting our new mixed two-level scheme is the observed good performace of prefix based labeling on XML data. The mixed two level scheme uses prefix based labeling whithin the recursive subproblems.

The rest of this section is organized as follows. Section 5.1 described the two-level tree partitioning algorithm of [2]. This algorithm is used both by the two-level labeling scheme of [2, 5], described in Section 5.2, and by our new mixed two-level labeling scheme described in Section 5.3.

5.1 The tree partitioning algorithm

The algorithm uses a parameter x that controls the size of the pruned subtrees. This parameter is set to $\frac{1}{3}$ by the algorithm of [2] and to $\frac{1}{2}$ by the improved two-level scheme of Alstrup and Rauhe [5] and by our new mixed two-level scheme. We recall that ℓ denotes the number of leaves in T. The algorithm has three phases. In our description below we omit some details that can be found in [2].

- The first phase prunes from T all subtrees having ℓ^x leaves or less, by cutting the edges leading to these subtrees. For example consider the tree in Figure 7(a). The tree has 27 leaves. When $x = \frac{1}{3}$ all subtrees containing three or less leaves are pruned. The dashed lines in Figure 7 indicate the pruned subtrees.
- The second phase groups the pruned subtrees into forests each having between ℓ^x and $2\ell^x$ leaves. We do the grouping such that the subtrees in each forest are either all children of a single node (as in the case of the forests B-E in Figure 7(b), or they are the set of children of some chain of nodes (as in the case of forest A. The nodes of the chain are denoted by a_1-a_3).
- The third phase adds to the remaining part of the tree a *representative leaf* per each pruned forest. When the parents of the trees in the forest form a chain we first contract the chain into a single node and attach the new representative leaf as a child of that node. Otherwise we attach the leaf to the parent of the subtrees in the forest. To continue with the above example, in Figure 7(c) the leaves a-e represent the forests A-E respectively, and the node a^* corresponds to the contracted chain a_1-a_3.

Let $\overline{T}$ denotes the tree resulting from the above algorithm after pruning, contraction, and adding representatives. The following lemma proved in [2] bounds the number of leaves in $\overline{T}$ and

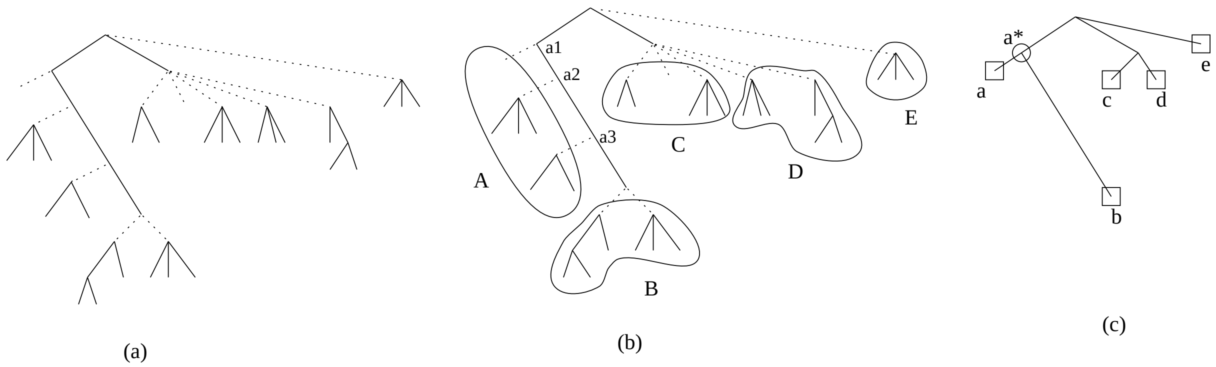


Figure 7: The pruning algorithm

the lengths of the contracted chains. We shall use these lemma to establish the bound on the label lengths.

Lemma 5.1 *[2] The number of leaves in $\overline{T}$ is at most $3\ell^{1-x}$ where ℓ is the number of leaves in T.*

Lemma 5.2 *[2] The length of each contracted chain is at most $2\ell^{x}$.*

5.2 The basic two-level interval scheme

The high level idea is to label independently $\overline{T}$ and each of the pruned subtrees, using the basic interval scheme, and then give the pruned nodes a label which is composed of their label in their own subtree together with the label of their representative leaf in $\overline{T}$. The contractions of chains in phase three of the pruning scheme introduce some difficulties. We have to distinguish between different nodes on the same contracted chain and label them such that we can resolve ancestor queries involving one or two such nodes.

To explain how we settle these difficulties we first extend the notion of *representative* and define a representative node in $\overline{T}$ for each node of T as follows.

Definition 5.3 *Let v be some node in T. The* representative *node of v in $\overline{T}$, denoted by $r(v)$, is defined as follows. If v was pruned from T then it is represented in $\overline{T}$ by the new leaf corresponding to the forest to which v belongs. If v was contracted as part of a chain into some node c in $\overline{T}$ then $r(v) = c$. Otherwise, $r(v) = v$.*

Following the improvement of Alstrup and Rauhe [5] we label the nodes on each contracted chain by lexicographically increasing binary strings, such that the label of a vertex closer to the root is lexicographically smaller. We assign those strings using an algorithm for constructing an alphabetic code such that the length of a string assigned to a node v on the chain is inversely proportional to the size of the subtrees that were pruned off v.[10]

For a vertex v that belongs to a contracted chain we define the *index* of v to be its lexicographic label as defined above. For a vertex v not on a contracted chain that has an ancestor u on a contracted chain we define the *index of* v to be the index of the lowest ancestor of v on a contracted chain.

[10]This is similar to the way we label the edges outgoing from a node when solving the MSL problem using the first method of Section 4.3.1.

The following theorem from [2] characterizes when v is an ancestor of w in T. The characterization is mainly in terms of the relation between the corresponding representatives $r(v)$ and $r(w)$. The first two conditions are rather straightforward; the proof of the third appears in [2]. The basic two-level interval labeling scheme which we describe below captures exactly the information needed to apply this characterization.

Theorem 5.4 *[2] Vertex v is an ancestor of vertex w in T iff one of the following conditions holds.*

1. *Vertices v and w are both pruned and have a common representative leaf in $\overline{T}$, (so v and w belong to the same pruned forest), and v is an ancestor of w in that forest.*
2. *Vertices v and w belong to a chain that has been contracted into a single node of $\overline{T}$, and the index of v is smaller than the index of w.*
3. *Vertex $r(v)$ is an ancestor of $r(w)$ in $\overline{T}$. If v belongs to a contracted chain and w belongs to the pruned forest of that chain, we also require that the v's index is no greater than that of w.*

We are now ready to define the labels. We first label $\overline{T}$ and each of the pruned forests, independently, using the interval scheme[11]. Then we define our labels according to the type of the node as follows.

Case 1. Node v is in $\overline{T}$, i.e. $r(v) = v$: The label of v is its label from $\overline{T}$.

Case 2. Node $v \in T$ was on a chain that was contracted into a single node c in $\overline{T}$: The label of v is the label of c from $\overline{T}$ plus the index of v in the chain.

Case 3. Node v was pruned: The label of v is the concatenation of the following three items. (i) the label of $r(v)$ in $\overline{T}$ (ii) the index of v, (iii) The label of v in its corresponding pruned forest.

It was shown in [2] that all the conditions of Theorem 5.4 can be easily tested based on the labels, and in [5] that the label length when the pruning parameter $x = \frac{1}{2}$ is bounded by $\frac{3}{2} \log n$.

5.3 The new mixed two-level scheme

Our mixed two-level scheme works similarly to the basic two-level scheme of Abiteboul, Kaplan, and Milo [2], labeling $\overline{T}$ and the pruned subtrees separately, with the final label of a pruned vertex obtained by a combination of its own label and the label of its representative in $\overline{T}$. The main difference is that in our new scheme the pruned forests are labeled using the compressed prefix scheme rather than the interval scheme. There are however few subtle technical issues that we need to address before we can present the exact label structure.

The compressed prefix labeling scheme has been defined in Section 4 for a single tree. Here, however, we need to apply it to a forest. We extend the compressed prefix labeling scheme to forests as follows.

1. Label each tree in the forest using the compressed prefix labeling scheme as we have defined it for trees.
2. Assign a binary string to each tree such that this collection of binary strings is prefix free.
3. The label of a node v in a tree T_i of the forest consists from a *code* and a *separator* as in Section 4.2. Here the string *code*(v) is the concatenation of the binary string assigned to T_i with the code of v in the labeling of T_i. The separator of v is simply the separator of v from the labeling of T_i.

[11] We point out that the interval scheme works in exactly the same way on forests. All one has to do is to number consecutively all the leaves of the forest and label internal nodes as before.

To conclude we need to explain how the prefix free strings are assigned to the trees $T_1, \ldots, T_k$ in each pruned forest. This is done according to one of the following two cases.

1. If the forest is not associated with a contracted chain then we consider the weights $w_1, \ldots, w_k$, where w_i is the number of nodes in T_i, and apply the H-max algorithm to this set of weights. The string b_i associated by H-max with w_i is the string which we assign to T_i.
2. In case the forest is associated with a contracted chain we assume that $T_1, \ldots, T_k$ are totally ordered such that if the parent of T_i is an ancestor of the parent of T_j on the chain then $i < j$. We assign the binary strings such that the string which corresponds to T_i is smaller than the string that corresponds to T_j for every $i < j$. We do that using one of the algorithms mentioned in Section 4.3.1 for assigning prefix free binary strings b_i to a totally ordered set of weights in a way consistent with the order such that $\max(w_i + |b_i|)$ is minimized.

The benefit from insisting on assigning the strings lexicographically when the forest is hanging off a contracted chain is that these strings can now play the role of the indices that we used in the two-level interval scheme. We use this observation as follows. We associate with each node v on a contracted chain the string assigned to the first subtree hanging off it in the total order $T_1, \ldots, T_k$ imposed on the forest. We define that string to be the *index* of v and extend the definition of index to pruned nodes as in Section 5.2. (Note that in Section 5.2 the indices were defined independently of the labeling of the pruned forest.) As in Section 5.2 we incorporate the index of each contracted node in its label. This modification, does not reduce the lengths of labels of the contracted nodes but it reduces the lengths of the labels of nodes in pruned subtrees. In the two-level interval scheme a label of pruned node also included the index of the node. Here the index of a pruned node is already incorporated implicitly in the label assigned to it by the prefix labeling of the forest containing it. This is formulated precisely in the following theorem which is analogous to Theorem 5.4. The proof is straightforward and hence omitted.

