# Efficiently Supporting Structure Queries on Phylogenetic Trees *

Susan B. Davidson Junhyong Kim Yifeng Zheng

*University of Pennsylvania*

*susan@cis.upenn.edu, junhyong@sas.upenn.edu, yifeng@cis.upenn.edu*

## Abstract

*With phylogenetics becoming increasingly important in biomedical research, the number of phylogenetic studies is increasing rapidly and huge mount of phylogenetic data has been generated and stored in databases. How to efficiently extract information from the data has become an important research problem.*

*In this paper, we focus on a class of important queries on phylogenetic trees: structure queries which include least common ancestor, minimal spanning clade, tree pattern match and tree projection. After analyzing the characteristics of the phylogenetic tree as well as structure queries, we propose a storage system based on labeling using RDBMS and design algorithms for query evaluation. We implement these algorithms and compare them with existing techniques. Performance studies prove the efficiency of our strategy.*

## 1 Introduction

Phylogenetics – the science of identifying and understanding evolutionary relationship between different species – has become increasingly important in biomedical research. For example, within epidemiology it has been used to trace contact histories of infectious diseases [9], to identify the geographic origins of outbreaks (as in the case of West Nile Virus [13]), and to predict the timing of new introductions [17].

In response to the demand for phylogenetic trees, the number of phylogenetic studies is increasing rapidly and a variety of phylogenetic tree generation algorithms [12, 6, 10, 26, 29, 19, 18] have been proposed. The number of trees published is doubling every 5 years, and the number of sequences in GenBank that might be used to build trees is doubling even faster, roughly every year [1]. The size and scope of individual trees are also increasing rapidly, as recent publications of trees with hundreds to thousands of species demonstrate.

The growth of phylogenetic information and the need for on-line archival storage and retrieval has led to the establishment of several database systems, most notably TreeBase[23, 22] and Tree of Life[14] (ToL). ToL contains a single tree, and although it is still far from complete it is quite large; the current tree represented in XML format is almost 30MB [14]. TreeBase[23] currently contains more than 3000 trees.

To extract data of interest from these databases, various specialized search tools have been developed. TreeBase provides a keyword-based search tool which allows a user to enter a tree ID, the name of a taxon, or other identifying features to search the database of trees. ToL employs visualization techniques that allow the user to view a section of the tree, expand or contract portions of the tree, and to link to supporting literatures. However, neither of these systems allow users to search the structure of a phylogenetic tree. Since the structure of a phylogenetic tree models the important information about the taxa contained within the tree, structure search is very important.

Recent research efforts have therefore begun to consider structure queries on phylogenetic trees [27, 20]. [27] focuses on pattern match queries: given a query tree (sample phylogeny), find all trees that contain the query structure. Their technique is to decompose the pattern into a set of paths, and try to score the trees in the database with the number of matched paths. However, the method cannot be extended to structure queries whose input does not contain structural information, such as least common ancestor queries: given two species, find their least common ancestor. [20] focuses on least common ancestor and minimal spanning clade queries. By storing each tree edge as a tuple in a relational database, they can translate these queries into SQL expressions using transitive closure (provided by many relational systems, such as Oracle). A major shortcoming of this approach is that transitive closure can be very expen-

* This work was funded by NSF ITR EF 03-31654 entitled "BUILDING THE TREE OF LIFE: A National Resource for Phyloinformatics and Computational Phylogenetics".

sive for large data sets [28].

This paper presents a storage scheme and optimization techniques for efficiently supporting structure queries on phylogenetic trees. The structure queries supported include *least common ancestor*, *minimal spanning clade*, *tree pattern match*, and *tree projection*. Our method is based on a Dewey numbering scheme [30] which encodes the information of the path from the root to a node. Experimental results show that our approach performs well and scales to large data sets.

The outline and contributions of this paper are:

1. In Section 2, phylogenetic trees and structure queries are defined.

2. A labeling scheme based on structure information is presented in Section 3; a database schema based on this labeling scheme is then designed to store phylogenetic tree information.

3. Efficient algorithms for evaluating structure queries are presented in Section 3 using the proposed database schema.

4. Section 4 details the experimental results that demonstrate the efficiency of our strategy.

We close the paper by discussing related and future work.

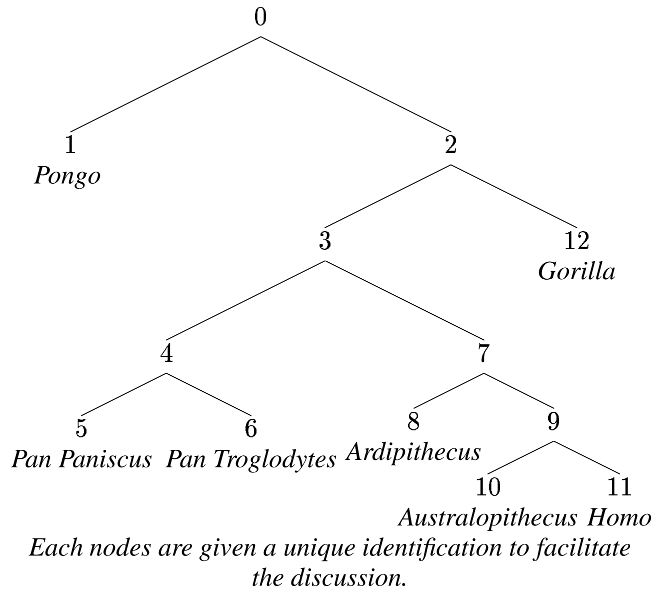
## 2 Data Model and Queries

A common data model for representing evolutionary relationships between species is a tree. Phylogenetic trees have the following special characteristics and requirements:

