A Framework for Querying Pedigree Data

B. Elliott, S.F. Akgul, Z.M. Ozsoyoglu, E. Manilich
Case Western Reserve University
{bxe7, sfa3, mxo2, eam15}@case.edu

Abstract

Genealogy information is becoming increasingly abundant in light of modern genetics and the study of diseases and risk factors. As the volume of this structured pedigree data expands, there is a pressing need for better ways to manage, store, and efficiently query this data. Building on recent advances in semi-structured data management and proven relational database technology, we propose a general-purpose Pedigree Query Language (PQL) and evaluation framework for elegantly expressing and efficiently evaluating queries on this data. In this paper, we describe how the problem of modeling and querying pedigree data differs from XML, present an overview of PQL, and present efficient evaluation for key parts of the language. Experimental results using real data show significant (>850%) performance improvement for complex queries over naïve evaluation.

1. Introduction

A pedigree is a hierarchical structure used for expressing and/or visualizing the hereditary relationships between individuals, often including known phenotypes for these individuals (see Figure 1). This data is particularly important in biology and medicine for understanding the hereditary behavior of certain diseases and traits. In medicine, using the pedigree structure, disease risk calculations for an individual can be performed, the possible risk factors for a newborn can be predicted, or the effects and characteristics of a certain disease can be analyzed and the location of the disease causing mutation can be spotted.

Collecting large amounts of pedigree data can be a difficult and tedious task and most of the available data was once rather small, making it feasible to perform analysis using straightforward methods. Due to this lack of data, the problem of storing, querying and visualizing large amounts of pedigree data efficiently, hasn't received much attention. The available pedigree data management systems are composed of a graphical interface built on top of an ad hoc database structure without taking efficiency and scalability into consideration. However, some projects currently in development, such as the Cleveland Clinic Colon Cancer Patient Pedigree data [5] and the Utah Population Database [11], involve large amounts of patient pedigree records. The Utah Population Database includes 1.6 million genealogy records, and a total of 8.7 million records including birth, marriage and death certificates, and cancer records. The Polyposis Registry at the Cleveland Clinic includes pedigree data and medical conditions that may be related to cancer for families (~750) with a rare inherited colorectal cancer syndrome. The rarity of the syndrome and importance of comparative analyses of familial relationships signify the need for a pedigree query language, even for relatively modest-sized pedigree data repositories such as [5]. A pedigree query language that can precisely express queries on patterns of familial relationships and their associations to phenotypic features and genotypes is important and may lead to new discoveries.

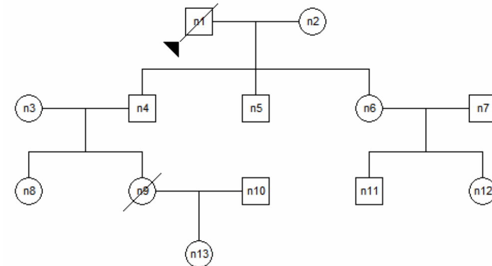

Figure 1: Sample pedigree tree

Q1. Find all ancestors of an individual.
Q2. Find all first-degree (FDR) and second-degree (SDR) relatives of a proband individual who were diagnosed with cancer before age 50.
Q3. Find all individuals with three or more ancestors diagnosed with cancer on the paternal side of the family, but no diagnosed ancestors on the maternal side.
Q4. Find all individuals marked as probands with at least 2 descendants diagnosed with at least three polyps before age 35.
Q5. Display all family members at high risk based on Church's clinically defined scoring system (number of FDR, SDR, age) [4].
Q6. Display all family members or families diagnosed with certain type of extra-colonic cancer after certain surgical procedure.
Q7. Find all individuals with the mutation at codon 1556.

Figure 2: Sample pedigree queries

Some sample pedigree queries for colon cancer data are listed in Figure 2. The desirable properties of a

Proceedings of the 18th International Conference on Scientific and Statistical Database Management (SSDBM'06)
0-7695-2590-3/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

pedigree query language include (i) expressive power and (ii) intuitive syntax and semantics. That is, a pedigree query language should be powerful enough to express the set of meaningful queries that can be defined over the pedigree data in an easy to understand way. A good approach to accomplish (ii) is to use the syntax of commonly used data query languages such as SQL, XPath [20] or XQuery, and defining the axis names in an intuitive way.