Theorem 5.5 *Vertex v is an ancestor of vertex w in T iff one of the following conditions holds.*

1. *Vertices v and w are both pruned and have a common representative leaf in $\overline{T}$, (so v and w belong to the same pruned forest), and v is an ancestor of w in that forest.*
2. *Vertices v and w belong to a chain that has been contracted into a single node of $\overline{T}$, and the index of v is smaller than the index of w.*
3. *Vertex $r(v)$ is an ancestor of $r(w)$ in $\overline{T}$. If v belongs to a contracted chain and w belongs to the pruned forest of that chain, we also require that the index of v is smaller (lexicographically) than the label of w in its pruned subtree.*

It follows that we can use the following labeling of the nodes. As before we define our labels according to the type of the node. Note that Cases 1 and 2 are essentially the same as in the two-level interval scheme, while in Case 3 the index component is now omitted.

Case 1. Node v is in $\overline{T}$, i.e. $r(v) = v$: The label of v is its interval label from $\overline{T}$.

Case 2. Node $v \in T$ was on a chain that has been contracted into a single node c in $\overline{T}$: The label of v is the interval label of c from $\overline{T}$ plus the index of v in c.

Case 3. Node v was pruned: The label of v is the concatenation of the following two items. (i) the interval label of $r(v)$ in $\overline{T}$ and (ii) the label of v from the prefix labeling of its pruned forest.

We answer ancestor queries based on these labels as implied by Theorem 5.5. The following theorem gives an upper bound on the length of the labels produces by the mixed two-level scheme. It follows from Theorem 4.5 by choosing the pruning parameter $x = \frac{1}{2}$ when we partition the tree. We provide a proof in the Appendix.

Theorem 5.6 *The length of the labels produced by the mixed two-level scheme is at most* $\frac{3}{2}\log n + O(\log\log n)$ *if we use* $x = \frac{1}{2}$ *in the tree partitioning algorithm.*

6 Experimental results

In this section we describe our experimental results. Section 6.1 compares the performance of the algorithms on real XML files. Section 6.2 compares the performance of the algorithms on two different types of random trees.

We start with a brief overview of the various physical representations tested in the experiments. We considered several variants of variable-length and fixed-length physical representations. In variable length representation, different labels can have different lengths. Each label then essentially consists of two parts, a fixed length prefix specifying the overall length of the label (in bits), followed by the label value itself.

For fixed-length representation we considered two options: fixed-per-index and fixed-per-file. Fixed-per-index here means that all the labels in the index are allocated the same length, determined by the maximum length of a label in the index. In contrast, in fixed-per-file all the labels in a particular file are given the same length, as determined by the longest label in that file (labels of different files may have different lengths). To parse a label we now need to know to which type of file it belongs. Recall that the actual identification of a node in the index consists of the file id plus the node label in the file. One possibility is to incorporate into the file id information about the length of its labels. Alternatively, in a distributed environment, we can group the files based on their maximum label length and assign different set of machines to manage each group. Then, all labels stored on a single machine have the same length.

To avoid dealing with complex bit manipulations, some systems may choose to work at the byte boundary. In this case the length of the label is then rounded up to 8 bits (1 byte), 16 bits (2 bytes), etc. Therefore, we compared the performance of different algorithms once assuming that a label can occupy any number of bits, and once assuming each label is rounded up to a byte boundary.

6.1 XML data

The performance of the prefix labeling schemes on real XML data was better than their worst case guarantees. The typical XML file is relatively balanced and does not have the pathological structure required to make our algorithms achieve their worst case bounds. We have tested the algorithms on two independent sets of XML files picked randomly among the files collected by the XML Web-crawler of [36]. The first set contained 1822 files, with an average of 526 nodes per file and has been collected by the crawler on 22/5/2000. The second set contained 509 files, with an average of 338 nodes per file, and was collected at 12/10/2000. We ran the experiments twice, once counting the storage for labeling the document items only, and once counting both the storage for labeling the document items, and the storage for labeling the words they contain. When we label the words, and use labels of variable length we ran the weighted version of the algorithms as described in Section 3.4.

At the bottom line, prefix labeling schemes, even simple ones, perform about 30% better than traditional interval labeling on these sets of files. Notice that the average file size in both sets of files was relatively small. We expect that on collections with larger files our algorithms will perform even better, as the contributions of the small additive constants that they incorporate would become more negligible.

We also compared the overall space consumption of the best labeling scheme for each physical representation. As intuitively expected the largest index is obtained using fixed-per-index representation and rounding the labels to byte boundaries. The smallest index is obtained using bit based variable-length representation. The gap between the worst and best representations is about 40%. The second best representations are the bit based fixed-per-file and the byte based variable length representations. They both about only 5% off from the variable bit representation. See Table 1.

Despite careful tuning of our implementations of the two-level schemes to avoid large constant factors, our experiments show that these schemes perform worse than the compressed prefix scheme and even the simple prefix scheme.

Type of representation #	Best scheme	Index size in bytes
Variable length bit	2L-D	2041418
Variable length byte	2L-D	2160848
Fix length bit	C-T-MSL	2104175
Fix length byte	C-T-MSL	2556771
Fix length for all files bit	C-T-MSL	2878326
Fix length for all files byte	C-T-MSL	2878326

Table 1: The size of the index of the best labeling scheme for each possible physical representation.

6.1.1 Variable length encoding

Here we allow different labels (even of nodes in the same tree) to have different lengths. The algorithms participating in the experiment were the interval scheme, all our prefix labeling schemes tuned to minimize the average length of a label, and the two-level schemes described in Section 5. Below we describe the details of each of our implementations. Furthermore, next to each algorithm, we give (in parenthesis) a short name that will be used later to refer to it in the diagrams presenting the results.

Interval scheme (I).

This is an implementation of the algorithm described in Section 1. The field structure of each label is shown in Table 2.[12]

Field #	length in bits	Field description
1	1	A type bit which is one if and only if the node is a leaf
2	4	The length of fields 3 and 4
3	according to field 2	The leaf-label if field 1 is on and the left endpoint of the corresponding interval if bit 1 is off
4	according to field 2	Exists only if bit 1 is off and then it contains the right endpoint of the corresponding interval

Table 2: The label structure for the interval scheme in variable length representation.

Minimum Sum Labeling (MSL).

[12]The maximum number of leaves in any file was no greater than 2^{16} so we need at most 16 bits for each leaf-label, and 4 bits to specify the length of a leaf label.

This is an implementation of the simple prefix labeling scheme described in Section 3.2. The field structure of each label is shown in Table 3.[13]

Field #	length in bits	Field description
1	5	The length of field 2
2	according to field 1	The label of the node as described in Section 3.2

Table 3: The label structure for the MSL scheme in variable length representation.

Compressed Total Ordered MSL (C-T-OMSL). This is an implementation of the algorithm described in Section 4.3.1. As mentioned above, an analysis of the XML trees showed that they were fairly balanced, hence most of the paths in the path decomposition degenerate to contain a single node. Note that in such paths, the partial order $\prec$ on the children of the corresponding contracted node is empty and the *separator* is not needed (since there are no path nodes that need to be separated). To exploit this observation we implemented the following improvement.
(i) When labeling the edges outgoing of such nodes we did not impose a total order on the children and simply used Huffman's algorithm (see Section 3.1) to label the outgoing edges.
(ii) We omitted the separator part of the label of nodes on degenerate paths. From the definition of the algorithm follows that this does not hurt our ability to answer ancestor queries.
The field structure of each label is shown in Table 4.

Field #	length in bits	Field description
1	1	A type bit which is set if and only if the node is a path of length one in the path decomposition
2	5	The length of field 3
3	according to field 2	The contents of the third field depends on whether the type bit is set or not. If the type bit is set then the third field contains the code part of the label of the node. Otherwise, the third field is further partitioned into two subfields. The first subfield contains the code part of the label concatenated with the separator part of the label. The second field contains the index of the first bit of the separator in the first field. The lengths of each of these two subfields is determined as follows. If the length of the third field is b bits then the length of the second subfield is the smallest x such that $2^x \geq (b-x)$.

Table 4: The label structure for the C-T-OMSL scheme in variable length representation.

Compressed Partial Ordered MSL (C-P-OMSL). The implementation follows the lines of the high level description of the algorithm given in Section 4.3.1. (simpler than the detailed algorithm in the proof of Theorem 4.5.) It works as follows. If $p \in \hat{T}$ is a node that represents a path containing a singleton vertex v the algorithm uses Huffman's algorithm to assign strings to the edges outgoing from p as C-T-OMSL. Otherwise p represents a path of length $m \geq 2$. Let $q_1, \ldots, q_m$ be the children of p in $\hat{T}$, and let $w_i = size_T(v_i)$ where v_i is the node closest to the root on q_i. Let k be maximum such that $\sum_{j=1}^{k} w_j \leq \sum_{i=k+1}^{m} w_i$. By the definition of our path decomposition process $v_{k+1}, \ldots, v_m$ are all children of the last node on p and therefore are unrelated in $\prec_p$. We impose an arbitrary

[13]In all files the maximum length of a label was smaller than 2^{32} so we need 5 bits to specify the length of a label.

total order consistent with $\prec_p$ on $w_1, \ldots, w_k$ and compute an optimal tree T_1 that minimizes the sum of lengths of the paths and is consistent with the total order we chose. We apply Huffman's algorithm to obtain a tree T_2 with leaves $w_{k+1}, \ldots, v_n$. Finally we create a new tree T' with T_1 as its left subtree and T_2 as its right subtree. We use T to assign binary strings to the edges outgoing of p. As for C-T-OMSL we omit the separator part of the label of nodes on degenerate paths (paths of length one). The field structure of the labels is identical to that in C-T-OMSL, as shown in Table 4.

Two-level interval scheme (2L-I). This is an implementation of the two-level algorithm described at [2] with the idea of using lexicographic indices of [5] as we described in Section 5.2. Figure 8 shows the structure of the labels for the different types of nodes.