1) In theory, phylogenetic trees are unordered binary trees since it is almost impossible for a species to evolve into more than two species at the same time. Occasionally, trees with slightly larger fanout will be built if insufficient information is available; however, this is rare and the fanout is always small. Although it is important to determine if two nodes (species) have the same parent or ancestor in phylogenetic research, there is no obvious biological reason to sort the siblings. Trees are therefore considered to be unordered.

2) The leaves in a phylogenetic trees are tagged. The tag attached to a leaf is always unique, and typically denotes a species name.

3) The information attached to nodes is typically large, representing either sequence information (several million characters) or information about the model used to build this node in the tree.




*Each nodes are given a unique identification to facilitate the discussion.*

Figure 1: Phylogenetic tree for hominidae[14]

Formally, we can define a phylogenetic tree as follows:

**Definition 2.1:** A phylogenetic tree $T$ can be defined as a tuple($V$ $\Sigma$, $tag$, $parent$, root), where

• $V = V_I \bigcup V_L$ is a finite set of nodes where $V_I$ is the set of internal nodes and $V_L$ is the set of leaf nodes

• $\Sigma$ is a finite alphabet of node tags.

• $tag$:$V_L \to \Sigma$ is the *tag function*; $tag(n)$ returns the tag name of $n$, which can be either a tag or $\epsilon$.

• $parent$:$V \to \{\epsilon\} \cup V_I$ is the *parent function*; $\rho(n)$ returns the parent node of $n$ if it exists and $\epsilon$ otherwise.

• $root \in V$ is root of the tree. ■

Phylogenetic trees may also have more information associated with nodes or edges. For example, the edges may be weighted to represent evolutionary time. Here, we give a very basic model to simplify the presentation.

For example, Figure 1 shows the phylogenetic tree for Hominidae [14] where $tag(node_{11})$ = "Homo" and $parent(node_3) = node_2$.

Biologists frequently exchange and store phylogenetic trees using the NEXUS [15] format. In the NEXUS format, a pair of parentheses is used to rep-

resent an interior node, a string to identify a leaf node, a comma to separate two sibling, and an optional real number preceded by a colon to denote the weight of the incoming edge of the node. For example, the subtree rooted at $node_7$ in Figure 1 can be represented as (*Ardipithecus*, (*Australopithecus*, *Homo*))). Sometimes, biologists also want to identify the internal nodes and use an extended NEXUS format in which strings are also used to identify internal nodes. For example, if we use *id* to identify an internal node, the subtree rooted at $node_7$ in Figure 1 can be represented as (*Ardipithecus*, (*Australopithecus*, *Homo*)$node_9$)$node_7$.

We also define the ancestor relationship as follows:

**Definition 2.2:** Given a phylogenetic tree ($V$, $\Sigma$, $tag$, $parent$) and a node $n \in V$, $ancestor(n) = \{m|\exists l_1 \ldots l_k \in V \ (l_1 = n \wedge l_k = m \wedge \forall 1 \leq i \leq k-1 \ parent(l_i) = l_{i+1})\}$ The function $isancestor(m, n)$ returns true if and only if $m \in ancestor(n)$. ■

**Queries.**

Structure queries are used to determine relationships between species or to check if a given pattern exists in a given tree.

**Least Common Ancestor:** Least common ancestor is an important structure query on phylogenetic trees. Although it is not frequently used by biologist directly, it is the basic component for other structure queries. Least common ancestor finds the common ancestor of a set of nodes which is farthest one from the root.

**Definition 2.3:** Given a phylogenetic tree $T(V, \Sigma, tag, parent)$, the *least common ancestor* of a set of nodes $n_1, \ldots, n_k \in V$, denoted as $lca(n_1, \ldots, n_k) = \{m \mid isancestor(m, n_1) \wedge \ldots \wedge isancestor(m, n_k) \wedge \nexists l \ (isancestor(l, n_1) \wedge \ldots \wedge isancestor(l, n_k) \wedge isancestor(m, l))\}$ ■

For example, we may ask the following query:
$Q_1$ : Find the least common ancestor of *Homo* and *Gorilla*.
Using the tree in Figure 1, the result would be $node_2$.

**Minimal Spanning Clade:** Minimal spanning clade is often used when biologists want to find all species which are closely related to the species they are working on. The minimal spanning clade is defined as follows.

