Compact labeling schemes for ancestor queries

(Extended Abstract)

Serge Abiteboul* Haim Kaplan† Tova Milo ‡

Abstract

We consider the following problem. Give a rooted 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 only at the labels, if one node is an ancestor of the other. The best known labeling scheme is rather straightforward and uses labels of size at most $2\log n$, where n is the number of vertices in the tree. Our main result in this paper is a labeling scheme with maximum label size close to $\frac{3}{2}\log n$.

Our motivation for studying this problem is enhancing the performance of Web search engines. In the context of this application each indexed document is a tree and the labels of all trees are maintained in main memory. Therefore even small improvements in the maximum label size are important.

There are no lower bounds known for this problem except for an obvious lower bound of $\log n$ that follows from the fact that different vertices must have different labels. The question whether one can find even shorter labels remains an intriguing open question.

1 Introduction

The problem we study in this paper is the following. Given a tree T, what are the most compact labels that one can attach to nodes in the tree such that, given the labels of two nodes v and w, it is possible to determine in constant time – by looking only at the labels – if v is an ancestor of w. When measuring the quality of a labeling scheme, we will look at the size of the maximal assigned label, as a function of the size of the tree. (The issue of average size is left for future work.) We will be interested in the **exact** function rather than its order of growth - constant factors do matter! (e.g. labels with size bounded by, say $1.5\log(n)$, will be considered significantly better than $2\log(n)$[1].) Note that clearly one cannot go below $\log(n)$ bits - if the maximal label has less than $\log(n)$ bits, there are not enough labels in the domain even to just identify all nodes. But how much more is actually needed for answering ancestor queries?

To situate this somewhat abstract problem in today's Web oriented world and to motivate its study, we briefly describe next the practical origin of the problem together with some of the main features required by today's (and tomorrow's) Web search engines, and explain how a good solution to the above problem can contribute to better performance of such search engines. (A more detailed description is given in Section 2). The key observation is that Web documents obeying the XML standard [16] (the successor of the current HTML standard [18]) can be viewed as labeled trees (basically the parse tree of the document). Typical Web queries over such documents amount to testing component-of relationships among document items, which correspond to ancestor relationships among the analogous tree nodes [7, 17, 20, 1]. Web-search engines process such queries using an index structure that summarizes this ancestor information. Due to the enormous size of the Web data and to its distributed fashion, it is essential that queries be answered as much as possible using the index only, without accessing the actual documents. Therefore, when the engine comes to determine ancestor relationships, it has to do it based only on the index labels given to the nodes. To allow for good performance it is essential that the index structure (or at least a large part of it) resides in main memory. Observe that we are talking here about huge numbers - thousands of giga bytes of labels [4]. Since a main factor which determines the index size is the length of the labels, reducing this length, even by a constant factor, is a critical issue, contributing both to memory cost reduction and performance improvement.

Before presenting our results, let us consider a simple labeling scheme, which we call in the sequel the *parenthesis scheme*. Assume T is a rooted tree in which every internal node has at least two children. Label the leaves of T, say from left to right by consecutive integers starting from 1; then label each internal node v by a pair (interval) $[id_l(v), id_r(v)]$, where $id_l(v)$ $(id_r(v))$ is

*INRIA-Rocquencourt, 78153 Le Chesnay CEDEX, France, Serge.Abiteboul@inria.fr

†School of computer science, Tel Aviv University, Tel Aviv 69978, haimk@math.tau.ac.il

‡School of computer science, Tel Aviv University, Tel Aviv 69978, milo@math.tau.ac.il

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

the the label of v's leftmost (rightmost) leaf descendent. Ancestor test for two nodes then amounts to a simple interval containment test for the corresponding labels. It is easy to see that the size of the labels in this scheme is bounded by $2\log(n)$, n being the number of nodes. Interestingly, although rather naive, the bound obtained by this simple scheme is in fact the tightest we could find in the existing literature (see more on that in Section 2). So the question motivating the present work is can one do better?

It turns out that the answer is positive. In fact, we present here two such improved labeling schemes, trading simplicity and compactness. The first scheme is based on a two level partition of the tree, to a main top piece $\overline{T}$ and forest of pruned subtrees. Each subtree is labeled independently using the parenthesis scheme, with the final label of a vertex obtained by a combination of its parenthesized label and the label of its lowest (leaf) ancestor in $\overline{T}$. Intuitively, we take advantage of the fact that, in the regular parenthesis scheme, the label of a leaf is half the size of a label of an internal node, and use this to balance out the sizes of the combined labels and thereby reduce the maximum label size. A subtlety here is the dedicated pruning process that we use and which is essential to guarantee the compactness of the obtained labels. Using this two-level scheme, we obtain labels of size $\frac{5}{3}\log n + O(1)$. Our next step is to improve the construction by making it recursive. When applying the recursive scheme about $\log\log n$ times we obtain labels of size $\frac{3}{2}\log n + O(\log\log n)$. The crux here is to determine the optimal pruning policy for each recursion step, which we obtain by solving a system of k linear equations where k is the recursion depth.

We use the RAM model of computation, and assume that the size of a computer word is $\Omega(\log n)$ (hence the basic operations on the labels can be performed in constant time). In particular we assume that the RAM model includes the ability to perform bit manipulations like shift, bitwise AND, OR, XOR, and base-two discrete logarithm in constant time.

The paper is organized as follows. In Section 2 we present the practical motivation for the problem studied in the paper and consider related work. The basic two-level labeling scheme is then described in Section 3, and its recursive version in Section 4. Finally we conclude in Section 5.