The first field is of fixed length and it specifies the length of the label. The second field consists of two bits and it classifies the labels into four types. The first type consists of labels of nodes at the top level tree, $\overline{T}$. The second type consists of labels of contracted nodes. These labels contain the lexicographic index of the node in its chain, in addition to its interval in $\overline{T}$. The third and the forth types consist of labels of pruned nodes. Using one additional bit we distinguish between four types of pruned nodes. We classify a pruned node according to whether it is a leaf, and according to whether it is pruned off a contracted chain. All four subtypes of labels contain a leaf label of the representative in $\overline{T}$. Nodes that are pruned off chain contain their lexicographic index in the chain concatenated to their interval label inside the pruned subtree.

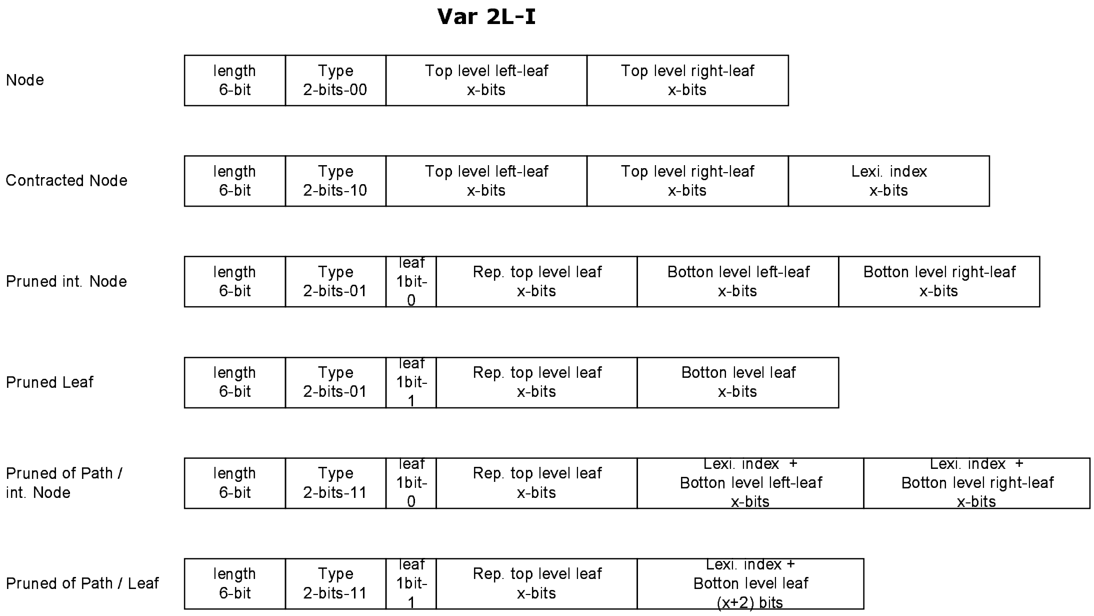


Figure 8: The description of the fields of the labels in the 2L-I scheme. The first field determines the length of the label by storing a value x which specifies the lengths of all variable length fields.

Two-level Mixed scheme (2L-M). This is an implementation of the mixed two-level scheme described in Section 5.3. Figure 9 shows the structure of the labels for the different types of nodes that arise when we decompose the tree.

For all types of nodes the first field (of fixed length of 6 bits) specifies the length of the label. The second field consists of a single bit which is one if and only if this is a label of a pruned node.

Top level nodes are further classified into nodes in $\overline{T}$ and contracted nodes by an additional bit, the *contracted node bit* (CN bit). If this bit is one then the label is of a node on a contracted chain so in addition to its interval label in $\overline{T}$ (fields 3 and 4) it contains a lexicographic index in its last field. The label of a pruned node v further contains the label of its representative leaf (third field), and the concatenation of the label of the pruned tree containing v with the label of v inside this tree (last field). Recall that we labeled the trees as described in Section 5.3, and we labeled nodes inside the trees using our C-T-MML implementation. Note that in contrast with the format of the labels in the 2L-I scheme here we do not distinguish between different types of pruned nodes.

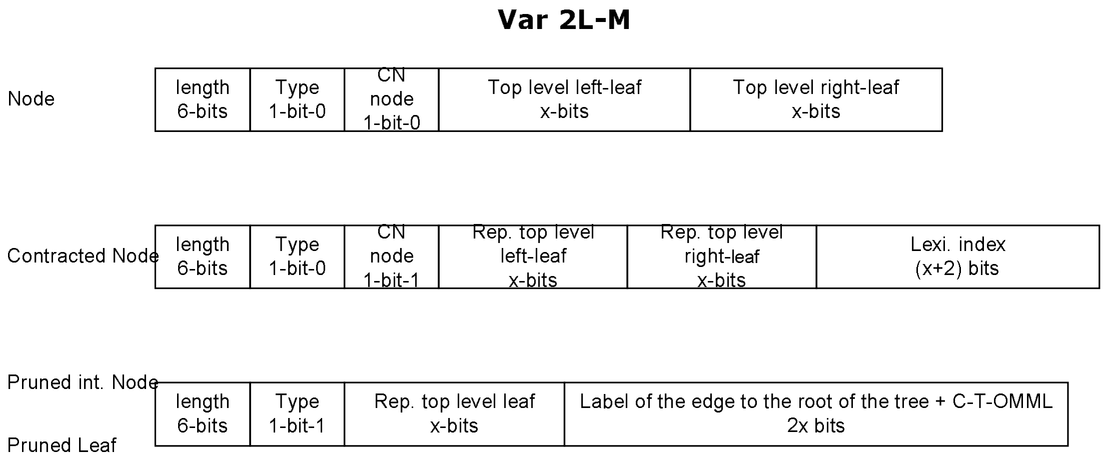


Figure 9: The description of the fields of the labels in the 2L-M scheme. The first field determines the length of the label by storing a value x which specifies the lengths of all variable length fields.

Degenerate two-level scheme (2L-D). This variant exploits the fact that in our XML data a leaf is typically the only child of its parent. We prune every leaf that is a single child of its parent and encode the remaining part of the tree using the interval scheme. We then labeled each pruned leaf with the label of its parents, and an extra bit to distinguish between the label of the pruned leaf and that of its parent.

The label is structured as the label of the interval scheme with an additional type bit to distinguish between the label of a pruned leaf and the label of its parent.

The results. Our results for the first data set are shown in Figure 10(a). The results for the second data set were similar and are not shown. For each algorithm we computed the total memory required to store the labels for all trees in the data set, then we normalized those numbers w.r.t to the space consumed by the interval scheme. We see that for variable length physical representation all the prefix-based labeling schemes improve upon the interval scheme.

The degenerate two-level scheme (2L-D) obtains the most compact labels. The reason is that among the nodes of the XML trees in our data set about 40% are leaves, and furthermore most of them are single children of their parents. Thus 2L-D labels about 80% of the nodes on average with approximately the same number of bits as the label of a leaf. Recall that in the interval scheme the length of the label of a leaf is about half the length of the label of an internal node.

We also see that compressed prefix labeling schemes are very effective in reducing the length of the labels. They shrink the label length by 20% on the average compared to the interval scheme. There was no significant difference when using an arbitrary total order (C-T-OMSL) compared to the case where we tried to exploit the special structure of the partial order (C-P-OMSL) at each node of $\hat{T}$.

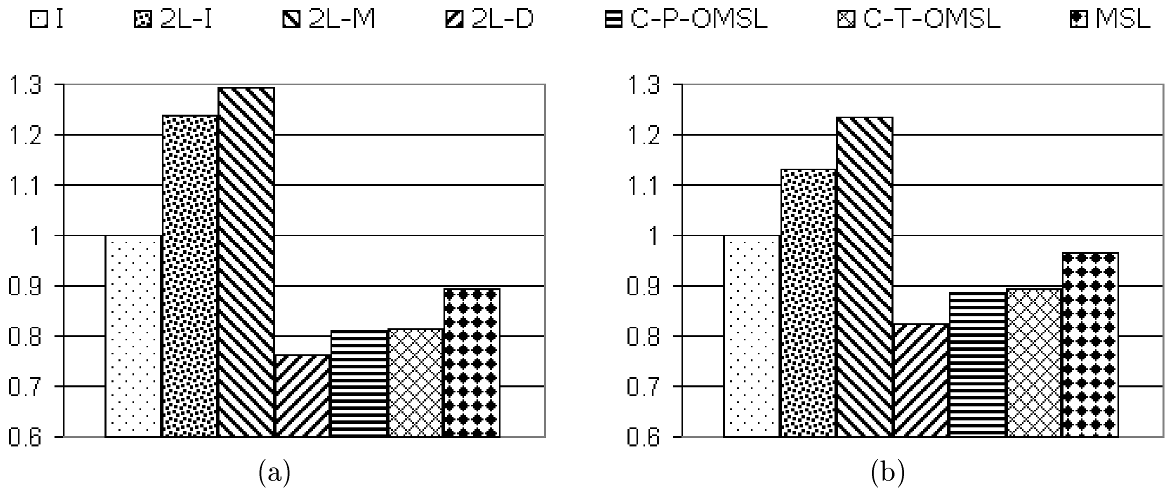


Figure 10: (a) Labels have different lengths. (b) Labels have different length and nodes have different weights.

Rounding to byte boundaries. We repeated the experiment when each label is forced to occupy an integral number bytes. In this setup each label starts with two bits that specify the length of the label in bytes (which was always between one and four for our data sets). The structure of the rest of the label for each algorithm was similar to what we described before. The only difference is that we do not store with the label the length in bits of any field. Since we know the length of the label in bytes we can parse it assuming some fixed proportion between variable length fields. When the number of bits is not a multiple of the number of fields we assume extra bits are allocated to the first field. The results stayed almost the same as without rounding and therefore are not shown.

We also considered an alternative physical layout where instead of using two bits to specify the byte length of the label we use the last bit of every byte to mark whether this byte is the last byte of the label. Using this alternative layout all algorithms performed slightly worse. This alternative layout, however, has the advantage that it does not impose any upper bound on the maximum size of a tree that we can encode without changing the structure of the physical label.