**Definition 2.4:** Given a phylogenetic tree $T(V, \Sigma, tag, parent)$ and a set of nodes $n_1, \ldots, n_k \in V$, the *minimal spanning clade*, denoted as $msc(n_1, \ldots, n_k)$, is the subtree rooted at $lca(n_1, \ldots, n_k)$. ■

For example, $Q_2$ : Find the minimal spanning clade of species *Homo*, *Gorilla* and *Pan Troglodytes*. The result is (((*Pan Paniscus*, *Pan Troglodytes*),(*Ardipithecus*, (*Australopithecus*, *Homo*))), *Gorilla*).

**Tree Pattern Match:** Tree pattern match is used when a biologist knows the relationships (a phylogenetic tree) between a set of species, and wants to find related research on this set of species. We define the tree pattern match as follows:

**Definition 2.5:** Given a query tree $Q(V', \Sigma', tag', parent')$ and a data tree $T(V, \Sigma, tag, parent)$, *tree pattern match*, denoted as $tpm(Q, T)$, return true if and only if there is a homomorphism from $V'$ to $V$. That is, there is a function $h : V' \rightarrow V$ such that:

- $\forall v' \in V' \ \exists v \in V, v = h(v')$
- $tag(v) = tag(h(v'))$
- $\forall w', v' \in V' \ w' = parent(v')$ $\rightarrow isancestor(h(w'), h(v'))$

■

For example, we may ask the following query:
$Q_3$ : Is the pattern ((*Gorilla*, *Ardipithecus*), *Homo*) in the tree of Figure 1?
The result would be false.

**Tree projection:** Sometimes, biologists are interested in a set of species, but may not know the relationships between them. In this case, they may go to a well known phylogenetic tree, such as TOL, and extract the subtree which only contains the relationship among the species they are interested in. We call this *tree projection* and define it as follows:

**Definition 2.6:** Given a data tree $T(V = V_I \bigcup V_L, \Sigma, tag, parent)$ and a set of nodes $S \subset V_L$, *tree projection*, denoted as $projection(T, S)$, returns a tree $(V' = V'_I \bigcup V'_L, \Sigma', tag', parent')$ such that $V'_L = S$ and there is a homomorphism $h : V' \rightarrow V$' such that:

- $\forall v' \in V' \ \exists v \in V, v = h(v')$
- $tag(v) = tag(h(v'))$
- $\forall w', v' \in V' \ w' = parent(v')$ $\rightarrow isancestor(h(w'), h(v'))$

■

Note that this differs from minimal spanning clade. For example, $Q_4$ : find the tree projection with given species *Homo*, *Gorilla* and *Pan Troglodytes*.
The result will be ((*Homo*, *Pan Troglodytes*), *Gorilla*) which is different from the result of query $Q_2$.

Since biologists are typically interested in a small set of species, the result of a structure query is usually relatively small.

## 3 Evaluating Structure Queries

Based on the properties of phylogenetic trees– in particular the fact that phylogenetic trees are unordered with small fanout – we use the Dewey numbering system [30] which is widely used in library book classification to label nodes and speed up structure queries [1].

The abstraction to phylogenetic trees is as follows: For each node $n$, we randomly order the outgoing edges and use the order as the label of the edge. Since there is a unique path $p$ from the root to a given node $n$, we concatenate the labels of edges appearing in $p$ and using the result string as the label for node $n$.

In this paper, we focus on a binary phylogenetic tree; however, the algorithms also hold for a fixed fanout tree. To clarify the following discussion, we introduce some terminology: Given a node id $n$, *label*($n$) denotes its label, *leftchild*($n$) denotes its left child and *rightchild*($n$) denotes its right child . Given a label $l$, *node*($l$) denotes the node id. Given two labels $l_1$ and $l_2$, *lcp*($l_1$,$l_2$) denotes their longest common prefix.

Ancestor/descendant relationship as well as common ancestors can be determined by comparing node labels as follows:

**Ancestor/Descendant** Node $m$ is a descendant of node $n$ if and only if *label*($n$) is a prefix of *label*($m$). Note that $n$ is the parent of $m$ if and only if its label is the string obtained by deleting the last character of $label(m)$.

**Common ancestor** A common ancestor of $m$ and $n$ has a label which is a common prefix of *label*($m$) and *label*($n$).

Note that a node label explicitly gives information of the path from the root to this node and therefore uniquely identifies this node.

**Example 3.1:** Figure 2 shows label information for the sample phylogenetic tree of Figure 1, where each tuple in the table corresponds to a node in the tree in Figure 1, the *tag* attribute represents the node tag information, the *id* attribute is the unique node identification and the *label* attribute in the table stores the label generated for this node. Consider species *Homo*, for which *label(Homo)*=010111. Since *label(Node*$_3$*)*= 010 which is a prefix of 010111, *Node*$_3$ is an ancestor of *Homo*, but not the parent. Furthermore, since *label(Pan Paniscus)*=01000, *Node*$_3$ is the common ancestor of species *Homo* and *Pan Paniscus* since 010 is a common prefix of 010111 and 01000. ■

[1] A similar scheme is also used in [21].

| *label* | *tag* | *id* |
|---|---|---|
| 0 | | 0 |
| 00 | Pongo | 1 |
| 01 | | 2 |
| 010 | | 3 |
| 0100 | | 4 |
| 01000 | Pan Paniscus | 5 |
| 01001 | Pan Troglodytes | 6 |
| 0101 | | 7 |
| 01010 | Ardipithecus | 8 |
| 01011 | | 9 |
| 010110 | Australopithecus | 10 |
| 010111 | Homo | 11 |
| 011 | Gorilla | 12 |