2 Motivation and related work

The Web is constantly growing and contains a huge amount of useful information, often presented in HTML format. To retrieve such data, people typically use search engines like Altavista [2] or Google [3] which provide full-text indexing services (the user gives a few words and the engine returns documents containing those words). An HTML document consists of text interspersed with tag fields such as `<I> ... </I>` to describe the presentation of the page, the inclusion of pictures, hyperlinks, forms etc. [18]. These tags do not provide however any information about the semantic nature of the document components, which makes the analysis of the documents content difficult. To remedy this, a new Web standard, the XML data exchange format [16], has been proposed. Just like HTML, XML documents feature tags. But rather than providing instructions on how the document is to be displayed[2] they provide information on the logical structure of the document, with each semantic element starting with an <itemname> tag and ending with an </itemname> tag. To see an example, consider Figure 1 which shows a fragment of an XML document describing books catalog. The `<Category> and </Category>` tags are used to delimit the information corresponding to one catalog category. Each category item consists of a sequence of tagged sub-items such as `Category_Name`, `Book`, etc.

Now users can ask not only full-text queries (e.g. find documents containing the word "Fielding") but rather 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$) [7, 17, 20, 1]. 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 [11, 6, 19].

Going back to the problem stated at the introduction, observe 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 query of the above forms amounts to finding nodes with particular tags (book, author, price) having certain ancestor relationship between them (e.g. book nodes that are ancestors of qualifying author and price nodes).

To understand how to implement such structural queries consider the following rather typical implementation of a search engine. The engine's heart is a full-text index which is essentially a big hash table whose entries are words in the indexed documents. For standard HTML documents, the entries will be arbitrary words, and the index will associate with each entry the ids of documents containing these words. For XML doc-

[2]This may be specified in a *style-sheet*.

```
<Catalog>
  <Category>
    <Category_Name> Literature & Fiction </Category_Name>
    <Book>
      <ISBN>12445</ISBN> <Title>Bridget Jones's Diary</Title>
      <Authors><Author> Helen Fielding </Author></Authors>
      <Price>$10.39</Price> ...
    </Book>
    <Book> ... </Book>
    ...
  </Category>
  <Category> ... </Category>
  ...
</Catalog>
```

Figure 1: The Books Catalog

uments the granularity can be refined. The entries in the hash table are the item names or values. To allow structural queries we give each item (node) in the XML document (tree) a unique label, and associate with an entry in the hash table the 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 descendent, i.e. component, of the other), queries of the above form 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 one billion and will grow to 100 billion by 2002 [4]. Search engines typically index 15% of these pages, consuming around 0.3 of the original document size (just for indexing the text with the document ids, not including separation to items or ancestor information). With 10KB text per page on the average, this already leads to about 500 giga for now and 50,000 for 2002. Experiments to evaluate the effect of adding the structural information to the index show that adding a label of just one integer (4 bytes) to each word occurrence almost doubles the index size [21]. 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.

Consider now the actual physical representation of the labels in the index. In a fixed-size representation each label is allocated a fixed space, determined by the maximal potential label size. In a variable-size representation labels are allocated a variable size space (consisting of a fixed prefix stating the actual size of the label, followed by the label itself) and the size of the index is determined by the average label size plus the log of the maximum potential size of a label (the space allocated for the fixed prefix). We study here the case of fixed-size representation, aiming to improve the bound on the maximal label size. The case of variable-size representation is left for future research.

Related Work. A variant of the parenthesis scheme was first described by Santoro and Khatib [14]. Instead of numbering the leaves from left to right, Santoro and Khatib suggested to 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 depth first traversal. Implementations of variants of the parenthesis scheme are integerated in the Xyleme XML warehouse system [21].

Peleg [13] considered informative labeling schemes of trees and graphs. In his paper Peleg formalizes the following general problem. Given a graph G and a function f on subsets 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 an 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 a *distance labeling scheme*. Peleg computes upper and lower bounds on the maximum label size for various graph classes and functions f.

Maybe the problem closest to ours among the ones 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 size of the labels needed for the separation level problem is close (within an additive $\log n$ factor) to the size of the labels needed for a *distance labeling scheme* of T. From results on distance labeling schemes [12, 9] of trees it follows that there is a labeling scheme for the separation level problem with labels of size $O(\log^2 n)$, 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[3] of the lowest common ancestor of v and w in T, and proves that this bound is tight up to a constant factor. For more problems within this framework see [13]. To the best of our knowledge, prior to our work here, there was no effort to optimize the constant factors for any problem within this framework.

A more distantly related problem is the problem of preprocessing a tree T such that subsequent lowest common ancestor queries on pairs $v, w \in T$ can be answered quickly. Several algorithms have been developed for this problem [15, 10, 8, 5] and variants of this problem which allow certain operations that modify T in between queries. The preprocessing part of these algorithms organize some information on the tree such that one can efficiently find lowest common ancestors and in particular, given v and w determine whether v is an ancestor of w. This information however is not organized in a distibuted fashion as a labeling of the nodes. We believe that the ideas in these algorithms may be useful in constructing labeling schemes. The problem of whether the information maintained by these algorithms can be distributed among the nodes so as to allow ancestor queries just from the labels of the nodes involved deserves further study. Nevertheless, even if the answer to the latter question turns out to be positive the size of the labels in such labeling scheme would be larger than $2 \log n$.