Weighting the trees. We also repeated the experiment this time counting the number of words in each leaf of the XML file and labeling these words in addition to the node itself. Accordingly, we changed the algorithms solving the MSL and OMSL problems to use these word counts as weights (as described in Section 3.4). The results for the first data set are shown in Figure 10(b). The results for the second set are similar. The relative decrease in the label length of most algorithms is the same as in the unweighted case. The compressed prefix schemes are about 10% better than the interval scheme. The 2L-D algorithm performs best and shrinks the sum of the labels by a little bit less then 20%.

6.1.2 Fixed length encoding - per file

In this experiment all labels of a given tree had to be of the same length. We assume that this length is known when accessing the label and need not be stored with the label. We compared algorithms analogous to the algorithms of Section 6.1.1 but tailored to minimize the maximum length of a label. We used MML rather than MSL as the simple prefix scheme (See Section 3.3). Instead of C-T-OMSL and C-P-OMSL we implemented analogous variants, denoted by C-T-OMML and C-P-OMML respectively. These algorithms solve the OMML problem rather than the OMSL problem on the compressed tree (See Section 4.3.1). The interval scheme and the two-level schemes (I, 2L-D, 2L-M, and 2L-I) are as in Section 6.1.1. Below we describe the changes in the structure of the labels produced by each implementation. Furthermore for each implementation we give (in parenthesis) a short name that will be used later to identify it in the diagrams presenting the results.

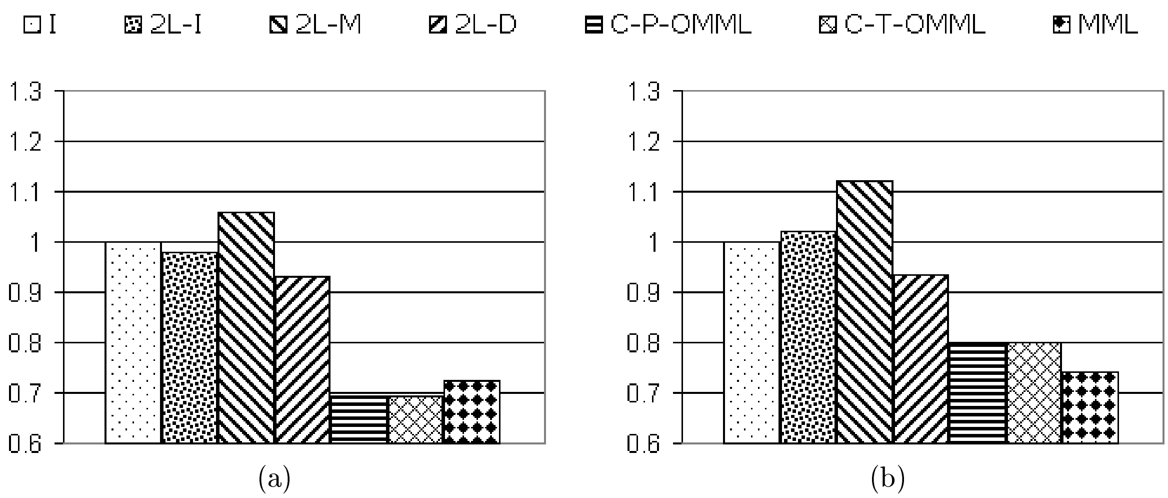


Figure 11: Fixed length (per file). (a) Results for the first data set. (b) Results for the second data set.

Interval scheme (I). Here we encode the two endpoints of the interval in two fields of fixed size capable of storing the largest index of a leaf in the file. For leaves the two fields store the same value.

Minimize the maximum label length (MML),
Compressed Total Ordered MML (C-T-OMML),
Compressed Partial Ordered MML (C-P-OMML). For these implementations we store the padded version of the node's label in a field large enough to store the largest label in the tree. The structure of the labels is as the one we used for MSL, C-T-OMSL, and C-P-OMSL, respectively.

Two-level interval scheme (2L-I). The structure of the labels for this implementation is shown in Figure 12. This structure is similar to the structure of the labels in variable length encoding (see Figure 8. Note that here we do not distinguish between pruned leaves and pruned internal nodes. Pruned leaves have the same value in their left-leaf and right-leaf fields.

Two-level Mixed scheme (2L-M). This is an implementation of the mixed two-level scheme described in Section 5.3. The structure of the labels is shown in Figure 13. This structure is similar to the structure of the labels in variable length encoding (see Figure 9).

Degenerate two-level scheme (2L-D). This is a variant of the interval scheme defined in Section 6.1.1. The label consists of two fixed length fields holding the endpoints of the interval. For a pruned node one of the two field is zero and for a leaf in the tree remaining after pruning the two fields are identical.

The results. Our results for the two data sets mentioned above are shown in Figure 11. The algorithms that perform best on the two data sets are the simple and compressed prefix labeling

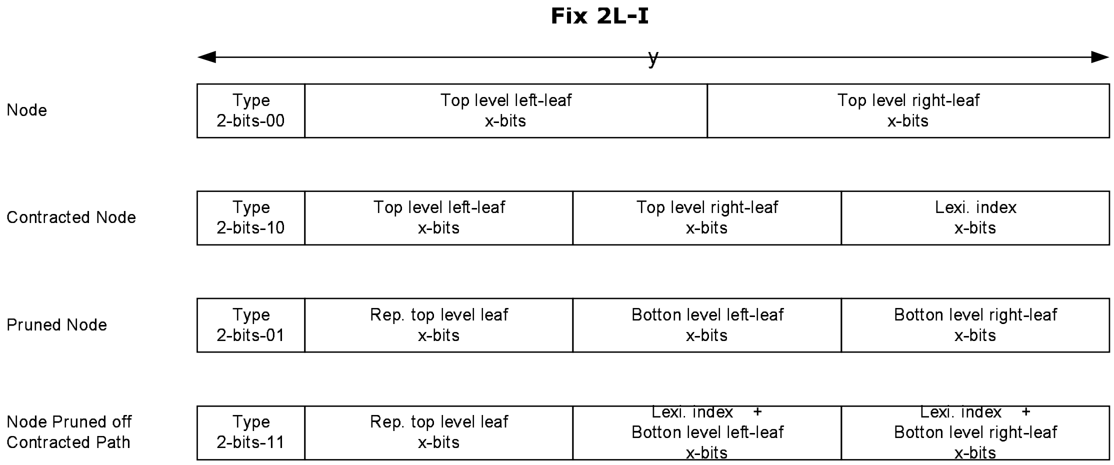


Figure 12: The structure of the labels of the 2L-I scheme in fixed length encoding. The fixed length is denoted by y and the length of each field is a fixed function of y.

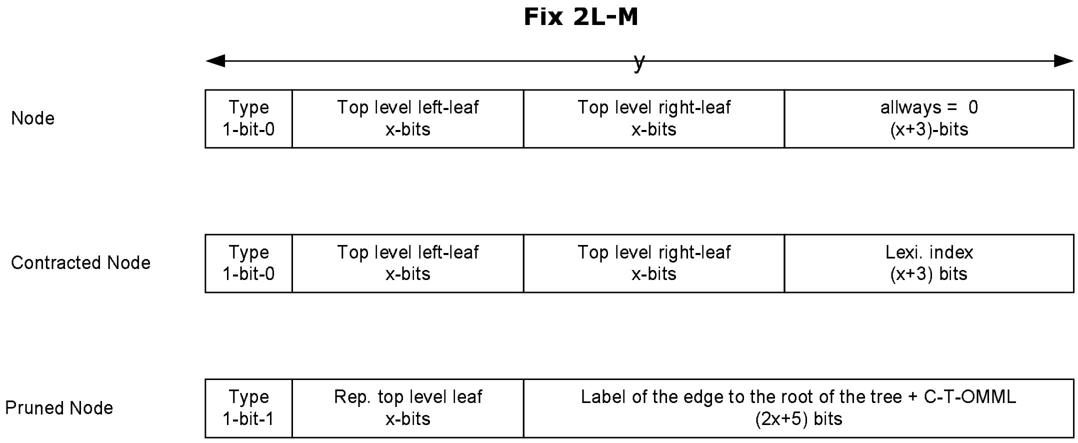


Figure 13: Fix Length 2L-M nodes fields description

schemes (MML, C-T-OMML, C-P-OMML). The compressed prefix schemes outperform slightly the simple prefix scheme on the first data set, while it is the other way around on the second data set. The reason is probably the larger average tree size in the first data set: As the file gets larger the compression is more likely to have some balancing effect on the tree, which pays for the overhead of a more complicated label structure. Overall we see that prefix labeling is about 30% more compact than the conventional interval scheme. Our experiments show that the two-level schemes perform worse than the compressed prefix scheme and even the simple prefix scheme perform better on real XML data. The more complicated label structure of these schemes did not pay off on the relatively small files that we labeled. Finally note that the 2L-D scheme, the winner of previous variable length experiment, is not performing so well for fixed length representation. This is because the maximum label length here is determined by the internal nodes of the tree (being longer than that of the leaves), hence the advantage of having many small leaf labels disappears.

As in the case of variable length encoding we repeated the experiment when each label is forced to occupy an integral number bytes. The results remain similar to what is shown in Figure 11. We also calculated the size of the index assuming each word in a leaf is being labeled separately (all with the same label).[14] This modification also did not significantly changed the results.

6.1.3 Fixed length for all files

In this experiment we compared the same algorithms as in Section 6.1.2 but required the labels in all the files be of the same length (as determined by the maximal length of a label in the whole index). The bit structure of the labels was the same as for the algorithms in Section 6.1.2 .