In addition to defining a pedigree query language, there is a need for a query evaluation strategy that supports the axis definitions of the pedigree query language and enables efficient evaluation of complicated queries on large amounts of pedigree data. The contributions of this paper can be listed as follows:

i. We use a directed acyclic graph to model the pedigree structure, where each node is annotated with gender, and has at most two incoming edges (i.e. parents).
ii. We give a comparison of hierarchically structured pedigree data and XML data in terms of structure and querying to see to what degree existing techniques and support for XML querying can be utilized.
iii. We define PQL, Pedigree Query Language, for expressing queries on pedigree data.
iv. We describe an efficient query evaluation strategy for PQL queries using labels (nodecodes) on the pedigree data.
v. We also demonstrate the efficiency of our pedigree query processing system by experimental results, where significant improvements are observed over the naïve approach for pedigree query evaluation.

2. Previous Work

To the best of our knowledge, there is no modern query language supporting efficient representation and evaluation of complex pedigree queries. Existing systems are either designed purely for visualization, support only simple queries with no method to compose them into richer queries, evaluate queries using naive iterative SQL calls, or are simply obsolete. In this section, we review some of these related systems and software for managing, viewing, and/or querying pedigree data.

Recent related work includes PViN [18] and Peditree [17], both of which are designed to display large pedigrees from relational databases, with PViN focusing on the use of modern graphics capabilities for fast drawing, and PediTree providing support for some calculations on the data (such as inbreeding estimation). However, the most complicated query described by both systems is simply retrieving all ancestors/descendants of an individual by iteratively performing an SQL query for every individual ancestor/descendant to check for additional relatives.

Other academic work includes PedHunter [1], which focuses on the task of verifying pedigrees and to that end provides functions (each using C language and one or more SQL queries) for finding relatives and for testing how sets of individuals are related, but are not convenient or efficient for constructing longer path queries. [19] describes how to answer a couple of structural queries using SQL, but it involves explicitly creating temporary relations and is overly verbose. One of the more relevant works [7], describes an early SQL-like language supporting path expressions, but it only supports the most basic steps, the evaluation is largely based on pointers and the queries end up being relatively verbose.

There are also several commercial software packages available, the most popular of which are Cyrillic [6] and Progeny [12]. Progeny 6 only supports simple Boolean queries on data fields that can only include relationships relative to the proband, or interactively selecting related individuals on the pedigree graph with no ability to specify additional conditions. Cyrillic 2.1 can calculate inbreeding and crossovers from phenotype data, but provides no support for structure-based querying.

3. Pedigree Modeling

A *pedigree* can be defined as "a simplified diagram of a family's genealogy that shows family members' relationships to each other and how a particular trait or disease has been inherited." [21].

We represent a pedigree as a directed graph composed of nodes and edges. In a *pedigree graph*, each *individual* (i.e. biological organism) in the pedigree is represented by a node, and nodes are connected by *directed* edges from parents to children. From a genealogical perspective, each individual has *exactly* two biological parents (one female and one male), but since pedigree data is usually incomplete, we instead state that, a node can have *at most* two parents (i.e. incoming edges). Unlike [7], we model parent-child relationships directly instead of using another type of node to represent them (i.e. a "marriage" node) in order to keep the graph representation uniform and keep the query language straightforward. Furthermore, the constraints placed on pedigree graphs give them special properties compared to general graphs, which we will use later. More specifically, a *pedigree graph* is a *directed acyclic graph* (DAG) where each node has indegree of *at most*

IEEE COMPUTER SOCIETY

2. In a pedigree graph, the nodes with no incoming edges (i.e. with indegree zero) are called *progenitors.* A pedigree graph has one or more progenitors.