3 Two level labeling

We start by describing a rather simple labeling scheme based on a two level partition of the tree, to a top piece $\overline{T}$ and a forest of pruned subtrees. Using this two-level scheme we obtain labels of size $\frac{5}{3} \log n + O(1)$. We call the algorithm that preprocess the tree by pruning subtrees the *Prune&contract algorithm*.

The idea at a high level is to label $\overline{T}$ and each of the pruned subtrees independently, using the parenthesis scheme. The label of a vertex in a pruned subtree is then a concatenation of its parenthesized label from the pruned subtree and the label of its lowest leaf ancestor in $\overline{T}$. When comparing labels of two vertices v and w there are three possible cases: if they are in different subtrees (hence their labels contain different ancestor leaf labels from $\overline{T}$) then they are clearly unrelated. If they belong to the same subtree, we can determine whether v is an ancestor of w by comparing their parenthesized 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.

Intuitively, we take advantage of the fact that in the parenthesis scheme the label of a leaf is half the size of a label of an internal node, and use this to balance out the size of the labels and thereby reduce the maximum label size. There are however several subtle technical issues associated with the pruning process, which we describe in Section 3.1. These issues arise since we want both the pruned subtree and the remaining part of the tree to have a small number of leaves (hence small parenthesized labels). This requires enhancing the pruning process with a *node contraction* mechanism, merging redundant leaves, and hence somewhat complicating the simple decision procedure sketched above.

We next describe the Prune&contract algorithm. Then in the following subsections we explain how to assign labels to the resulting subtrees and how to evaluate ancestor queries based on these labels. The algorithm uses a threshold parameter x. For the two-level scheme of this section x would be $\frac{1}{3}$. In the next section we show how to apply the partitioning scheme recursively to obtain even shorter labels. In this recursive scheme we will use the Prune&contract algorithm with different threshold parameters, and one of the main issues will be to determine its appropriate value for minimizing the label size.

3.1 The Prune&contract algorithm.

Without loss of generality assume that T is a rooted tree in which every internal node has at least two children[4]. For $v \in T$ we define $l_T(v)$ to be the number of leaves in the subtree of T rooted by v. We will abbreviate $l_T(v)$ to $l(v)$ when T is clear from the context. Assume n is the number of leaves in T. We modify T in three

[3] Identifier are given together with T as input.

[4] Otherwise add a child to each unary internal node, in which case the n in the following discussion becomes in the worst case proportional to the number of nodes in T. This is the reason that in Section 1 we used n as the number of nodes.

phases. The first phase prunes some subtrees from T; the second phase adds to the resulting tree some new leaves representing the pruned subtrees; finally the last phase merges some nodes to obtain a more compact tree. When describing each of these phases we will also point out some of the main properties of the obtained trees. These will prove to be useful later when considering the size of the obtained labels.

Phase 1 (Pruning Phase): We prune subtrees of T by cutting some of the edges. For every node $v \in T$ with $l(v) > 2n^x$ we look at each child u of v and consider the following two cases.
Case 1: if $l(u) \leq n^x$ we cut the edge (v, u),
Case 2: if $n^x < l(u) \leq 2n^x$ we cut all the edges (u, w) to the children w of u.

We denote by T^1 the part of T which remains after the pruning phase. The following lemma states some properties of the vertices in this T^1. We omit the proof which is straightforward.

LEMMA 3.1. *1) For every node $v \in T^1$, $l_T(v) > n^x$.*
2) If v is an internal node in T^1 (i.e. it has at least one child) then $l_T(v) > 2n^x$.
3) If $v \in T^1$ has a child u in T that was pruned and $l_T(u) > n^x$, then v must be a leaf in T^1 and $l_T(v) \leq 2n^x$.

Phase 2 (Adding leaves under heavy and binary nodes): In the second phase we add to T^1 some leaves representing the pruned subtrees. For that we first partition the pruned subtrees into groups and then add one leaf representative for each such group. Before describing the exact process we need to introduce the notion of *heavy nodes*: We call a node $v \in T^1$ *heavy* if some of its children in T, say $v_1, v_2, \ldots, v_p$, were pruned, and $\sum_{j=1\ldots p} l(v_j) > n^x$. By the definition of Phase 1 and Lemma 3.1, every leaf of T^1 must be a heavy node. Other vertices of T^1 may also be heavy. We will consider below separately two cases: heavy nodes (both leaves and internal), and non-heavy internal nodes.

<u>*Heavy nodes:*</u> Let $v \in T^1$ be a heavy node and let $v_1, v_2, \ldots, v_p$ be the pruned children of v in T (which therefore do not belong to T^1). We first partition $v_1, v_2, \ldots, v_p$ into groups $G_1, G_2, \ldots, G_{p'}$: If $\sum_{j=1\ldots p} l(v_j) \leq 2n^x$ then all nodes belong to a single group G_1. Otherwise they are partitioned such that for all groups $G_i, i = 1 \ldots p'$, $\sum_{j \in G_i} l(v_j) \leq 2n^x$, and furthermore, for all groups except perhaps the last one, $n^x < \sum_{j \in G_i} l(v_j)$. (I.e. all groups but perhaps the last contain at least n^x pruned leaves and no more than $2n^x$ such leaves.)

We find such a partition by the following recursive greedy algorithm: If $\sum_{1 \leq j \leq p} l(v_j) \leq 2n^x$ then the partition consists of only one group G_1 containing $v_1, v_2, \ldots, v_p$. Otherwise, find the minimum index i such $\sum_{j \leq i} l(v_j) > n^x$, and put $v_1, \ldots, v_i$ into group G_1. If $i < p$ then we partition $v_{i+1}, \ldots, v_p$ into groups $G_2, \ldots, G_{p'}$ recursively using the same algorithm.