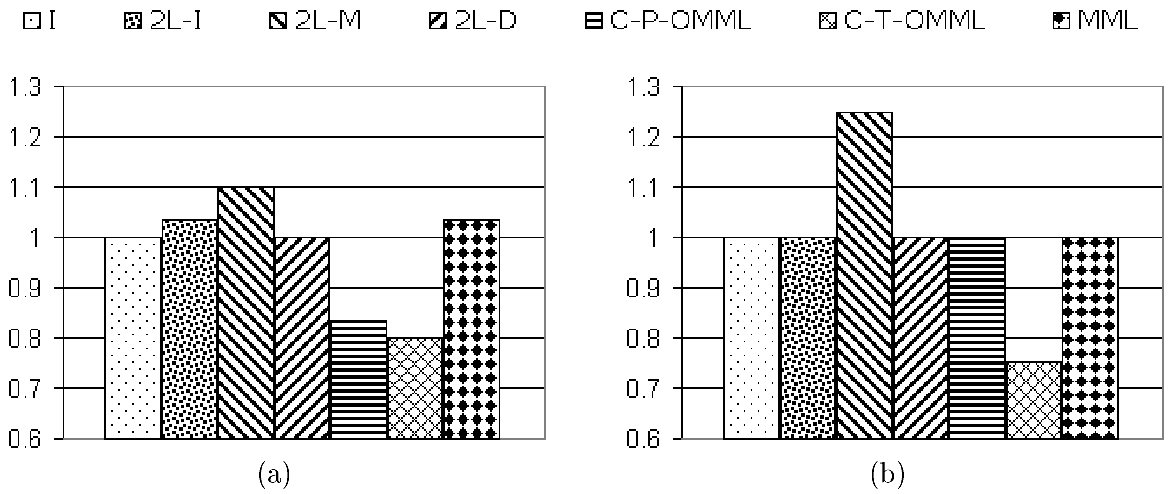


Figure 14: (a) Fixed Length for all files. (b) Fixed length for all files rounded to byte boundaries.

Our results for the first data set are shown in Figure 14(a). The results for the second data set were similar and are thus omitted. The compressed prefix schemes reduce the index size by about 20%. The performance of the two-level algorithms is slightly worse than the performance of the interval scheme. Here we see a large gap between the performance of C-T-OMML or C-P-OMML, and the performance of MML. This gap is due to the fact that in this experiment the

[14]In contrast with variable length encoding where algorithms may have calculated different labels when nodes had different weights, here the longest label determines the lengths of all labels so there is no point for the algorithms to try and shorten labels of nodes with large weight.

worst tree in the whole database determines the size of the label. The worst case guarantees of C-T-OMML and C-P-OMML allowed them to perform reasonably well even on the most skewed tree in the database, whereas MML performs worse on that tree. When we round the labels up to byte boundaries (Figure 14(b)) C-P-OMML looses its advantage. The bit length of its label is not small enough to require a byte less than the other algorithms. On the other hand, the algorithm C-T-OMML manage to get away with labels shorter by a byte.

6.2 Random trees

In order to understand the two-level algorithms better, evaluate the additive constant of these two-level schemes, and check how large need a tree to be for these schemes to win. We compared these algorithms on two distributions of random trees. The first are trees drawn uniformly at random from the set of ordered, unlabeled, full[15] binary trees with n vertices (for varying values of n) [16]. The second are trees drawn uniformly at random from the set of unordered labeled trees with n vertices [34] and no apriori bound on the degree (Our algorithms of course did not use the labels or the order of the nodes.).

Using variable length encoding we compared the implementations of the interval scheme (I), the compressed partial ordered MSL scheme (C-P-OMSL), the two-level interval scheme (2L-I), and the two-level mixed scheme (2L-M), all described in Section 6.1.1. Using fixed length encoding we used the same implementations except that we replaced the compressed partial ordered MSL scheme with the compressed partial ordered MML (C-P-OMML) scheme described in Section 6.1.2.

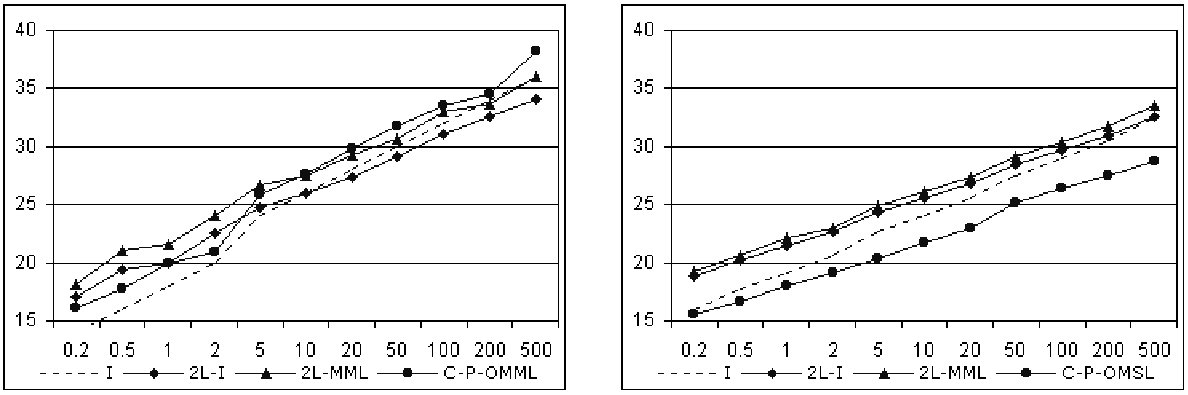
Our results for the random binary trees are shown in Figure 15, and the results for the random labeled trees are shown in Figure 16. We can see that the two-level interval scheme performs slightly better than the two-level mixed scheme on these distributions of trees. The additive constant of the two-level interval scheme with respect to $\frac{3}{2}\log n$ is about 7. For fixed length representation, on the random binary trees the two-level interval scheme becomes competitive when the tree has around 12K nodes. On the random labeled tree the two-level interval scheme becomes competitive on trees with about 25K nodes. For variable length representation the trees have to be even larger before the two-level schemes become competitive.

In contrast with the XML files the random trees are often quite deep (the random labeled trees with about 200K nodes that we generated had depths ranging from 680 to 2030). Therefore prefix schemes without compression necessarily perform bad and did not participate in the experiment. The performance of the compressed prefix schemes, however, is more interesting. For both families of random trees the appropriate compressed prefix scheme perform better than the interval scheme when using variable length representation but perform worse than the interval scheme when using fixed length representation. In particular this behavior holds when the average size of the random tree is the same as the average size of the XML files. Thus we conclude that at least with respect to fixed length representation the good performance of the compressed prefix scheme is due to the particular structure of the XML files.

7 Conclusion

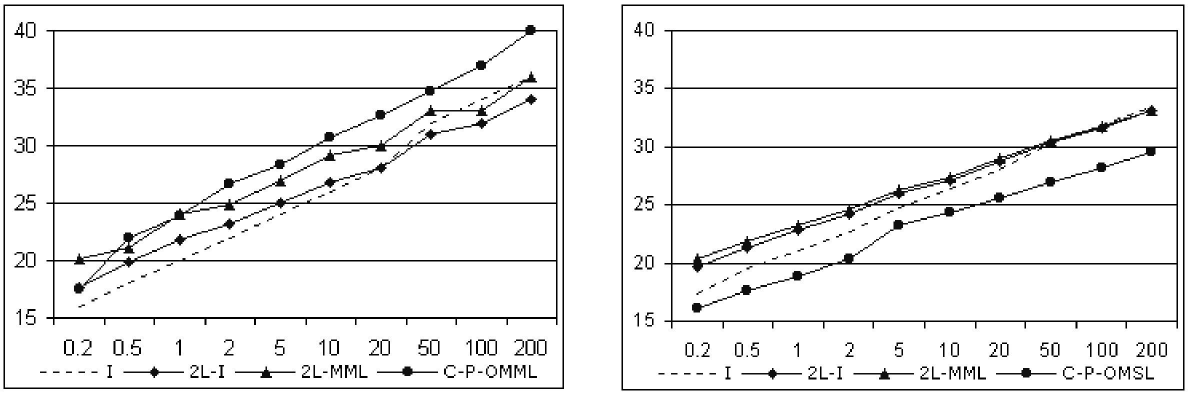
Motivated by the design of a compact index for massive amounts of XML data, we study in this work the problem of giving short labels to the nodes of XML trees such that, given the labels of two nodes, one can determine whether one node is the ancestor of the other. In contrast to the traditional interval labeling scheme, we introduce a prefix-based approach, demonstrating its

[15] Each node is either a leaf or has both a left child and a right child



(a) Fix Length (b) Variable Length

Figure 15: Results for random binary trees. The x-axis shows the size of the tree in thousands of nodes, and the y-axis denotes the average bit length of a label. (The average is over several random binary trees of the same size and in case of variable length representation the bit length of a label within a file is also averaged.)



(a) Fix Length (b) Variable Length

Figure 16: Results for random labeled trees with arbitrary degrees. The x-axis shows the size of the tree in thousands of nodes, and the y-axis denotes the average bit length of a label. (The average is over several random binary trees of the same size, and in case of variable length representation the bit length of a label within a file is also averaged.)

adequacy to the labeling of XML data. On the theoretical side we present a new prefix-based labeling scheme with an upper bound of $\frac{3}{2}\log n + O(\text{loglog n})$ on the length of a label. On the empirical side, we demonstrate the compactness of the proposed prefix labeling schemes on real XML data. We show that the typical structure of XML trees makes the prefix-based approach particularly appealing. Its space consumption is not only much lower than the theoretical worst-case guarantees but also significantly smaller than that of previous algorithms, even those having the same worst-case bounds. The experiments also illustrated, empirically, the importance of the worst case guarantees, in particular when we use a fixed length representation where one bad file can cause the label length to be very large.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu, *Data on the web: From relations to semistructured data and XML*, Morgan-Kaufmann, 1999.

[2] S. Abiteboul, H. Kaplan, and T. Milo, *Compact labeling schemes for ancestor queries*, Proceedings of the Twelfth Annual Symposium on Discrete Algorithms (SODA), 2001, pp. 547–556.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener, *The Lorel query language for semistructured data*, International Journal on Digital Libraries **1** (1997), no. 1, 68–88.

[4] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe, *Nearest common ancestors: a survey and a new distributed algorithm*, Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), 2002, pp. 258–264.

[5] S. Alstrup and T. Rauhe, *Improved labeling scheme for ancestor queries*, Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA), 2002, pp. 947–953.

[6] Stephen Alstrup, Philip Bille, and Theis Rauhe, *Labeling schemes for small distances in trees*, Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2003, pp. 689–698.

[7] *Altavista.*, `http://www.altavista.com`.

[8] A. Barkan and H. Kaplan, *Partial alphabetic trees*, Algorithms - ESA 2002, 10th Annual European Symposium, 2002, pp. 114–125.