Figure 2: Relational Representation of Hominidae Phylogeny

**Algorithm 1** Tree: labeling(Tree: $r$, String local)

1: **if** r is null **then**
2:     return
3: **end if**
4: **if** *parent*($r$) exists **then**
5:     *label*($r$) = concat(*label*(*parent*($r$)), local) // concat is the function to concatenate two string
6: **else**
7:     *label*($r$) = local;
8: **end if**
9: labeling(*leftchild*($r$), “0”)
10: labeling(*rightchild*($r$), “1”)

The labels can be constructed in a single-pass using depth-first traversal of the input phylogenetic tree as presented in Algorithm 1.

Next we will discuss how to evaluate structure queries using this labeling scheme.

### Least Common Ancestor.

The least common ancestor of two nodes $m$ and $n$ can be answered by finding the node whose label is the

**Algorithm 2** Node: lca(Node: $n$, Node:$m$)

1: $l_n$= *label*($n$), $l_m$= *label*($m$)
2: Compute the longest common prefix $l$ of $l_n$ and $l_m$
3: Return *node*($l$)

longest common prefix of *label*($m$) and *label*($n$). Details can be found in Algorithm 2. Note that *lca(n, m)* can be computed in time proportional to the size of the labels of the input nodes, which are bounded by the height of the tree.

**Example 3.2:** To answer query $Q_1$, we will get the labels of *Homo* and *Gorilla*, which are 010111 and 011. The longest common prefix of these two labels is 01. We then determine that $node_2$ has label 01, so *Homo* and *Gorilla* shared the least common ancestor $node_2$. ■

**Tree projection.**

To find a tree projection from a set of nodes, we first get labels of these nodes and sort the nodes by their labels in alphabetical order. Algorithm 3 can then be used to projection the tree. The algorithm works as follows: Starting with an empty tree $T$, we insert nodes into the tree in order. Since the order of labels represents the left-right order of the leaves in the data tree, at each point the node being inserted will become the rightmost leaf node in $T$ after insertion. To determine the parent of the new node $n$ in $T$, we find the first node $m$ on the path from the current rightmost leaf node $r$ to the root such that *label*($m$) is a prefix of *label*($n$).

**Proposition 3.3:** Let $k$ be the number of nodes in the input set $S$. Then the total number of comparison performed by Algorithm 3 is bounded by $3k$.
**Proof:** Observe that each time we insert a node, the number of comparisons is just one more than the number of nodes removed from the rightmost path. So the total number of comparison = $1 + c_1 + 1 + c_2 \ldots + 1 + c_k$, where $c_i$ is the number of nodes removed from the rightmost path when we insert the $i$th node. Once a node is removed from the rightmost path, it will be never considered again. So $c_1 + c_2 \ldots + c_k$ is bounded by the size of the result tree which is at most $2k$. Thus, the total number of comparison performed by Algorithm 3 is bounded by $3k$. ■

**Example 3.4:** To answer query $Q_4$, we first retrieve the labels of the input species *Homo*, *Gorilla* and *Pan Troglodytes*, which are 010111, 011 and 01001, respectively. We sort this label set and get the list 01001, 010111, 011. We first insert 01001 into an empty tree. To insert 010111, we compute the least common ancestor of 010111 and the rightmost leaf in the current tree (01001) and get the result 010. Since 01001 has no parent, we use 010 as the root of the new tree and get the tree (01001, 010111)010. We then insert 011, and compute the least common ancestor of 011 and the current rightmost leaf (010111) obtaining 01. Since the parent of node 010111 has label 010 which is not a prefix of 01, we must continue up the path to the root of the current tree. However, node 010 *is* the root, so we must create a new root 01, finally, obtaining the tree ((01001, 010111)010, 011)01. Using tags to represent nodes, this is the tree ((*Pan Troglodytes*, *Homo*),*Gorilla*). ■

**Algorithm 3** Tree: projection(Node list: $S = (n_1, \ldots, n_s)$)

```
1:  T = null
2:  for i = 1, i ≤ s, i++ do
3:      if T is null then
4:          T = the tree with only one node n_i
5:          r = n // use r to record the rightmost leaf
6:      else
7:          lca = lca(r,n)
8:          m = parent(r)
9:          while m is not null and label(m) is not a
                prefix of label(lca) do
10:             m = parent(m)
11:         end while
12:         if m is null then
13:             leftchild(lca) = T
14:             rightchild(lca) = n
15:             T = lca
16:         else
17:             leftchild(lca) = rightchild(m)
18:             rightchild(lca) = n
19:             rightchild(m) = lca
20:         end if
21:         r = n
22:     end if
23: end for
```

**Algorithm 4** Tree: msc(Node: $n$, Node:$m$)

1: $l$ = lca($n$, $m$)
2: Get leaves of the tree rooted by $l$, $Leaves$, and order them by label
3: projection($Leaves$)

**Minimal Spanning Clade.**

Using the least common ancestor algorithm, we can find the minimal spanning clade as follows. Given two nodes $m$ and $n$, we find their least common ancestor $l$. We then find all nodes $a$ in the tree for which *label(l)* is a prefix of *label(a)*, obtaining as a result a set of nodes.