In addition to ancestor/descendant relationships, our model also includes data associated with each individual. All Individual nodes have two required attributes: *Gender* (*Female, Male,* or *Unknown*) and an arbitrary unique identifier, *ID*. In addition to simple attribute fields, associated data can include structured XML data or associated relational tables. We will describe an intuitive extension to our language that uses XPath to search XML data associated with an individual in [8]. Even if the data is relational, XPath may still be used via a standard relational mapping, such as that described in [13].

4. Comparison with XML

Pedigree data resembles XML data in many ways. Pedigree structures are hierarchical, elements are connected with parent-child relationships and queries are defined on the structural relationships between the elements in the pedigree data. However, there are many crucial differences between pedigree data and XML data. These differences prevent the application of XML storage, labeling, indexing and analysis for pedigree data with straightforward modifications. In this section, we will analyze the differences between the two in terms of data structure, query expression and query evaluation.

Structural Differences: XML data has a strict tree structure with the exception of optional id and reference pointers. Every XML node has at most *one* parent node and any number of child nodes where each node represents an element, attribute, text, comment, etc. On the other hand, a pedigree graph is actually a *directed acyclic graph with at most two incoming edges per node* [8]). Since this is a more general structure than a tree, tree-based labeling and indexing schemes developed for XML (such as [2]) data cannot be used for pedigree data in the same way; new techniques are required. Note that since a pedigree graph is not even a planar graph the labeling techniques for planar graphs [15] cannot be used either.

Furthermore, XML element nodes are labeled by tags that can be used to distinguish nodes, which is queried along with the relationships between nodes in the tree structure. However, in pedigree data, there are no tags, but only an identifier and gender that must be associated with each node, although additional attributes and data may be present. In addition, while nodes in XML documents can have huge branching factors based on the type of data represented, in many organisms, such as humans, the pedigree structure typically has a very limited branching factor.

Query Differences: We can say that pedigree queries in general have two components – the part on the pedigree structure and the part on the data stored for each individual. For instance, the query Q2 "*Find all first-degree (FDR) and second-degree (SDR) relatives of the proband individual who were diagnosed with cancer before age 50*" (Figure 2) involves finding related individuals using the pedigree structure and then using the data stored for individuals to find out if they were diagnosed before age 50. Furthermore, this brings additional challenges and alternatives for a query optimizer trying to find efficient evaluation strategies.

Since there are no tags to utilize in pedigree data, structural queries are defined solely on relationships. This eliminates an important part of the query selectivity found in XPath, and query performance may become harder to deal with. XML queries are generally defined in one direction, which is from ancestors to descendants. While the ancestor and parent axis steps are available in XPath, they are not commonly used in practice. On the other hand, pedigree queries are often defined in both directions, as it is common to query for the parents of an individual, or his or her children, or the family altogether.

In general, pedigree queries can be categorized according to whether they focus on relationship structure or associated data. If we re-examine the queries in Figure 1, we can see that Q1-Q4 rely heavily on pedigree structure, while Q5 makes implicit use of it. However, on the other hand, Q6-Q7 are independent of the relationship structure, and so can be solved with an existing query language (such as SQL) as well as with a query language designed for pedigrees. Q3-Q5 are at the same time aggregate queries using aggregate functions (COUNT, SUM, etc.).

5. Query Language

In this section, we present an overview of our pedigree query language, PQL, which is a path query language, inspired by XPath ([20]). Its primary purpose is to provide a compact and easy to understand syntax for querying both the structure (i.e. parent-child relationships) of pedigree data and associated fields. It allows relative addressing of individuals in a pedigree via composing a series of one or more relationship *steps* into a rich path expression. Similar to XPath, expressions are evaluated to return either *sets* of individuals, a basic value (number, text, Boolean), or a

IEEE COMPUTER SOCIETY

set of basic values. For a more formal language specification for PQL, please refer to [16].