The following lemma proves that the size of every group produced by this greedy algorithm is indeed as we claimed. The proof is easy using the properties of T^1 from Lemma 3.1.

LEMMA 3.2. *Let v be a heavy node and let $v_1, v_2, \ldots, v_p$ be the children of v that were pruned. The partition of $v_1, v_2, \ldots, v_p$ into $G_1, G_2, \ldots, G_{p'}$ produced by the greedy algorithm described above is such that $\sum_{j \in G_i} l(v_j) \leq 2n^x$ for every $1 \leq i \leq p'$, and if $p' > 1$, $n^x < \sum_{j \in G_i} l(v_j)$ for every $1 \leq i \leq p' - 1$.*

We can also bound the number of the groups relative to the total pruned weight under v.

LEMMA 3.3. *Let v be a heavy node and let $v_1, v_2, \ldots, v_p$ be the children of v that were pruned. Let $k \geq 1$ be an integer such that $kn^x < \sum_{j=1\ldots p} l(v_j) \leq (k+1)n^x$. Then, the number of groups, p', produced by the greedy partitioning algorithm described above is at most k.*

Let v be a heavy node, and let $G_1, \ldots, G_{p'}$ be the groups into which we partitioned the pruned children of v. We now use the partition to add new leaves to T^1: For each such group G_j we add in T^1 a corresponding new child leaf g_j to v. We do this for all heavy nodes v except when v is a leaf with a single partition group G_1. (Since in this case adding one leaf to v will make v an internal node with a single child, which is redundant and we want to avoid.)

<u>*Non heavy internal nodes:*</u> Finally, for every $v \in T^1$ with more than one child in T^1, which had lost some of its children in T, but not enough to make it heavy, we add a new leaf child l_v.

We denote by T^2 the tree obtained from T^1 after phase 2.

Phase 3 (Contracting paths of unary nodes): In this phase we merge redundant nodes to obtain a more compact tree. In particular we focus on paths of unary nodes in T^2. Observe that although we have started with a tree T where all internal nodes had at least two children, some of these children may have been pruned in Phase 1 so now T^2 may contain (paths of) nodes with a single child. As in the previous phase, we first partition the nodes on such paths into groups and then contract each group to a single node.

Let $p =< u_1, u_2, \dots u_k >$ be a maximal path [5] in T^2 s.t. every u_i, $2 \le i \le k$, is the only child of u_{i-1} in T^2 and where u_k also has a single child in T^2. (In particular this means that u_k is not a leaf.) We note that all the nodes on the path are not heavy, as if they were, they should have gained a child in Phase 2 and become binary in T^2. For a node u_i on the path, we define the weight of u_i, denoted $w(u_i)$, to be the total number of leaf descendants of v's pruned children. As in Phase 2 we partition p into subpaths $p_1, \dots, p_{k'}$ s.t. the sum of the weights of the vertices in each subpath, except possibly the last one, is between n^x and $2n^x$. The partitioning is done greedily with an algorithm similar to the one used above for grouping of pruned subtrees. (We omit the details from this extended abstract.) We can then prove the following lemma analogous to Lemma 3.2.

LEMMA 3.4. *Let $p =< u_1, \dots u_k >$ be a maximal path in T^2 of nodes, each having a single child and where u_k is not a leaf. The partition of P into sub-paths $p_1, p_2, \dots, p_{k'}$ produced by the greedy algorithm is such that $\sum_{u_j \in p_i} w(u_j) \le 2n^x$ for every $1 \le i \le k'$ and if $k' > 1$, $n^x < \sum_{u_j \in p_i} w(u_j)$ for every $1 \le i \le k' - 1$.*

Since we assumed that all the internal nodes in T have at least two children, the weight of every node on p is at least 1. Moreover, since the total weight of the nodes on each subpath is at most $2 * n^x$, we have:

LEMMA 3.5. *The length of every subpath p_i in the partition of p is at most $2 * n^x$.*

Once all the maximal paths are partitioned, we complete Phase 3 by contracting each subpath p_i into a single node c_{p_i} and attach to it a new leaf node l_{p_i}. We will refer in the sequel to such a pair of a contracted node and a leaf node as a (c_{p_i}, l_{p_i}) *pair.*

Let T^3 be the tree produced by Phase 3. The following lemma gives an upper bound on the number of leaves in T^3 in terms of the number of leaves in T.

LEMMA 3.6. *The number of leaves in T^3 is at most $3n^{1-x}$ where n is the number of leaves in T.*

Proof. Let $s =< u_1, \dots, u_j >$ be a subpath that has been contracted when we formed T^3 by Phase 3, and let l_s and c_s be the vertices that correspond to s in T^3. We call s a *light* subpath if $\sum_{i=1}^{j} w(u_i) < n^x$ and a *heavy* subpath otherwise. Depending upon whether s is light or heavy we shall also call l_s and c_s light or heavy respectively. Our partitioning algorithm used in Phase 3 may create only one light subpath for each path it partitions. Therefore there are no two adjacent light nodes c_s and $c_{s'}$ in T^3.