If the user wishes a tree instead of a set of nodes, we retrieve the leaves for which *label(l)* is a prefix and sort them by their labels. We then projection over them (Algorithm 3) to obtain the tree (see Algorithm 4).

In our implementation (see Section 3), we cluster nodes in a tree by their label and index labels so that matching nodes can be found efficiently. The number of comparisons performed is therefore proportional to the number of matching nodes plus an index scan. Since the result is already sorted, Algorithm 3 can be directly applied to the result to obtain the tree.

**Tree Pattern Match.**

To answer a tree pattern match query, we also use the projection algorithm (Algorithm 3). Given a tree pattern $p$, we extract the set of leaves in $p$. Using the set of leaves as input, we projection a subtree $s$ from the given phylogenetic tree $t$. We then check whether or not $p$ and $s$ are equal (in the case of an exact match) or compute the difference between $p$ and $s$ as a measure of similarity in the case of approximate match. Algorithm 6 shows the exact pattern match algorithm.

To check if two phylogenetic trees rooted at $m$ and $n$ respectively are the same, we use the property that the leaf tags are unique. The idea is that we perform a depth first traversal of each tree and tag each internal node with the smallest tag of its descendant leaves; this can be done in linear time. Then we compare the tags of the two trees starting at the roots: we first check if the tags of $m$ and $n$ are the same; if not, return false. If they are the same, we recursively check that for each child of $m$ there is a child of $n$ with the same tag (and vice versa). Since the tree is unordered and binary, this entails 3 checks. The total number of comparisons is therefore linear in the number of nodes in the tree. The detailed algorithm is shown in Algorithm 5.

To compute the difference between two trees, we refer readers to [2].

**Example 3.5:** To answer query $Q_3$, we first retrieve the leaves of the input tree pattern which are *Gorilla*,

**Algorithm 5** boolean: equal(Tree: $r_1$, Tree: $r_2$)

```
Function Tree: tagging(Tree: r)
 1: if R is a leaf then
 2:     return
 3: end if
 4: tagging(leftchild(r))
 5: tagging(rightchild(r))
 6: if tag(leftchild(r)) ≤ tag(rightchild(r)) then
 7:     tag(r) = tag(leftchild(r))
 8: else
 9:     tag(r) = tag(rightchild(r))
10: end if
Function boolean: Compare(Tree: r1, Tree: r2)
 1: if r1 is null then
 2:     if r2 is null then
 3:         return true
 4:     else
 5:         return false
 6:     end if
 7: else
 8:     if r2 is null then
 9:         return false
10:     end if
11: end if
12: if tag(r1) != tag(r2) then
13:     return false
14: end if
15: if Compare(leftchild(r1), leftchild(r2)) then
16:     if Compare(rightchild(r1),rightchild(r2))
        then
17:         return true
18:     else
19:         return false
20:     end if
21: else
22:     if Compare(leftchild(r1), rightchild(r2)) then
23:         if Compare(rightchild(r1), leftchild(r2))
            then
24:             return true
25:         else
26:             return false
27:         end if
28:     else
29:         return false
30:     end if
31: end if
Function boolean: equal(Tree: r1, Tree: r2)
 1: tagging(r1)
 2: tagging(r2)
 3: return Compare(r1,r2)
```

**Algorithm 6** boolean: pattern-match(Tree: $P$)

1: get the leaf set of $P$: $S$
2: Get the label of element in $S$ and order $S$ by label
3: $T$=projection($S$)
4: **if** equal($T$,$P$) **then**
5: return true
6: **else**
7: return false
8: **end if**

*Ardipithecus* and *Homo*. Applying the projection operation, we get a subtree ((*Ardipithecus*, *Homo*), *Gorilla*). Applying the tree equality function of Algorithm 5, we find that ((*Ardipithecus*, *Homo*), *Gorilla*) is not the same tree as the input pattern ((*Gorilla*, *Ardipithecus*), *Homo*), therefore, we return false. ■

## 4 Experimental Results

To evaluate the effectiveness of our method we built a prototype system using C++ and a leading commercial relational database system. We generated phylogenetic trees using r8s [25]. Based on Algorithm 1, a data loader parses the phylogenetic trees, generates a tuple for each node in each tree, and stores them in the database. The schema of the database is <*tid*, *label*, *tag*> where *tid* is used to distinguish different trees, *label* is the label of the node, and *tag* records the name of the species which is used to uniquely identify a node. The relation is clustered by {*tid*, *label*}. An index on {*tid*, *tag*} is also built to improve performance. We study the performance of our method and compare it with two related systems. Experimental results show the effectiveness of our approach.

### 4.1 Experimental Setup

The experiments were performed on a 1.5GHz Pentium 4 machine running Linux, with 512MB memory and one 40GB hard disk (7200rpm). The database is installed on another machine with the same configuration and running Windows 2000. All experiments were repeated 10 times, and the average processing time was calculated disregarding the maximum and minimum values.

