PhyQL: A Web-Based Phylogenetic Visual Query Engine

Shahriyar Hossain, Munirul Islam, Jesmin, Hasan M Jamil

Integration Informatics Laboratory, Department of Computer Science, Wayne State University, Detroit, MI 48201

shah_h@wayne.edu, munirul@wayne.edu, jamil@cs.wayne.edu

Department of Genetic Engineering and Biotechnology, University of Dhaka, Bangladesh
mailjesmin@yahoo.com

Abstract

Biologists are often interested to query published phylogenetic data for research purposes. PhyQL, a web-based visual phylogenetic query engine, can be quite useful on this regard. In PhyQL, we have implemented a data model and a visual query language to interact with hierarchically classified tree elements. To hide textual query submission, PhyQL provides a design interface to build the query visually. The users can build simple to complex queries using the query operators. PhyQL separates the application layer from the data layer by a logic layer leading to reduced query tools development time. Moreover, PhyQL provides interactive tree views in radial, phylogram and dendrogram layout. It can be accessed online at http://integra.cs.wayne.edu/softwares/phyql/.

1 Introduction

Current genomic research is progressing quite rapidly due to advancement in bioinformatics and computation technologies. This is producing a great amount of data and revolutionizing the realm of phylogenetic studies. Most phylogenetic studies until recently have been restricted to 100 taxa or less [10]. Due to high throughput sequencing, large scale phylogenetic studies are now easily executable. As newer genomes are being sequenced, this trend of computing larger trees is going to increase. Hence, the need to store these trees for comparative exploration is greater than before. Thus we see a proliferation of a good number of phylogenetic databases (e.g. TreeBase [4], PhyloFinder [7]). These databases not only store the phylogenetic trees but also store the tree metadata (e.g. author, journal, year etc.). Due to the high volume of each database, advanced database technologies must be deployed in order to manage the storage, curation, retrieval and analysis of these biological data. The data is indexed and efficient algorithms are employed to reduce query execution times. As genomics and biology press forward to new frontiers, new paradigms must be developed to accommodate future needs. Therefore, we feel a strong need to propose a new technological framework with the ability to handle these data with better efficiency and higher abstraction of its execution.

Hence we propose PhyQL, a phylogenetic database with a visual query engine based on a visual query language [10]. PhyQL aims to overcome the following limitations of the previously developed systems:

- Accessing data through complex web forms limits user query capabilities. Because these interfaces do not allow the user to submit queries outside of its scope. Also, the queries do not consider trees as first class citizens. This seriously reduces the query expressiveness and is often frustrating from users perspective. Declarative queries have been very popular in relational databases based on select-project-join operations. Thus, a suitable phylogenetic query language will allow us to achieve the flexibility and efficiency supported in structured query languages.

- In TreeBASE and PhyloFinder, the structure queries are submitted in a parenthesized notation (newick [3] format). Given high nesting and large number of taxa, writing the structure query can be very hard. We feel that visual query interface is a convenient way to release the burden from the user.

- PhyQL returns a list of trees given a user specified query. Now, many of the trees are quite large and difficult to view in a single frame. So, it calls for interactive interfaces for visual exploration of the trees. Current phylogenetic databases do not offer flexible tree viewers and suffers from illegibility problems given very large trees.
2 Overview of PhyQL

PhyQL is a phylogenetic database. Its novel features are its visual query interface and interactive tree viewer interface. Its main function is to query phylogenetic trees. In [10], the user queries on a phylogenetic database were divided into three classes based on searching collections of trees and their internal relations; and searching individual trees. The classifications are Select, Join and SubSet queries.

3 Implementation Details

PhyQL is a web tool written in Java since Java is open source and portable. For visualization, we used Java Universal Network/Graph Framework (JUNG) [2] and PREFUSE [9] Library. The database was stored in MySQL 5.0.22. The datalog queries were tested using XSB. Besides we used J2EE to build the client-server platform for web access. The client side was tested using Java Applet so that an user can use database online from any Java-enabled web browser.

3.1 System Architecture

Initially the user creates the visual query combining query operators. The user query tree is first translated to an xml document. This translated document is then submitted to the Logical Query Engine, which is the heart of the system. Its job is to translate visual queries into logical queries. The Logical Query Engine communicates with the database, fetches the trees from the databases and finally sends the translated trees in GraphML [1] notation to the tree viewer. The export module is responsible for downloading the trees in XML or NEXUS [11] format by the user. For test purposes, we downloaded a 2004 TreeBASE image from TBMap (http://flinnaeus.zoology.gla.ac.uk/~rpage/tbmap/) [13].

3.2 Language

The query operators in the graphical query language can be combined together to create from simple to complex forms of queries. For a full review of the operators, we refer the interested reader to [10]. The operators we implemented are:

- **Root**: The Root operator extends the internal node operator. It specifies an internal node which is also the root of the tree.
- **Leaf**: The Leaf operator is used to specify a LEAF node of the query tree. In [10], Jamil et. al. mentioned applying associated node properties along with the LEAF operator: taxa name, alias, and taxa description.
- **Internal Node**: It is depicted as a ‘?’ . This operator identifies the LCA of one or more nodes. In Figure 2 (a) node c, d and the parent of node a and b have an LCA which is the child of the root of the tree. In the special case of having only one child, this operator reduces to the ancestor.
- **Subtree**: Given a set of labeled nodes, this operator extracts a projected tree from an existing tree. The algorithm for the Subtree operation is very simple. From the set of labeled nodes, it builds a subtree up to their LCA. Each internal node in a phylogenetic tree must have at least three edges connected to it. So, any internal node in the initial projected subtree is skipped that doesn’t meet this criteria.
- **Join**: The join operator is defined as a function that considers two nodes from two trees as join parameters. For node joining, only the equality (=) condition on node labels is considered. If the joining condition fails, an empty tree is returned. For example, two trees can be joined together where the leaf of a tree and the root of another tree have the same label.