For the purposes of this proof only we consider the tree $\overline{T}^3$ obtained from T^3 as follows. We remove each light node c_s that has a non-leaf child and replace it by an edge connecting its parent directly to one of its children that is not a leaf. Note that both the parent and every non-leaf child of c_s is not a node associated with a light path so $\overline{T}^3$ is well defined. We also remove leaves l_v which were added as children of non-heavy nodes in Phase 1. We next prove that $\overline{T}^3$ contains at most n^{1-x} leaves. To that extent we associate with each leaf of $\overline{T}^3$ a set of n^x leaves of T such that sets associated with different leaves are disjoint.

The tree $\overline{T}^3$ has four kinds of leaves. We consider them one by one. The first group contains leaves that were added as children of heavy nodes in Phase 2. Let v be a heavy node in T^1 that has gained leaves in T^2. If v had lost z of its descendent leaves in T in the pruning phase, and $kn^x < z \le (k+1)n^x$ then by Lemma 3.3 v has gained at most k leaves. We associate with each of these new leaves n^x of the z original leaf descendants of v in T which were pruned.

The second group consists of the heavy leaves l_s which were added in Phase 3. Since s is heavy, the total number of leaf descendants of nodes v such that (u, v) was cut in the pruning phase and $u \in s$ is at least n^x. We associate these leaves with l_s.

Third, there are leaves which are original nodes of T but haven't gained children since all their descendants fit in one group of Phase 1. The forth group consists of leaves l_s associated with light paths s such that c_s has only leaf descendants in T^3. Let s be such a path. The last node of s has a single child v in T^2. Node v becomes a child of c_s in T^3 and since c_s has only leaf descendants in T^3, v must be a leaf both in T^2 and T^3. Moreover, v is a node of T and cannot be a leaf added in Phase 2, since every leaf added in Phase 2 has a parent in T^2 whose degree is more than one. We conclude that each node of the forth group has a sibling from the third group. We associate leaves of T with leaves in these two groups as follows.

If w is a leaf in the third group (i.e. w is a node of T) that has no sibling in the forth group (i.e. a node associated with a light subpath) then we associate with w all its leaf descendants in T (all were pruned in Phase 1). Otherwise w has a leaf sibling of the form l_s where s is a light path. By lemma 3.1 the parent of w in T (which is the last node on s) had at least $2n^x$ leaf descendants and all of them were pruned. We associated half of these leaves with v and the other half with l_s.

Clearly the sets associated with different leaves of

[5] By *maximal* we mean that there is no path which satisfies the conditions and contains P.

$\overline{T}^3$ are disjoint. Each such set is of size at least n^x, so there are at most n^{1-x} such sets and we obtain that $\overline{T}^3$ has at most n^{1-x} leaves.

Last we account for leaves in T^3 that are not in $\overline{T}^3$. There are two kinds of such leaves. First, there are leaves that were added in Phase 1 as children of nodes of degree at least 2 in T^1 that had lost some subtrees in the pruning phase but not enough to make them heavy. The parents of these leaves are also internal nodes in $\overline{T}^3$ so there is at most one such leaf per internal node of $\overline{T}^3$. Second, there are leaves associated with light paths s, such that c_s had a nonleaf child in T^3, say v. As mentioned before v cannot be associated with a light path and hence it is an internal node of $\overline{T}^3$. If we charge c_s to v then each internal node of $\overline{T}^3$ is being charged by at most one leaf l_s associated with a light path.

Every internal node of $\overline{T}^3$ had at least two children so the number of internal node in $\overline{T}^3$ cannot be greater than n^{1-x} which is the number of leaves. We associated all the leaves in $T^3 - \overline{T}^3$ with internal nodes of $\overline{T}^3$ such that at most two leaves from $T^3 - \overline{T}^3$ are associated with each internal node of $\overline{T}^3$ and therefore the theorem follows. □

3.2 Node representatives. We are now almost ready to define the labels of nodes in the tree. As mentioned above, the idea is to label independently T^3 and each of the pruned subtrees, using the basic parenthesis labeling scheme, and then give the pruned nodes a label which is composed of their own parenthesized label plus the label of some representative ancestor leaf node in T^3.

To explain how exactly this is done we first need to define formally the notions of *representative nodes* and *index* and analyze their properties. Observe that the definitions are slightly more general than actually needed here - for example we define representatives also to nodes in T^3 and not just in the pruned subtrees. This however will be proved useful later when we consider the recursive version of the labeling scheme.

DEFINITION 3.1. *Let v be some node in T. The* representative *node of v in T^3, denoted $r(v)$, is defined as follows. If $v \in T \cap T^3$ then $r(v) = v$. Otherwise, we consider the following two cases.*

Case 1. v **was contracted as part of subpath** s **in Phase 3:** *v is represented by the node $c_s \in T^3$.*

Case 2. v **was pruned from** T **in the pruning phase:** *Let u be the farthest ancestor of v in T that is still pruned, and let w be the parent of u in $T \cap T^1$.*

(i) If w is a heavy node that gained leaves in Phase 1, v is represented by the leaf g_i which corresponds to the partition group that contains u. Otherwise if w is a heavy leaf that has not gained leaves (because it had only one group in the partition), v is represented by w itself.
(ii) If w was a non heavy node that had more than one child in T^1 then v is represented by the leaf l_w added as a child of w by Phase 1. Otherwise if w was a non heavy node with a single child (hence was contracted as part of some subpath s in Phase 3), then v is represented by the leaf l_s added to the contracted path in Phase 3.