We compare the performance of our system with two other systems processing structure queries: [20], which is based on the transitive closure primitive provided by the relational database and denoted as TC; [27], which is based on path match and denoted as PM. We denote our method as LS (labeling scheme).

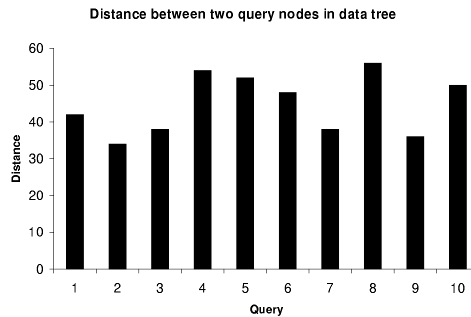



Figure 3: Distance between input pair nodes in the data tree

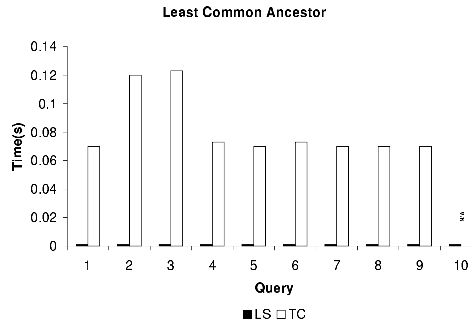



Figure 4: Execution time of LCA queries

Note that PM is a main memory algorithm, while TC and LS need to visit databases.

### 4.2 Least Common Ancestor (LCA)

The first experiment tests the least common ancestor query. Due to the lack of availability of large phylogenetic trees, in this experiment we use r8s [25] to simulate a phylogenetic tree with 0.5 million nodes according to a Yule stochastic process. The size of data and indexes for TC and LS are 35MB and 50MB respectively. We randomly pick 10 pairs of nodes as input; the distance between the node pairs is show in Figure 3. Figure 4 shows the execution time of least common ancestor queries; the database connection time is not included. Note that the PM method is absent in this test since it cannot support LCA queries. Here as well as throughout the rest of this section, whenever a method cannot support a particular query it will be omitted from the performance graphs.

We can see that both TC and LS work well. LS is based on string comparisons on labels which run very fast and cannot be accurately measured; we use 0.001 seconds as its running time. TC takes less than 0.14 second to run each query except for query 10. It is interesting to see that there is no clear relationship between the running time of the transitive closure

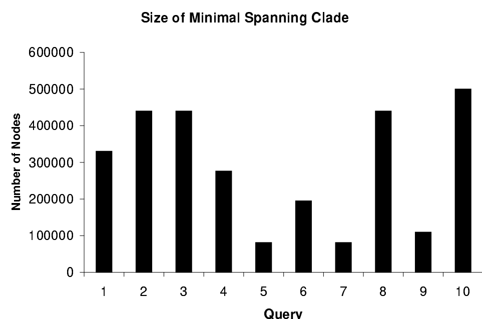



Figure 5: Number of nodes in the result MSC

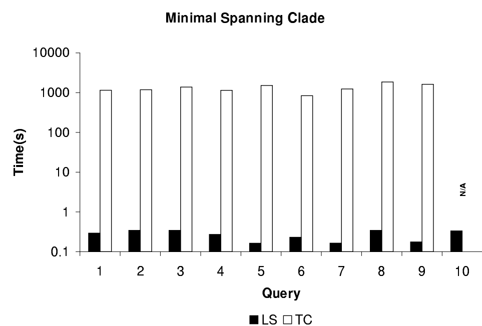



Figure 6: Execution time of MSC queries

method and the distance between two input nodes. For query 10, since the least common ancestor of the two given nodes is the root, and the root has not been stored as a tuple in TC, the result is not available.

## 4.3 Minimal Spanning Clade (MSC)

The next experiment tests the performance of minimal spanning clade queries. We use the same data and query set as LCA. The number of nodes in the resulting minimal spanning clade is shown in Figure 5. The query execution time is shown in Figure 6.

As we can see, except for query 10 which TC doesn't support, our method outperforms TC by several orders of magnitude: LS takes less than 0.35 second to find the result set while TC takes around 1000 seconds. It is curious as to why TC performs so differently for LCA and MSC queries. The reason is that in a tree, each node has only one parent, so transitive closure for LCA can be implemented as a set of recursive selections. However, since a node can have a set of children, the transitive closure for MSC must be implemented as a set of recursive joins, which are very expensive. Also the execution time of transitive closure has no clear relationship to the size of the query result.

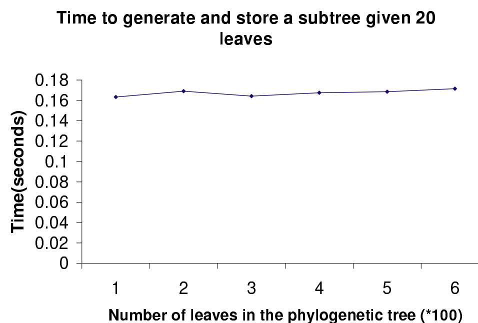



Figure 7: Time to projection a subtree with a given number of leaves from different sized phylogenetic trees

## 4.4 Tree projection

We also did two experiments to test the performance of our method on tree projection queries. Since TC and PM don't support tree projection, we only have one curve in the experimental results.