3.3 Translator

The translator is the heart of the system. It translates the visual query tree to an XML document. Using XML, we can pass additional constraints on the nodes using attributes values. For example, consider Figure 2 (a) is translated to the XML document depicted in 2 (b). For every node in the visual query, we create an element in the XML with the type attribute of the element specifying the visual operator. We
also pass the tree-wide query attributes such as “author” as an attribute of the root element of the XML. We convert this XML into a mixture of relational and logical queries. Let us consider the logical query first. In this stage, we traverse the nodes of the query tree in post order and come up with a logical conjunction which will be

\[
\text{isRoot}(X) \text{ is true if } X \text{ is the root of a tree.}
\]

The first two of the above mentioned predicates are directly mapped to the database facts. The rest are derived from them. We will explain the translation process by following each step for the example in Figure 2. Let us consider the lowest level of nesting which is a leaf node. For example,

\[
\text{isRoot}(Y) \text{ is true if } Y \text{ is the root of a tree.}
\]

is translated into the query predicate \( \text{node}(Y, 'a') \). Like in Prolog, the logical variable \( Y \) is system generated and is bound to ‘a’. The query string is passed to its parent node ‘?’.

\[
\text{isRoot}(X) \text{ is true if } X \text{ is the root of a tree.}
\]

This gets translated into \( \text{edge}(Y, Y_6), \text{edge}(Y, Y_5) \). The temporary variables \( Y_4 \) and \( Y_6 \) are the logical variables associated with node \( a \) and \( b \). Now, we can rewrite the ‘*’ node in the following manner:

\[
\text{node}(Y_5, 'e'), \text{edge}(Y_0, Y_1), \text{node}(Y_3, 'c'), \text{node}(Y_5, 'b'), \text{edge}(Y_4, Y_5), \text{node}(Y_6, 'a'), \text{edge}(Y_4, Y_6), \text{lca}(Y_3, Y_4, Y_2), \text{node}(Y_7, 'c'), \text{lca}(Y_2, Y_7, Y_8), \text{edge}(Y_0, Y_2), \text{isRoot}(Y_0)
\]

Finally, we can rewrite the sub tree of the root replacing the leaf ‘e’ and the ‘*’ node with their corresponding logical variables.

\[
\text{node}(Y_1, 'a'), \text{edge}(Y_0, Y_1), \text{edge}(Y_0, Y_2), \text{isRoot}(Y_0)
\]

And, the translation for the root would be \( \text{edge}(Y_0, Y_1), \text{edge}(Y_0, Y_2), \text{isRoot}(Y_0) \). The entire logical query for this example is stated in the Figure 2 (c). After we have gathered all the trees that return true for this logical query, we apply a simple relational selection on them based on the author name.

\[
\text{node}(Y_5, 'e'), \text{edge}(Y_0, Y_1), \text{edge}(Y_0, Y_2), \text{isRoot}(Y_0)
\]
3.4 Logical Query Engine

The query engine receives logical queries from the translator module described in the previous section. One of our goals was to separate the rule engine from the database engine. We chose XSB [15] as our Logical Query Engine. XSB’s in-memory database queries are an order of magnitude faster than tuProlog [14] and DataLog [6]. Using logic predicates, we could reduce the time complexity for LCA to $O(h)$.

4 Related Work

We will mainly compare our work with TreeBASE and PhyloFinder. Though TreeBASE maintains a relational database, it stores the trees in newick format. To overcome this limitation, Nakhleh et. al. [12] proposed storing trees in an edge table and use Datalog predicates to recursively query the relational database. But, they could not devise any logical query engine to execute the predicates directly. They converted the transitive closure and LCA predicates into SELECT statements with the CONNECT BY primitive of ORACLE which is not part of the SQL standard. These theoretical Datalog rules had several limitations. Firstly, they proposed a top-down rule for finding the ancestor of two nodes. This may result into a time complexity of $O(h^2)$ which is clearly costlier than our $O(h)$ approach. The Least Common Ancestor (LCA) query proposed in [12] suffers from using the not operator.

To reduce the time complexity in LCA queries, PhyloFinder preprocesses the trees and store additional labeling information along with a node. In [8], Davidson et. al. proposed storing dewey labeling scheme [5] in the nodes. On the other hand, the Crimson [16] system eliminates this problem by storing the labels in nested subtrees to avoid long chains. But these labeling procedures are not suitable for dynamic environments. Because for dynamic trees, the tree nodes must be re-labeled each time the tree structure changes. PhyQL eliminates this problem by recursively computing the LCA for a set of nodes. Thus it can easily handle static and dynamic trees equally.

5 Conclusions and Future Research

PhyQL offers a simple web-based visual query interface based on a phylogenetic query language. Thus the user can concentrate on the query semantics rather than its syntax. The tree query operations are fully logic-based in PhyQL. Any modifications to query tools only requires change in logic rules drastically reducing program development time. The proposed architecture for PhyQL can be applied not only to the phylogenetic trees but also to protein-protein interaction networks, metabolic pathways etc.

References