Query starting steps: There are two fundamental ways to start path queries: (i) making a general query that finds all individuals in the database who meet certain conditions, or (ii) making a query relative to a specific individual. General queries begin with the "Individual" starting step, representing *any* individual, and the simple query of "Individual" by itself returns the set of all individuals in the database. If there is a specific context for a query, then the query begins with the individual's unique ID or the shortcut "." to represent a set of one or more selected context individuals. These will be shown in later examples.

Basic axis steps: *Axis steps* (or simply *steps*) are used to navigate the relationship structures of pedigrees and may be combined to form a *path*. We begin by defining the fundamental steps that form the basis of more complex steps. Conceptually, from an initial set of one or more individuals, we find the set of all individuals who are related to the previous set in one of the specified manners. The basic steps are: Parent, Child, Ancestor, Descendant, and Self. Much like folder paths, a slash "/" is used before each step in PQL. Thus, the query "Individual/Parent" finds all the individuals who are parents and "ID123/Descendant" finds all the descendants of the individual identified as "ID123". Where the Parent and Child steps return the related individuals one step above or below in the pedigree graph, the Ancestor and Descendant steps return *all* the appropriate individuals, regardless of the number of generations that separate them. The Self step returns the same set of individuals as in the previous step (so "Individual/Self" returns the same results as "Individual") and is needed for some advanced queries.

Gendered steps: While the Parent and Child are fundamental steps, it is very common for pedigree queries to involve gender. PQL supports these queries in a compact way by adding a gender specifier as a postfix to any step. In genetic data, an individual's gender may be Female, Male, or Unknown. These are represented by the postfixes "{F}", "{M}", and "{U}", with an additional gender postfix of "{A}" (for *any* gender) that is implied automatically if one of the others in not specified. For example, the query "ID65 / Parent{F}" returns the mother of individual ID65 and "./Child/Child{M}" returns all the male grandchildren of the selected individual(s).

Simple Attributes: In pedigree databases, there are typically many additional fields/features that may be specified for each individual, which are called *attributes* in PQL. Each attribute has a unique name (such as "Name", "Birthday", etc.) and has an associated basic value (number, text, date, etc.). Attributes are written as "@" followed by their name (i.e. "@Birthday") and appear at the end of a path expression. For example, "ID78/Parent/Parent{M} @Name" returns the names of ID78's grandfathers and "Individual@Birthday" returns the birthdays of all individuals. Note that as data may be incomplete, many attributes may be missing, so the previous query has the semantics "the birthdays of all individuals with a birthday attribute specified."

Conditional Steps—predicates: PQL allows optional Boolean expression predicates to be specified after each step as an additional filter after the step name which is written as "[...]". Any attributes or additional path expressions inside this predicate is *relative to the step that immediately proceeds it*, so "Individual [@Name = 'Bob'] / Parent [@Age > 35]" returns the parents of individuals where the individual’s name is 'Bob' and the parents are more than 35 years old. In addition to Boolean expressions involving attribute values (as seen in the previous example), predicates may also check for the existence of an attribute or path expression. Existence conditions are implied by the lack of a comparison operator such as "=", ">". For example, "ID23/Descendent[@Birthday]" returns all descendents of ID23 for whom a birthday is known and "Individual{M}[/Child/Child{F}]" generates all men who have a granddaughter. Note that conditional predicates can appear on a step that is inside another conditional predicate.

PQL supports the use of many of the typical expressions found in other languages' predicate filters for conditional steps. Expressions may include Boolean logic (AND, OR, NOT), text string manipulation (concat, substring, contains, etc.), common mathematical operators/functions (+, *, -, ceiling, etc.), parentheses, and, of course, path expressions. For example, "Individual [@Proband = True] / Parent {F} [@Age > 45 AND (@HasCancer = True OR @HasPolyps)]" finds mothers of probands who are older than 45 years and have been diagnosed with cancer or have polyps.A general expression may also be evaluated as a query, and can return simple strings, numbers, or Boolean values in place of a set.