### Effect of varying the size of the phylogenetic tree

In the first experiment, our target is to understand how our algorithm scales. That is, to determine the effect on the execution time of projection when the number of input leaves is constant, the size of the phylogenetic tree varies, and the leaves are randomly chosen over the phylogenetic tree. To do so, we simulate phylogenetic trees with 100, 200, . . ., 600 leaves using HyPhy-II [24].

As shown in Figure 7, the time of projecting a subtree with a given set of leaves is not really affected by the size of the phylogenetic tree.

### Effect of varying the number of selected leaves

In the second experiment, we try to understand the effect on the execution time of projection as the number of leaves varies. To measure this, from a fixed phylogenetic tree we randomly select sets of leaf nodes varying the number of nodes. We use HyPhy-II [24] to simulate a phylogenetic tree with 2000 leaves as input and vary the number of leaves selected (10, 20, . . . 200 leaves).

In contrast to Figure 7, the time of generating a subtree with a given set of leaves is affected by the number of selected leaves.

## 4.5 Tree Pattern Match

In the last experiment, we test the performance of tree pattern match queries. We simulate a phylogenetic tree with 200 nodes using HyPhy-II [24] as the data tree. We randomly select 10, 20, . . ., 60 leaves and

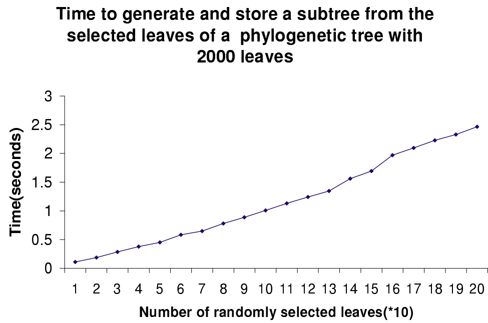



Figure 8: Time to projection a subtree with different number of leaves from a phylogenetic tree with 2000 leaves

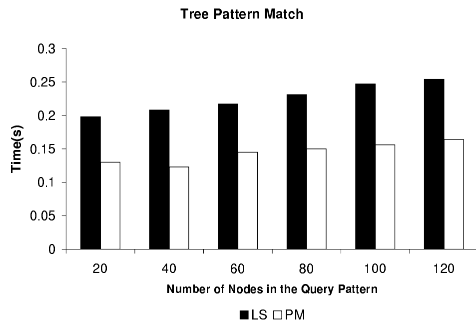



Figure 9: Execution time of pattern match queries

projection a set of subtrees. We use this set of subtrees as query trees. The result is shown in Figure 9. As we can see that, our method based on database engine is comparable to the main memory algorithm PM.

## 5 Related work

Several database systems [23, 14] have recently been created to store and retrieve phylogenetic trees. The Tree of life [14] is a resource which provides a uniform and linked framework to browse information on phylogenetic relationship as well as various characteristics of organisms, and provides links to related information available on the Internet.

TreeBASE [23, 22] uses a relational database to store phylogenetic trees and the data matrices used to generate them from published research papers. The phylogenetic trees themselves are stored as a BLOB attribute in NEXUS format [15] while other information (for example, the author of the tree) is stored as attributes. Keyword-based queries on attributes other than trees are supported. [27] proposes a method to extend TreeBASE to support structure-based queries (e.g. finding a particular tree pattern) by navigating the NEXUS format phylogenetic tree files using a general-purpose program language.

The next version of TreeBASE will enable structure-based queries by storing the tree structure explictly using the technique of [20]. By storing each edge in the tree explicitly, least common ancestor (LCA) queries can be computed using the transitive closure primitive supported in many commercial relational database systems.

[20, 21] also discuss the requirements of a phylogenetic tree database.

LCA has been well studied in the algorithms literature [8, 4, 3]. [8] describes the first linear preprocessing time, linear space, and constant query time algorithm for LCA. [7] observes that LCA is equivalent Range Minimum Query (RMQ) by giving a linear time algorithm to reduce LCA to RMQ using depth-first search, and a linear time algorithm to reduce RMQ to LCA using a cartesian tree construction. Based on the reduction of LCA to RMQ, [4] gives a linear preprocessing time, linear space, and constant query time algorithm to answer LCA. This algorithm is simpler than the algorithm in [8], and is in turn simplified by [3]. All of these algorithms need to randomly access several different data structures in an interleaved manner, and do not extend well to the database context.

Several techniques have also been developed for manipulating partially ordered sets and ontologies [11, 5, 16]. These techniques are good for processing small graphs (trees) in main memory, and have specialized operations for this application domain.

## 6 Conclusion and Future work

In this paper, we summarize several important structure queries on phylogenetic trees. Based on an analysis of the characteristics of phylogenetic tree and of structure queries, we proposed a storage system based on a Dewey labeling scheme. We then discuss how to efficiently evaluate structure queries. Our experiments show that this implementation using a relational engine has very good performance and scalability.