[9] Michael A. Bender and Martin Farach-Colton, *The LCA problem revisited*, LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, 2000, pp. 88–94.

[10] O. Berkman and U. Vishkin, *Recursive star-tree parallel data structure*, SIAM Journal of Computing, vol. 22, 1993, pp. 221–242.

[11] S. Brin and L. Page, *The anatomy of a large-scale hypertextual Web search engine*, Proc. 7th International World Wide Web Conference (WWW7)/Computer Networks **30** (1998), no. 1–7, 107–117.

[12] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu, *A query language and optimization techniques for unstructured data*, Proceedings of ACM-SIGMOD International Conference on Management of Data, 1996, pp. 505–516.

[13] C.Gavoille, D.Peleg, S.P'erennes, and R.Raz, *Distance labeling in graphs*, Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms (SODA), ACM Press, 2001, pp. 210–219.

[14] J. Chen, D. DeWitt, F. Tian, and Y Wang, *NiagraCQ: A scalable continuous query system for internet databases*, Proceedings of ACM-SIGMOD International Conference on Management of Data, 2000, pp. 379–390.

[15] R. Cole and R. Hariharan, *Dynamic lca queries on trees*, Proc. Tenth ACM-SIAM Symposium on Discrete Algorithms (SODA), 1999, pp. 235–244.

[16] Arnold D.B. and Sleep M.R., *Uniform random generation of balanced parenthesis strings*, ACM Trans. Program. Lang. Syst. **2** (1980), 122–128.

[17] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, *A query language for XML*, Computer Networks (Amsterdam, Netherlands: 1999) **31** (1999), no. 11–16, 1155–1169.

[18] D.Peleg, *Proximity-preserving labeling schemes and their applications*, In Proc 25th Int. Workshop on Graphs-Theoretic Concepts in Computer Science (WADS) (1999), 30–41.

[19] H. Gabow, *Data structures for weighted matching and nearest common ancestors with linking*, Proc. 1st ACM-SIAM Symposium on Discrete Algorithms (SODA), 1990, pp. 434–443.

[20] A. M. Garsia and M. L. Wachs, *A new algorithm for minimum cost binary trees*, SIAM J. Comput. **6** (1977), no. 4, 622–642.

[21] E. N. Gilbert and E. F. Moore, *Variable length binary encoding*, Bell systems technical journal **38** (1959), 933–968.

[22] D. Harel and R. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput. **13** (1984), no. 2, 338–355.

[23] T. C. Hu, D. J. Kleitman, and J. K. Tamaki, *Binary trees optimum under various criteria*, SIAM J. Appl. Math. **37** (1979), no. 2, 514–532.

[24] T. C. Hu and C. Tacker, *Optimum computer search trees*, SIAM J. Appl. Math. **21** (1971), 514–532.

[25] D. A. Huffman, *A method for the construction of minimum-redundancy codes*, Proc. IRE, 40, 1952, pp. 1098–1101.

[26] D. E. Knuth, *Optimum binary search trees*, Acta Infomatica **1** (1971), 14–25.

[27] K. Mehlhorn, *A best possible bound for the weighted path lenght of binary search trees*, SIAM J. Comput. **6** (1977), no. 2, 235–239.

[28] D. Peleg, *Informative labeling schemes for graphs*, Mathematical Foundations of Computer Science (MFCS), 2000, pp. 579–588.

[29] N. Santoro and R. Khatib, *Labeling and implicit routing in networds*, The Computer J. **28** (1985), 5–8.

[30] B. Schieber and U. Vishkin, *On finding lowest common ancestors: Simplification and parallelization*, SIAM J. Comput. **17** (1988), no. 6, 1253–1262.

[31] W3C, *Extensible markup language (xml) 1.0*, http://www.w3.org/TR/REC-xml.

[32] ———, *Extensible stylesheet language (xsl)*, http://www.w3.org/Style/XSL/.

[33] ———, *Xsl transformations (xslt) specification*, http://www.w3.org/TR/WD-xslt.

[34] Douglas B. West, *Introduction to graph theory*, 1996.

[35] *Xdex*, `http://www.xmlindex.com`.

[36] Xyleme, *A dynamic data warehouse for the xml data of the web*, http://www.xyleme.com.

APPENDIX

In this appendix we prove worst case bounds on the length of the labels produced by the algorithms for the OMML problem in Section 4, and an upper bound on the length of label of the mixed two-level scheme presented in Section 5.3. We first prove Theorem 4.4. We repeat the statement of this Theorem below but for other definition see Section 4.

Theorem 4.4 *Let $\hat{T}$ be a tree with a given partial order $\prec_p$ in each internal node p. Assume that we choose a total order, $\prec_p^t$ in each internal node p consistent with $\prec_p$, and then solve the OMML problem by labeling the edges outgoing from p using an optimal string assignment consistent with this total order. Then the length of the labels is bounded by $\log(\hat{n}) + 2d$, where d is the depth of $\hat{T}$, and $\hat{n}$ is the number of nodes in $\hat{T}$.*

For the proof we need the following lemma which is due to Mehlhorn [27].

Lemma A.1 (Ordered weights) [27] *Let $L = w_1, \ldots, w_k$ be an ordered list of weights. There is a linear time algorithm that finds a lexicographically ordered list of prefix free binary strings $b_1, \ldots, b_k$ such that for every $i = 1 \ldots k$, $|b_i| \le \log(W/w_i) + 2$, where $W = \sum_{i=1}^{k} w_i$.*

Proof. To prove the lemma we construct a binary tree and place each weight w_i at a leaf whose distance from the root is at most $\log(W/w_i)+2$. It is straightforward to construct the corresponding binary strings from this binary tree. We only sketch the algorithm which constructs the tree, it is easy to establish its correctness by induction.

The algorithm is recursive and uses the partial sums $s_j = \sum_{i=1}^{i=j-1} w_i + w_j/2$, $1 \le j \le k$. We define a recursive procedure $construct - tree(e, f, C, \ell)$ that places a node at depth $\ell - 1$ in the tree which is an ancestor of $w_e, \ldots, w_f$. The parameter C is maintained such that $C \le s_e$, and $s_f \le C + \frac{W}{2^{\ell-1}}$. The algorithm constructs the tree by calling $construct - tree(e = 1, f = k, C = 0, \ell = 1)$.

The procedure construct-tree works as follows. If $e+1 = f$ it produces the tree in Figure 17(a). Otherwise we split to one of the following three cases.

1. If $s_e \le C + \frac{W}{2^\ell}$ and $s_f \ge C + \frac{W}{2^\ell}$ then we find the index j such $s_{j-1} \ge C + \frac{W}{2^\ell}$ and $s_j \ge C + \frac{W}{2^\ell}$. If $j = e+1$ we produce the tree in Figure 17(b). If $j = f$ we produce the tree in Figure 17(c). Otherwise we produce the tree in Figure 17(d).

2. If $s_e \ge C + \frac{W}{2^\ell}$ then we produce the tree in Figure 17(b).

3. If $s_f \le C + \frac{W}{2^\ell}$ then we produce the tree in Figure 17(c).

□

Proof. (of Theorem 4.4) Consider an algorithm M that solves the OMML problem by labeling the edges outgoing of a node p as follows. Let $q_1, \ldots, q_k$ be the children of p. Define $w_i = size(q_i)$[16] and apply the algorithm of Lemma A.1 to $w_1, \ldots, w_k$ ordered by $\prec_p^t$. (Note that these are not the same weights that we use to solve the OMML problem rather the weights we use to solve the OMSL problem.)

Since we find an optimal solution to the OMML problem where $\prec_p$ is replaced by $\prec_p^t$ at each node p, the maximum length of a label that we produce is no larger than the maximum length of a label produced by algorithm M. Therefore it suffices to show that the length of a label produced by algorithm M is bounded by $\log \hat{n} + 2d$.

[16] In this proof $size(q_i)$ is defined assuming each node of $\hat{T}$ has weight 1.

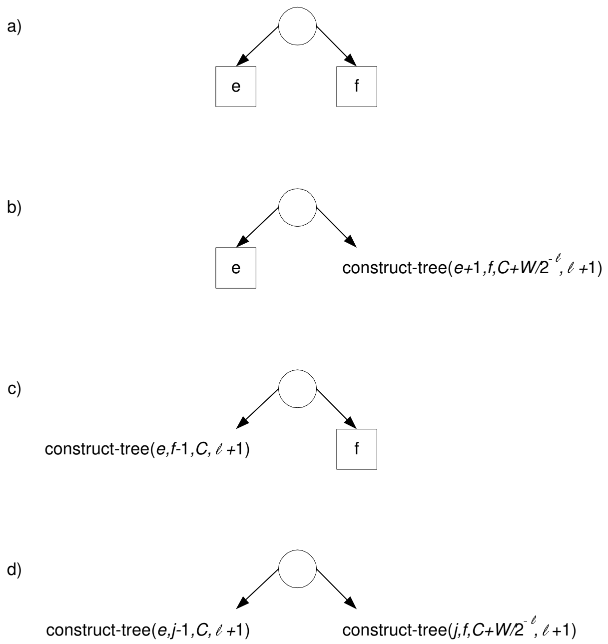


Figure 17: The cases of the algorithm in the proof of Lemma A.1

Since the labels of the leaves are the longest ones it suffices to prove that the label which M assigns to each leaf is bounded by $\log \hat{n} + 2d$. Let v_i be a leaf, and let $v_1, \ldots, v_{i-1}, v_i$ be the nodes on the path from the root to v_i (v_1 being the root). From Lemma A.1 and the definition of algorithm M we have that

$$|label(v_i)| \leq \Sigma_{j=1\ldots i-1}(\log(size(v_j)/size(v_{j+1})) + 2) = \log(size(v_1)/size(v_i)) + 2(i-1) \ .$$

Since v_1 is the root and v_i is a leaf, we have that $size(v_1) = \hat{n}$ and $size(v_i) = 1$. Also, i is bounded by d, the depth of the $\hat{T}$. It follows that $|label(v_i)| \leq \log(\hat{n}) + 2d$. □

Next we prove Theorem 4.5.