Set expression steps: There are a variety of advanced queries that effectively need to follow

IEEE COMPUTER SOCIETY

multiple types of paths at the same time. A simple example is the query Q="find all people affected by a genetic disease *along with* all their descendents". The result of this query is a union of the results of the "/Descendant" and the "/Self" steps. For this purpose, the XPath language defines a special "descendant-or-self" axis step; there are fewer "frequently used' axis steps in XPath, so it is easier to pre-define the combinations like this. However, to compactly represent some pedigree queries, it is convenient to be able to specify several alternative steps to follow at the same point in the query in a flexible manner. For example, finding all first degree relatives is equivalent to finding parents, children and siblings and we would like to do this in a single step. To allow this, PQL allows expression steps, which are represented as "/(...)", which replaces a single step name with a parenthesized path set expression. Note that the result of any step in PQL is a *set* of individuals. First, we introduce the union (or alternation) operator "|", which defines the next step as the set represented by following both steps *simultaneously*. Therefore, "/StepA/(StepB | StepC)" is a compact way to represent "/StepA/StepB" union "/StepA/StepC".

Example 5.1. Using a set expression for steps (with union operator), the query Q above can be expressed as Q= Invididual [@HasDisease = True] / (Self | Descendant). Similary, the query "all the first degree relatives of ID45" can be expressed as Q'= ID45/(Parent | Child | Sibling), where the Sibling step will be defined below.

In addition to union, set expression steps may also contain *set difference*, which is represented as "-". This allows us to define the /Sibling step ("my parent's children besides myself") in terms of Parent, Child, and Self steps using the following step expression: "/(Parent/Child - Self)". Note that any path that returns a set of individuals can be used as an operand inside a step expression which is a set expression. Additional parentheses may be optionally used to form more complex expressions, but the outermost parentheses are mandatory. While union and set difference operators seem to be all that is required for most reasonable queries, the language could include set intersection ("StepA & StepB") and set complement ("!StepA") operators as well in step expressions.

User-defined (macro) steps: For efficiently expressing common relationship steps and starting points, PQL allows special *macro*, or short-cut, steps to be defined and expanded automatically when the query is evaluated. Similar to "/Sibling" step defined as "/(Parent/Child - Self)", "/Mother" and "/Grandfather" defined as "/Parent{F}", and "/Parent/Parent", respectively are examples of user defined steps. Another example is the *proband*, who is the individual within a pedigree who was first identified with the disease. For disease data, it is common to start queries at the proband, which can defined as the macro starting point as "Individual[@Proband = True]".

Aggregate Functions: Aggregation functions Count(), SUM(), AVG(), MIN(), MAX(), etc., can be used as usual. For example, "COUNT (ID12 /FirstDegreeRelative [@HasCancer=True])" returns the number of first degree relatives of individual ID12 who have been diagnosed with cancer. Similarly, "AVG (Individual{M} [@HasCancer=True]@Age)" finds the average age of males diagnosed with cancer.

Aggregate functions can also be used in conditional predicates. For example, "Individual{M}[COUNT (/Ancestor[@HasCancer=True]>3)]" returns male individuals who has more than 3 ancestors with cancer.

Combined use with XPath: Data associated with an individual is often more complex than simple attributes, and may require several additional relational tables or XML-structured data to fully represent. For medical data, this may include details about surgeries, medications, complex diagnoses, etc. If we assume that there is an (optional) XML document associated with each individual, then we can use standard XPath [20] to query it. In this case, the first step beyond an Individual node in a PQL expression is the document root of an associated XML document (or the name of the associated document), and further steps are within the XML document using XPath. In contrast to XPath, every PQL step returns an Individual node, thus it is meaningful to allow XPath queries to be appended at any arbitrary PQL path expression.

Example 5.2. Consider an XML document PatientData including details of surgeries for each individual. The following PQL/XPath query "ID39 / Descendant [/PatientData.xml / Surgeries/Surgery [@Type='Partial Colectomy']]" returns all individuals who are descendants of individual ID39 and have had a partial colectomy. In this case, "/Surgeries/Surgery [@Type='Partial Colectomy']" is evaluated as an XPath expression using tag-based child steps on the XML data of each descendant of ID39 (who were found using PQL).