In ongoing research, we are investigating structure query operations among multiple phylogenetic trees, such as unions, difference and joins, and how to extend our techniques to support queries on more complex biological data, such as biopathways. In the future, we plan to investigate more general query languages which contain these basic operations. We are also interested in update operations on phylogenetic trees, permitting local rearrangements of phylogenetic trees, and facilitating the curation of phyloge-

netic data.

This work is being performed in the context of the Cyberinfrastructure for Phylogenetic Research (CIPRes) project funded by NSF (http://www.phylo.org/), and will be used for a massive simulation database representing a "gold standard" against which phylogenetic tree reconstruction algorithms can be tested.

## 7 Acknowledgments

We would like to thank Sampath Kannan for many valuable discussions, and ours reviewers for their constructive comments.

## References

[1] Assembling the tree of life. http://research.amnh.org/biodiversity/.

[2] N. Amenta, F. Clarke, and K. S. John, editors. *A Linear-Time Majority Tree Algorithm.*, 2003.

[3] M. A. Bender, G. Pemmasani, P. P. Sumazin, and S. Skiena. Finding least common ancestors in directed acyclic graphs. In *Proceedings of SODA*, 2001.

[4] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly Parallelizable Problems (Extended Abstract). In *Proceedings of STOC*, 1989.

[5] D. Connolly, F. van Harmelen, I. Horrocks, D. L. McGuinness, P. F. Patel-chneider, and L. A. Stein. Daml+oil (march 2001) reference description. http://www.w3.org/TR/daml+oil-reference.

[6] J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, Inc., 2003.

[7] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of STOC*, 1984.

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

[9] D. Hillis and J. Huelsenbeck. Support for dental hiv transmission. *Nature*, 369:24–25, 1994.

[10] J. Huelsenbeck and D. Hillis. Success of phylogenetic methods in the four-taxon case. *Syst. Biol*, 42:247–264, 1993.

[11] M. Install. Partially ordered set. http://mathworld.wolfram.com.

[12] J. Kim and T. Warnow. Tutorial on phylogenetic tree estimation. citeseer.nj.nec.com/254275.html.

[13] R. S. Lanciotti, J. T. Roehrig, and V. Deubel. Origin of the west nile virus responsible for an outbreak of encephalitis in the northeastern united states. *Science*, 286:2333–2337, 1999.

[14] D. Maddison and W. Maddison. The tree of life web project. http://tolweb.org/tree/phylogeny.html.

[15] D. Maddison, D. Swofford, and W. Maddison. NEXUS: an extensible file format for systematic information. *Syst. Biol*, 46:590–621, 1997.

[16] D. L. McGuinness and F. van Harmelen. Owl web ontology language. http://www.w3.org/TR/owl-features.

[17] K. McGuire, E. C. Holmes, G. F. Gao, H. W. Reid, and E. A. Gould. Tracing the origins of louping-ill virus by molecular phylogenetic analysis. *Journal of General Virology*, 79:981–988, 1998.

[18] B. Moret, J. Tang, L.-S. Wang, and T. Warnow. Steps toward accurate reconstructions of phylogenies from gene-order data. *J. Comput. Syst. Sci.*, 65:508–525, 2002.

[19] B. Moret, L.-S. Wang, and T. Warnow. Toward new software for computational phylogenetics. *IEEE Computer*, 35:55–64, 2002.

[20] L. Nakhleh, D. Miranker, F. Barbancon, W. H. Piel, and M. Donoghue. Requirements of Phylogenetic Databases. In *Proceedings BIBE 2003*, 2003.

[21] R. D. M. Page. Phyloinformatics: Towards a phylogenetic database. *Data Mining in Bioinformatics(in press)*.

[22] W. H. Piel, M. J. Donoghue, , and M. J. Sanderson. TreeBASE: A database of phylogenetic information. In *In Proceedings of the 2nd International Workshop of Species*, 2000.

[23] W. H. Piel, M. J. Donoghue, , M. J. Sanderson, M. Walsh, T. Eriksson, C. Henze, and K. Rice. Treebase: a database of phylogenetic knowledge. http://www.treebase.org/treebase/.

[24] S. L. K. Pond, S. V. Muse, and S. D. Frost. Hyphy. http://www.hyphy.org/.

[25] M. Sanderson. r8s. http://ginger.ucdavis.edu.

[26] M. Schoniger and A. Von Haeseler. Performance of maximum likelihood, neighbor-joining, and maximum parsimony methods when sequence sites are not independent. *Syst. Biol*, 44(4):533–547, 1995.

[27] H. Shan, K. G. Herbert, W. H. Piel, D. Shasha, and J. Wang. A Structure-Based Search Engine for Phylogenetic Databases. In *Proceedings of SSDBM*, 2002.

[28] S. Shi, E. Stokes, D. Byrne, C. Corn, D. Bachmann, and T. Jones. An enterprise directory solution with db2. *IBM Systems Journal*, 39, 2000.

[29] J. Tang and B. Moret. Scaling up accurate phylogenetic reconstruction from gene-order data. *Bioinformatics*, 19:305–312, 2003.

[30] V. Vesper. Let's do dewey. http://www.mtsu.edu/ vvesper/dewey.html.