Note that all the nodes of T residing in single connected subtree that was pruned at Phase 1 share the same representative node in T^3. For each node $w \in T^3$ that represents pruned vertices we denote by $F(w)$ the forest of nodes of T that it represents. We also assume that vertices on a contracted subpath s are indexed consecutively such that the node closest to the root of T gets the index 1. We then generalize the notion of *index* to other nodes in the tree as follows.

DEFINITION 3.2. *For a vertex v that belongs to a contracted subpath we define the* index of v *as its index on its contracted subpath. For a vertex v not on a contracted subpath having some ancestor u on a contracted subpath we define the* index of v *to be the index of its lowest ancestor on a contracted subpath. Finally for every other node v we define the* index of v *as zero.*

The following theorem 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 labeling scheme which we describe in the next subsection captures exactly the information needed to apply this characterization, and determine whether v is an ancestor of w based solely on their labels. The proof of the theorem follows from the definitions and the algorithms we described in a straightforward way.

THEOREM 3.1. *Vertex v is an ancestor of vertex w in T iff one of the following conditions holds.*

1. *Vertices v and w have a common representative which is a leaf of T^3 and v is an ancestor of w in the forest associated with that leaf.*
2. *Vertices v and w have a common representative which is an internal node of T^3 and the index of v is smaller than the index of w.*
3. *Vertex $r(v)$ is an ancestor of $r(w)$ in T^3. If $r(v)$ and $r(w)$ constitute a (c_s, l_s) pair (i.e. resulted from the contraction of some subpath s) we also require that the v's index is no greater than that of w.*

3.3 Assigning labels to nodes. We first label T^3 using the basic parenthesis scheme described in the introduction, and denote this labeling $P(T^3)$. Similarly, we label the pruned nodes using the basic parenthesis scheme: for every $w \in T^3$ that represents pruned vertices we label the corresponding pruned forest $F(w)$, and denote this labeling by $P(F(w))$. Once this is done we are ready to define the labels of the vertices in T using the labels in $P(T^3)$ and the $P(F(w))$'s. We define our labels according to the type of the node:

Case 1. Node v is in T^3, i.e. $r(v) = v$: The label of v is its label from $P(T^3)$.

Case 2. Node v was contracted, i.e. $r(v) = c_s$ for some subpath s: The label of v is the label of c_s from $P(T^3)$ concatenated with the index of v.

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 $P(T^3)$. Note that in this case $r(v)$ must be a leaf of T^3. (ii) The index of v. (iii) The label of v in $P(F(r(v)))$.

More precisely, the bit representation of the labes is as follows. To destingush between the different cases we will use one extra bit which will be zero if v is a node of type 1 or 2, and 1 if it is a node of type 3. When the bit is zero it would follow by two fixed length fields. The first field contains a label of a node in T^3 and the second field contains an index (which will be zero for Case 1). When the bit is 1 it would follow by three fixed length fields. The first field contains a label in $P(T^3)$ of a leaf, $r(v)$, in T^3. The second contains an index of a vertex, and the third contains the label of v in $P(F(r(v)))$. In both types of labels we refer to the index field as the *index of the label.*

What is the size of the obtained labels? The length of the first field in a label of a node of type 1 or 2 is $2 * \lceil \log(3 * n^{1-x}) \rceil$ bits. By Lemma 3.6 this suffices to represent any label of $P(T^3)$. The length of the second index field in such label is $\lceil \log(2n^x) \rceil$ bits. By Lemma 3.5 this suffices to represent any index of a vertex in a contracted subpath since the length of any such subpath is at most $2n^x$. Summing up we obtain that the length of a label of nodes of type 1 and 2 is

$$2 * \lceil \log(3 * n^{1-x}) \rceil + \lceil \log(2n^x) \rceil + 1 = (2-x) \log n + O(1)$$

Consider now a label of a node of type 3. For the first field we need $\lceil \log(3 * n^{1-x}) \rceil$ as it is a label of a leaf in $P(T^3)$. For the index field we need $\lceil \log(2 * n^x) \rceil$ as in the labels of nodes of type 1 or 2. For the third field we need $2 * \lceil \log(2 * n^x) \rceil$, since the maximum size of a forest, $F(w)$, is $2 * n^x$. Summing up we obtain that the total size of a label of a node of type 3 is

$$\lceil \log(3*n^{1-x}) \rceil + 3*\lceil \log(2*n^x) \rceil + 1 = (1+2x) \log n + O(1)$$

To balance out the lengths of the different types of labels we choose $x = 1/3$ and obtain that the maximum size of a label is $\frac{5}{3} \log n + O(1)$.

3.4 Answering ancestor queries. Given two labels $L(v)$ and $L(w)$ we next describe how to decide whether v is an ancestor of w. The algorithm closely follows Theorem 3.1. We first find the leftmost field (skipping the type bit) in which $L(v)$ and $L(w)$ disagree. We then consider the following three cases.

Case 1: The type bit of both $L(v)$ and $L(w)$ is '1' and they agree on the first field. In this case both v and w were pruned from T and are represented by the same leaf $l = r(v) = r(w)$ in T^3. Therefore, the answer is positive iff v is an ancestor of w in $F(l)$. Since the third field contains the labels of v and w in $P(F(l))$ we compare these labels and answer positively if the interval of v contains the interval of w.

Case 2: The type bit of both $L(v)$ and $L(w)$ is '0' and they agree on the first field. In this case v and w were both on the same contracted subpath. Therefore, we answer positively if and only if the index of v is smaller than the index of w on the contracted subpath. (Recall that the index of v or w on the subpath is stored in the second field of their labels).