Theorem 4.5 *Let T be a tree, and let $\hat{T}$ be the compressed tree of T. There is an algorithm that finds an ordered prefix free assignment of $\hat{T}$ such that the length of the resulting labels of nodes in T is bounded by $2\log n$.*

We prove this theorem constructively by giving an algorithm that assigns the labels. Similarly to our second algorithm in Section 4.3.1 this algorithm finds an ordered prefix free assignment of $\hat{T}$ by exploiting the fact that the set of children of the last node on each path is heavy. As our previous algorithms this algorithm labels the edges outgoing from each node v independently and consistently with the partial order.[17]

Let v be an internal node in $\hat{T}$ and let $u_1, \ldots u_k$ be the children of v ordered consistently with $\prec_v$, and let $w_i = size_T(u_i)$ (the number of descendants of u_i in T). For technical reasons in our proof it would be convenient that each weight is smaller than half of the sum of all weights. Therefore we

[17]It handles the internal nodes in any order, i.e. we do not need to solve the problems in the nodes bottom up as our previous algorithms for OMML did.

add weights $w'_j = 1$, for every $1 \le j \le m$, where m is the length of the path corresponding to v. We define a partial order $\prec$ on the weights w_i, $1 \le i \le k$, and w'_j, $1 \le j \le m$ based on $\prec_v$ as follows. Recall that $\prec_v$ is defined by a partition of the children of v into sets $S_1, \ldots S_m$ such that for every $u \in S_i$ and $u' \in S_j$, $u \prec_v u'$ iff $i < j$. We partition the weights $w_1, \ldots w_k$ and $w'_1, \ldots, w'_m$ into sets $S^w_1, \ldots, S^w_m$ where S^w_i contains all the weights of the children in S_i and w'_i, and define $x \prec y$ if $x \in S^w_i$ and $y \in S^w_j$ for $i < j$. Since the sum of the weights w_i, $1 \le i \le k$, and w'_j, $1 \le j \le m$, is exactly $size(v)$, and by the definition of the path decomposition $size(u_i) < size(v)/2$, we obtain that each weight is no larger than half the sum of all the weights.

To label the edges outgoing from v we label the weights using the algorithm specified in the proof of Lemma A.2. The label we associate with the edge (v, u_i) is the label of the weight w_i.

Lemma A.2 (balanced partially ordered weights) *Let $w_1, \ldots, w_t$ be a list of weights, such that for all $i = 1 \ldots t$, $w_i \le W/2$, where $W = \sum_{i=1}^{t} w_i$. Let $\prec$ be a partial order relation on the weights defined by a partition of the weights to sets $S_1, \ldots, S_m$, where $x \prec y$ iff $x \in S_i$ and $y \in S_j$ for $i < j$. Furthermore assume that the sum of the weights in S_m is at least $W/2$. Then, there exists a list of prefix free binary strings $b_1, \ldots, b_n$ such that whenever $w_i \prec w_j$, b_i lexicographically precedes b_j, and furthermore for every $i = 1 \ldots t$, the appropriate case of the followings holds.*

1. *If $\frac{W}{3} < w_i$ then $|b_i| \le \log(W/w_i) + 1$.*
2. *If $\frac{W}{4} < w_i \le \frac{W}{3}$ then $|b_i| \le \log(W/w_i) + \log(3)$.*
3. *If $\frac{W}{6} < w_i \le \frac{W}{4}$ then $|b_i| \le \log(W/w_i) + 2$.*
4. *If $w_i \le \frac{W}{6}$ then $|b_i| \le \log(W/w_i) + \log(6)$.*

Before proving this lemma we prove Theorem 4.5.

Proof. (of Theorem 4.5) Since the labels assigned to edges outgoing of each node are prefix free and consistent with the partial order it is clear that we obtain an ordered prefix free labeling of $\hat{T}$. Last we prove that the length of the resulting labels of T is bounded by $2\log n$. To establish that we show that the length of a label of a leaf of $\hat{T}$ is at most $2 \log size_T(r(\hat{T})) = 2\log(n)$.

Let v_i be a leaf in $\hat{T}$, and let $v_0, \ldots, v_{i-1}, v_i$ be the nodes on the path from the root of $\hat{T}$ to v_i (v_0 being the root). From Lemma A.2 and the discussion preceding it follows that

$$|label(v_i)| \le \sum_{j=0\ldots i-1} \log(size(v_j)/size(v_{j+1})) + C_j \ , \tag{1}$$

where C_j is one of $1, \log(3), 2, \log(6)$, depending on the ratio between $size(v_j)$ and $size(v_{j+1})$. Note that the sum in Equation (1) telescopes, and $size(v_i) = 1$, so we obtain

$$|label(v_i)| \le \log(size(v_0)) - 1 + \sum_{j=0}^{i-1} C_j \ . \tag{2}$$

Thus, to conclude the proof we only need to show that $\Sigma_{j=0\ldots i-1} C_j \le \log(size(v_0))$.

Let a be the number of indices $0 \le j \le i-1$ such that $\frac{1}{3} < size(v_j)/size(v_{j+1}) \le \frac{1}{2}$, let b be the number of indices $0 \le j \le i-1$ such that $\frac{1}{4} < size(v_j)/size(v_{j+1}) \le \frac{1}{3}$, let c be the number of indices $0 \le j \le i-1$ such that $\frac{1}{6} < size(v_j)/size(v_{j+1}) \le \frac{1}{4}$, let d be the number of indices $0 \le j \le i-1$ such that $size(v_j)/size(v_{j+1}) \le \frac{1}{6}$. It follows from Lemma A.2 that $\sum_{j=0\ldots i-1} C_j \le a \times 1 + b \times \log(3) + c \times 2 + d \times \log(6)$, and it follows from the definition of a, b, c, d that $2^a \times 3^b \times 4^c \times 6^d \le size(v_0)$. These two inequalities imply that $\Sigma_{j=0\ldots i-1} C_j < \log(size(v_0))$. □

In order to prove Lemma A.2 we need the following lemma first.

Lemma A.3 (Unordered weights) *Let $w_1, \ldots, w_t$ be a set of weights and let $W = \Sigma_{i=1}^{t} w_i$. There exists a prefix free list of n binary strings $b_1, \ldots, b_t$, such that for every $i = 1 \ldots n$, $|b_i| \leq \log(W/w_i) + 1$.*

Proof. To prove the lemma we construct a binary tree and place each weight w_i at a node whose distance (in edges) from the root is at most $\log(W/w_i) + 1$. Once this is done we can easily give each weight a corresponding binary string by following the path from the root to the node while adding a 0 when going to a left child and adding a 1 when going to a right child.

The tree is constructed from the root to the leaves as follows. We start with a tree that contains a root node with two children (which are leaves at this point). We place every item of weight at least half the total weight in one of these leaves and if we still have items that have not been placed (this will always happen unless there are only two items with equal weights) we "expand" the unoccupied leaves. By "expanding" a node we mean making it an internal node and attaching two new children to it.

At stage i we have unoccupied leaves only at level i (we may have leaves containing items at lower levels). We place in each such leaf only items not yet placed and having weight greater than $\frac{W}{2^i}$, and expand the rest of the leaves.

If we associate weight of $\frac{W}{2^i}$ with each leaf at level i then a straightforward induction shows that at each level the sum of the weights of the unoccupied leaves is at least as large as the sum of the weights of the remaining items. Correctness of the algorithm easily follows from this inductive argument. □

We are now ready to prove Lemma A.2.

Proof. (of Lemma A.2) We construct a binary tree with the weights in its leaves where each weight satisfies the appropriate one of the following conditions.

Condition 1: A weight w_i such $\frac{W}{3} < w_i$ is at distance at most 2 from the root.

Condition 2: A weight w_i such that $\frac{W}{4} < w_i \leq \frac{W}{3}$ is at distance at most 3 from the root.

Condition 3: A weight w_i such that $\frac{W}{6} < w_i \leq \frac{W}{4}$ is at distance at most 4 from the root.

Condition 4: A weight w_i such that $w_i \leq \frac{W}{6}$ is at distance at most $\leq \log(W/w_i) + \log(6)$ from the root.

In addition the order of the leaves from left to right is consistent with the partial order on the weights. Once this is done it is straightforward to derive the desired binary strings from the leaves.

We first partition $w_1, \ldots, w_t$ into two lists denoted by L' and L'' such that the followings hold.

1. The list L' is sorted consistently with $\prec$.
2. The items in L'' are pairwise unrelated according to $\prec$.
3. If $a \in L'$ and $b \in L''$ are related then $a \prec b$.
4. The sum of the weights of the items in L' is no greater than $W/2$.

We partition $w_1, \ldots, w_t$ by first imposing a total order on them which is consistent with $\prec$ in which the items in S_m are ordered by increasing weights. To simplify notation we assume that the indices of w_i are such that w_i precedes w_j in this total order iff $i < j$. Let j be maximum such that $\sum_{i=1}^{j} w_i \leq W/2$. We define $L' = \{w_1, \ldots w_j\}$ and $L'' = \{w_{j+1}, \ldots, w_t\}$. This partition obviously

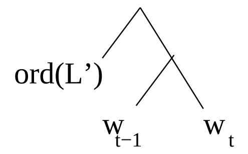


Figure 18: The tree constructed in Case 1

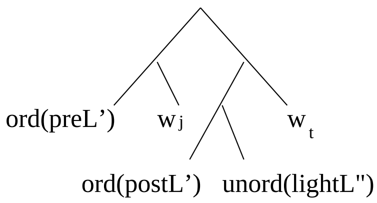


Figure 19: The tree constructed in Case 2a

satisfies requirements (1), (3), and (4) above. Since the sum of the weights of the last block, S_m, is at least half the total sum we also have that $L'' \subseteq s_m$, so (2) also holds. Let $W' = \sum_{i=1}^{j} w_i$ and let $W'' = \sum_{i=j+1}^{t} w_i$.

For an unordered list of weights L, we denote by $\text{unord}(L)$ the tree constructed by the algorithm in the proof of Lemma A.3. For a ordered list of weights L we denote by $\text{ord}(L)$ the tree constructed in the proof of Lemma A.1. We also use ℓ_i to denote the distance (in edges) of weight w_i from the root of the tree that we construct. We construct the tree according to one of the following cases.