Case 3: The first field of v differs from the first field of w. We check whether $r(v)$ is an ancestor of $r(w)$ in T^3 by comparing their labels in $P(T^3)$ (which are stored in the first field of their labels). If not the answer is negative. Otherwise, the answer is positive unless $r(v)$ and $r(w)$ are a (c_s, l_s) pair. Notice that we can deduce that $r(v)$ and $r(w)$ are a (c_s, l_s) pair from the labels of v and w: this holds iff (i) the type bit of v is '0' and its index field greater than zero, (ii) the type bit of w is '1', and (iii) the label of $r(w)$ in $P(T^3)$ is the leftmost or rightmost parenthesis of the label of $r(v)$. If $r(v)$ and $r(w)$ are indeed a (c_s, l_s) pair we answer positively if and only of the index of v is no greater than that of w.

4 Smaller labels via recursion

In this section, we improve the construction from Section 3 by making it recursive. We explain next the implications to each point considered in the previous section. When applying the recursion about $\log \log n$ times, we obtain labels of size $\frac{3}{2} \log n + O(\log \log n)$, n being, as before, the number of leaves in T.

Prune&contract. We denote the number of recursion levels by k. Let $x_1, \ldots, x_k$ be a sequence of numbers that we shall fix later. We start by applying the Prune&contract algorithm from Section 3.1 recursively as follows. First we process T using Prune&contract with x_1 as the pruning threshold. Let $T_1 \equiv T^3$ be the resulting tree. Then we apply Prune&contract again to T_1 with a threshold x_2 and denote the resulting tree T_1^3 by T_2. We continue recursively and at each step define T_i, $1 \le i \le k$, to be the tree T_{i-1}^3 obtained by applying Prune&contract to T_{i-1} with threshold x_i. (With T_0 being the original tree T.)

Node representatives. Every vertex of T has a representative $r(v)$ in T_1 as defined in Section 3.1. Similarly each vertex $v \in T_i$, $0 \le i \le k-1$, has a representative $r(v) \in T_{i+1}$. The *representative at level* i, $1 \le i \le k$, of a node $v \in T$, denoted $r^i(v)$, is defined inductively as $r^i(v) = r(r^{i-1}(v))$, with $r^1(v) = r(v)$.

Assigning labels to nodes. As before we label T_k with the simple parenthesis scheme and denote this labeling by $P(T_k)$. Then we do the same for the pruned subtrees: For each leaf $w \in T_i$ such that $F(w)$ is not empty we label $F(w)$ with the simple parenthesis scheme and denote this labeling $P(F(w))$. Finally, we define the labels of the original tree T based on these labels. We start below with an intuitive description of the label, then we give the exact bit structure.

The label of a node $v \in T$ starts with some prefix of length $\lceil \log(k) \rceil$ bits (to be defined below), followed by the label of $r^k(v)$ in $P(T_k)$. Then we continue iteratively and for every i from k down to 1 we concatenate the index of $r^{i-1}(v)$ along with the label of $r^{i-1}(v)$ in $F(r^i(v))$ (if $F(r^i(v)) \neq \emptyset$).

To describe the bit structure of a label we first need to define the *level of a vertex* $v \in T$, denoted by level(v). level(v) is the minimum j for which $r^j(v)$ is a leaf and $\neq v$, or $k+1$ if no such j exists. As mentioned above, our labels will start with a prefix. This prefix will contain level(v)mod$(k+1)$ (i.e. it contains level(v) unless level$(v) = k+1$ in which case it contains 0). The structure of the consecutive bits is based on the lemma below which follows immediately from the definition of the Prune&contract process.

Lemma 4.1. *If $r^i(v)$ is a leaf then for every $j > i$ $r^j(v)$ is also a leaf.*

The label of a node v contains $k+1$ fixed length fields following the prefix. We index them from k down to 0. The length and the content of these fields depend on level(v): The k^{th} field contains the label of $r^k(v)$ in $P(T_k)$. Then, for every j from $k-1$ down to level$(v)-1$ the j^{th} field is split into two components. The first component contains the index of $r^{j-1}(v)$, and the second component contains the label of $r^{j-1}(v)$ in $F(r^j(v))$. Following this sequence of pairs we have a sequence of singleton fields, one field for every j from level$(v)-2$ down to 0, each containing the index of $r^j(v)$.

The size of the labels in the recursive scheme. Each field of a label is of fixed length that is determined by the maximum possible value that can be assigned to that field. Recall that by Lemma 3.6 the number of leaves in T_k is at most $3^k n^{(1-S)}$ where $S = \sum_{i=1}^{k} x_i$. Also, by Lemma 3.5 for every $1 \le i \le k$ the index of $r^{i-1}(v)$ is at most $2 * n^{x_i}$. So, for a vertex v such that level$(v) = k+1$ we need $2 * \lceil \log(3^k n^{(1-S)}) \rceil$ bits for the first field and $\lceil \log(2 * n^{x_i}) \rceil$ for the index of $r^{i-1}(v)$ for $1 \le i \le k$. So the total size of such label, without the prefix, is

$$2 * \lceil \log 3^k n^{(1-S)} \rceil + \sum_{i=1}^{k} \lceil \log(2 * n^{x_i}) \rceil \tag{4.1}$$