Case 1: L'' contains two weights w_{t-1}, w_t where $w_{t-1}, w_t > W/3$.
By the definition of L'' it does not contain any other weights. We construct the tree as in Figure 18. The weights w_{t-1}, w_t clearly satisfy condition (1). Clearly $W' < W/3$ so $w_i \leq W/3$ for every $1 \leq i \leq t-2$. By Lemma A.1 and the structure of the tree in Figure 18 we obtain that $\ell_i \leq 1 + (\log(W'/w_i) + 2) \leq 3 + \log((W/3)/w_i) \leq \log(W/w_i) + \log(3)$ for every $1 \leq i \leq t-2$. So every w_i, for $1 \leq i \leq t$ satisfy the appropriate one among Conditions (2), (3), and (4).

Case 2: L'' contains a single weight w_t where $> W/3$. We consider here two sub-cases:
Subcase 2a: L' contains one weight $w_j > W/3$.
Let $lightL'' = L'' - \{w_t\}$, and let $preL'$ and $postL'$ be the sublists of weights in L' before and after w_j (w.r.t $\prec$), respectively. We construct the tree as in Figure 19. Condition (1) clearly holds for w_j and w_t. Since $W' \leq W/2$, and $w_j > W/3$ we have that $w(preL') < W/6$ and $w(postL') < W/6$, where $w(L)$ is the sum of the weights of the elements in L. For $w_i \in L' - \{w_j\}$, we have that $\ell_i \leq 3 + (\log(\max\{w(preL'), w(postL')\}/w_i) + 2) \leq 5 + \log((W/6)/w_i) \leq \log(W/w_i) + \log(6)$. Since for $w_i \in L' - \{w_j\}$, $w_i < W/6$ we have that they satisfy Condition (4).

Finally consider w_{t-1} the largest item in $lightL''$. If $w_{t-1} > W/6$ then by definition of L'', w_{t-1} is the only item in $lightL''$ and therefore it satisfies Condition (2). Otherwise w_{t-1} and therefore all the weights in $lightL''$ are no larger than $W/6$. This implies, by the definition of L'', that $W'' \leq W/2 + W/6$ and since $w_t > W/3$ we have that $w(lightL'') < W/3$. Summarizing we have that every weight $w_i \in lightL''$ is smaller than $W/6$ and $\ell_i \leq 3 + (\log(w(lightL'')/w_i) + 1) \leq 4 + \log((W/3)/w_i) \leq \log(W/w_i) + \log(6)$, so it satisfies Condition (4).
Subcase 2b: L' contains no weights $> W/3$.
Consider the tree in Figure 20(a). Condition (1) holds for w_t, and an argument identical to the one in Case (2a) shows that the appropriate condition holds for all weights in lightL".

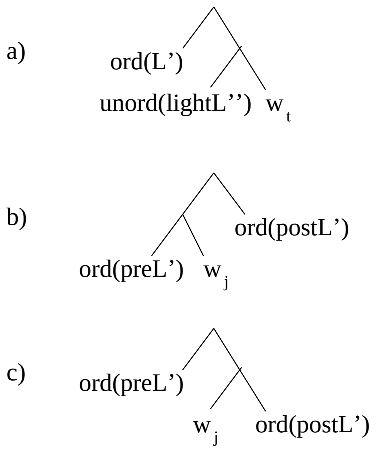


Figure 20: The tree constructed in Case 2b

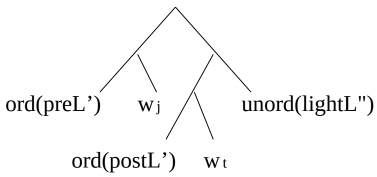


Figure 21: The tree constructed in Case 3a

Finally, consider $w_i \in L'$. For all the weights $w_i \in L'$ we have that $\ell_i \leq 1 + \log(W'/w_i) + 2 \leq 3 + \log((W/2)/w_i) = \log(W/w_i) + 2$. Therefore for every i such that $w_i \leq W/4$ either Condition (3) or Condition (4) holds. There can be at most one weight $w_j \in L'$ such that $w_j > W/4$. Assume that there is a weight $w_j > W/4$, let $preL'$ and $postL'$ be the sublists of weights in L' before and after w_j (w.r.t $\prec$). By inspecting the algorithm in the proof of Lemma A.1 we observe that if $w(preL') \leq w(postL')$ then it produces the tree in Figure 20(b) and otherwise it produces the tree in Figure 20(c). In either case w_j satisfies Condition (2).

Case 3: All weights in L'' are not greater than $W/3$. We consider two subcases:
Subcase 3a: L' contains one weight $w_j > W/3$.
Let w_t be the heaviest weight in L'' and let $lightL'' = L'' - \{w_t\}$. Let $preL'$ and $postL'$ be the sublists of weights in L' before and after w_j (w.r.t $\prec$). Consider the tree in Figure 21. It is clear that w_j satisfies Condition (1) and w_t satisfies the appropriate among Conditions (2), (3), or (4).

From the definition of L'' follows that $w(lightL'') < W/2$. Furthermore, it also follows from this definition that if $lightL''$ contains a weight greater than $W/4$ then this must be the only weight in $lightL''$. In this case this element is at distance 2 from the root and therefore satisfies Condition (2). Otherwise all the weights in $lightL''$ are smaller than $W/4$ and for each such weight w_i we have that $\ell_i \leq 2 + (\log(w(lightL'')/w_i) + 1) \leq 3 + \log((W/2)/w_i) \leq \log(W/w_i) + 2$ so either Condition (3) or Condition (4) holds for w_i. Since $W' \leq W/2$ and $w_j > W/3$ we have that $w(preL') < W/6$ and $w(postL') < W/6$. So by an argument identical to the one in Case (2a) we obtain that all these elements satisfy condition (4).
Subcase 3b: All weight in L' are no greater than $W/3$.
Consider the tree in Figure 22. For $w_i \in L'$ such that $w_i \leq W/4$ we have that $\ell_i \leq 1 + (\log(W'/w_i) + 2) \leq 2 + \log(W/w_i)$ so it satisfies Condition (3) or Condition (4). If there exists $w_j \in L'$ such that $w_j > W/4$ then as in Case 2 we observe that w_j would be at distance 2 from the root of ord(L')

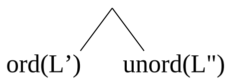


Figure 22: The tree constructed in Case 3b

and therefore would satisfy Condition (2).

Similarly, for $w_i \in L''$ such that $w_i \leq W/4$ we have that $\ell_i \leq 1 + (\log(W''/w_i) + 1) \leq 2 + \log(W/w_i)$. If there exists $w_j \in L''$ such that $w_j > W/4$ then by inspecting the algorithm in the proof of Lemma A.3 we observe that each such w_j will be placed at distance at most 2 from the root of unord(L) and therefore would satisfy Condition (2). $\Box$

Last we prove Theorem 5.6 from Section 5.3 that bounds the length of the labels produced by the mixed two-level scheme.

Theorem 5.6 *The length of the labels produced by the mixed two-level scheme is at most* $\frac{3}{2}\log n + O(\log\log n)$ *if we use* $x = \frac{1}{2}$ *in the tree partitioning algorithm.*

First we extend Theorem 4.5 to the case we use it to label a forest of trees rather than a single tree as described in Section 5.3.

Lemma A.4 *Let n be the size of the pruned forest. The maximum length of a label (*code *concatenated with a* separator*) obtained by applying the compressed prefix scheme to the forest is bounded by* $2\log n + 2$.

Proof. Let $v_1, \ldots, v_k$ be the nodes of a contracted chain, and let n_i denote the number of nodes in a subtree rooted at v_i in the forest pruned off of this chain. Recall that the label of a node in the subtree T_i rooted by v_i is the concatenation of the label of T_i in the lexicographic prefix labeling of its forest with the label of the node in the compressed prefix labeling of T_i. By Theorem 4.5 we have an upper bound of $2\log(n_i)$ on the length of a label in the compressed prefix labeling of T_i. From Lemma A.1 follows that the label assigned to T_i is of length at most $\log(n/n_i) + 2$. Summing this up we obtain that the length of a label of a node in T_i is bounded by $2\log(n_i) + \log(n/n_i) + 2 = \log(n_i) + \log(n) + 2 \leq 2\log(n) + 2$ $\Box$

Proof.[of Theorem 5.6] We use a type bit to distinguish between nodes of different types. It is zero if v is a node in $T \cap \overline{T}$, or a node on a contracted chain, and 1 if v is in a pruned subtree.

- When the type bit is zero it would follow by two fixed length fields. The first field contains an interval label of a node in $\overline{T}$. The second field contains a lexicographic index (which will be zero for nodes in $T \cap \overline{T}$). By Lemma 5.1 the length of the first field is $2 * \lceil\log(3 * n^{1-x})\rceil$ bits. Since by Lemma A.1 the size of a forest is at most $(4 * n^x)$[18], the length of the second field is $\lceil\log(4n^x) + 2\rceil$ bits. Summing up we obtain that the length of a label of nodes of type 1 and 2 is $(2 - x)\log n + O(1)$.
- When the type bit is 1 it is followed by two fixed length fields. The first field contains the label of a **leaf** node, $r(v)$, in $\overline{T}$ (for the interval labeling of $\overline{T}$), and the second contains the compressed prefix label of v in its pruned forest. For the first field we need $\lceil\log(3 * n^{1-x})\rceil$ as

[18] Recall that in the pruning phase we grouped the trees to forests each having at most $2\ell^x$ leaves, and all the internal nodes had at least two children.

it is a label of a leaf in $\overline{T}$ [19]. The second field contains the compressed prefix label of the node in its forest. This label consists of the concatenation of a *code* and *separator*, and an index of where the code ends and the separator starts. Since the size of the forest is at most $4n^x$, we obtain by Lemma A.4 that the length of the code concatenated with the separator is at most $2\log(4n^x)+2$. Adding the length of the index of where the code ends and the separator starts we obtain that the length of the second field is is at most $2\log(4n^x) + O(\log\log n)$. Summing up we obtain that the total length of a label of a node of type 3 is $(1+x)\log n + O(\log\log n)$

By choosing $x = 1/2$ we balance out the lengths of the different types of labels and obtain that the maximum length of a label is $\frac{3}{2}\log n + O(\log\log n)$. □

[19] Note that for leaf nodes the label is half size of that of an internal node, since we only need to keep the leaf number rather than two endpoints of an interval.