For a vertex v such that level$(v) < k+1$ the k^{th} field of the label of v contains a label of a leaf in $P(T_k)$. The length of this field is $\lceil \log 3^k n^{(1-S)} \rceil$. For every $k > i \ge$ level$(v) - 1$ the i^{th} field contains a pair. The magnitude of the first component which is the index of $r^i(v)$ is at most $2 * n^{x_{i+1}}$ by Lemma 3.5. The number of leaves in $F(r^i(v))$ is at most $2 * n^{x_i}$. Thus to describe the label of $r^i(v)$ which is a leaf in $F(r^{i+1}(v))$ in $P(F(r^{i+1}(v)))$ for $i \ge$ level(v) we need $\lceil \log 2 * n^{x_{i+1}} \rceil$ bits. To describe the label of $r^{\text{level}(v)-1}(v)$ in $P(F(r^{\text{level}(v)}(v)))$ we need at most $2 * \lceil \log 2 * n^{x_{\text{level}(v)}} \rceil$. By a similar argument, for every level$(v) - 2 \ge i \ge 0$ the size of the ith field is $\lceil \log 2 * n^{x_{i+1}} \rceil$. Summing up we obtain that the size of the label of v without the prefix is

$$\lceil \log 3^k n^{(1-S)} \rceil + \sum_{i=\text{level}(v)}^{k} 2 * \lceil \log(2 * n^{x_i}) \rceil + 3 * \lceil \log(2 * n^{x_{\text{level}(v)}}) \rceil + \sum_{i=1}^{\text{level}(v)-1} \lceil \log(2 * n^{x_i}) \rceil \tag{4.2}$$

It is clear from the above two equations that in order to minimize the maximum length of a code we need to find $x_1, \ldots, x_k$ that make the length of the codes for level$(v) = 1, \ldots, k+1$ equal. By solving this system of k equation in k unknowns $x_1, \ldots, x_k$ and neglecting some constant factors we obtain the following solution

$$x_i = \frac{2^{i-1}}{2^{k+1} - 1} \tag{4.3}$$

Choosing the x_i's accordingly we obtain a labeling scheme where the maximum label size is

$$\frac{3}{2} \log n + \frac{\frac{5}{2} \frac{1}{2^{k+1}}}{1 - \frac{1}{2^{k+1}}} \log n + O(k) \tag{4.4}$$

For $k = \log \log n$ the size of a label becomes $\frac{3}{2} \log n + O(\log \log n)$.

Answering ancestor queries. Finally we give an algorithm to determine, given the labels of v and w, whether v is an ancestor of w. At the heart of the algorithm are the following two main observations.

- It is easy to see that if v is an ancestor of w then level$(v) \geq$ level(w). The algorithm thus first compares the prefixes of the labels of v and w to verify that indeed level$(v) \geq$ level(w) and answers negatively if not.
- Consider the various representatives of v and w along the history of the Prune&contract applications. They start as different nodes but may at some point get to be represented by a single node. Observe that if this happens then from this point and on they (and thus their respective labels) behave in exactly the same way. So in fact to determine the relationship between the two nodes we need to look at their last representatives, just before this merger, and analyze their relationship. For that we find the leftmost field that is different in the labels of v and w and then compare the representative codes stored in this field in a similar manner to what we did in the basic algorithm in Section 3.4.

The full algorithm together with the relevant implementation details and a correctness proof will appear in the full version of this paper.

5 Concluding remarks

We presented two tree labeling schemes with labels of size $\frac{5}{3} \log n + O(1)$ and $\frac{3}{2} \log n + O(\log \log n)$ bits resp., such that, given only the labels of v and w, we can test in constant time if v is an ancestor of w. Clearly $\log n$ is a lower bound on the size of the labels as there cannot be two vertices with the same label. But the question whether there exists a labeling scheme with shorter labels is an intriguing open question.

Another interesting line for future research is to find a labeling scheme that allows additional queries while still keeping the labels compact. A candidate for such a query is the *parent* query which also has application to Web search engines, (one may sometimes search for XML items that *directly* contain some specific items, and not just in transitive containment). We are currently working on enhancing our labels such that parent queries are also supported.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan-Kaufmann, 340 Pine Street, Sixth Floor San Francisco, CA 94104, October 1999.

[2] Altavista. `http://www.altavista.com`.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *7th WWW*, 1998. `http://www.google.com`.

[4] D. Butler. Souped-up search engines. *Nature*, 405:112–115, May 2000.

[5] R. Cole and R. Hariharan. Dynamic lca queries on trees. In *SODA*, pages 235–244, 1999.

[6] R. Cover. The xml cover pages, extensible markup language - industry news. http://www.oasis-open.org/cover/xmlNews.html.

[7] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for xml. In *International World Wide Web Conference*, 1999.

[8] H. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *SODA*, pages 434–443, 1990.

[9] C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance labeling in graphs. 1999.

[10] D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–335, May 1984.

[11] Oasis. Organization for the Advancement of Structured Information Standards. `http://www.oasis-open.org`.

[12] D. Peleg. Proximity-preserving labeling schemes and their applications. In *Proc 25th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, pages 30–41, 1999.

[13] D. Peleg. Informative labeling schemes for graphs. Technical Report MCS00-05, The Weizmann Institute of Science, January 2000.

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

[15] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, December 1988.

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

[17] W3C. Extensible stylesheet language (xsl). http://-www.w3.org/Style/XSL/.

[18] W3C. Html home page. http://www.w3.org/MarkUp/.

[19] W3C. The w3c query languages workshop, dec 1998, boston, massachussets. http://www.w3.org/TandS/QL/QL98/cfp.html.

[20] W3C. Xsl transformations (xslt) specification. http://www.w3.org/TR/WD-xslt.

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