As illustrated in Example 5.2, combined PQL/XPath expression allows predicate conditions on complex metadata to be expressed succinctly. There are, however, a couple of restrictions on XPath queries embedded within a PQL query. XPath queries

Proceedings of the 18th International Conference on Scientific and Statistical Database Management (SSDBM'06)
0-7695-2590-3/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

attempting to use the parent axis step ("..") to traverse back above the document into the pedigree graph are not supported. In addition XML documents associated with an Individual node *cannot* use PQL step names as the document root (the tags specified would be interpreted as a pedigree graph structural step, not an XPath step). Again, if the data is relational, we can map it to XML [13].

6. Query Evaluation

In this section, we describe an approach for PQL query evaluation by using a PQL to SQL transformation, and evaluating the SQL query on relational tables. The hierarchical structure of pedigree data is encoded using a labeling method so that it can be utilized effectively for the evaluation of queries that require matching structural patterns.

The bottleneck for evaluating PQL queries efficiently is the implementation of the parent/child and ancestor/descendant steps. We use a labeling method that allows these basic relationships on pedigree graphs to be decided with a single label comparison step. More complicated structural relationship can be expressed in terms of these basic steps, combined with gender predicates, which can be processed as a simple attribute test on an Individual node or via labels. The XPath sections of the queries are processed according to the XPath Language Specification and evaluating these XPath fragments is outside the scope of this paper. Any XPath evaluation system, such as [2], can be used with the pedigree querying system.

Labeling for PQL- NodeCodes for Pedigree graphs: Evaluation of queries involves traversing the pedigree graph, which may be costly for some queries. For example, a query involving all male ancestors of a given individual may require several joins to evaluate using the parent information stored with each individual. In order to expedite such queries requiring graph traversal, we use a labeling system based on [BBL, SOO] called NodeCodes for pedigree graphs. In the nodecode system, unique codes are assigned to every node, so that, given the nodecode of a node (individual) in the pedigree graph, the form of the nodecodes of all descendants/ancestors can be determined without traversing the graph. Similarly, given nodecodes of two nodes, one can tell if there exist paths between these two nodes, and if there exist, all the connecting paths can be derived from the nodecodes without graph traversal.

Nodecodes were originally defined for graphs with a single source node and were binary strings. Since a pedigree may have several progenitors (nodes with in-degree 0), consider adding a virtual source node R, and make all the progenitors in the pedigree children of R. We also use nodecodes, which are sequences of integers and delimiters. The integers denote the sibling order, and the delimiters denote the generations, as well as indicating the gender of the node. The delimiter "*" could be ".", ",", or ";" denoting female, male or unknown respectively. Then for each node u in the graph nodecode(u), is assigned using a depth-first-search traversal starting from the source node as follows:

- If u is the source node, then nodecode(u)={the empty string}.
- Let u be a node with nodecode x, and v_0, v_1, … v_k be u's children in sibling order, then nodecode(v_i) = xi*, where $0 \le i \le k$.

Note that, the virtual source node is not actually used, and the progenitors of the pedigree have nodecodes i* where i=0,1,…, where order indicates the age of progenitors, 0 is oldest, and * indicates gender. Figure 3 shows a pedigree graph labeled with nodecodes. Below are some properties of the nodecode system, which follow from the definition [3, 14].

Lemma 1. A nodecode can be assigned to one and only one node in a directed acyclic graph.

Lemma 2. In a single source DAG, there is a one-to-one correspondence between the nodecodes of a node v and the paths from the source node to node v.

Following is a restatement of Remark 2 for pedigree graphs.

Lemma 3. Let Progenitors(v) denote the set of nodes which are the progenitors of a node v, and Nodecodes(v) denote the set of nodecodes of v in a pedigree graph. Then, there is a one-to-one correspondence between the nodecodes n in Nodecodes(v) of v and the set of all paths from u in Progenitors(v) to v.

This implies that the number of nodecodes of a node v is the same as the number of paths from the virtual source node to v, which is ≥ |Progenitors(v)|. Actually, if there is no inbreeding (crossover of paths) then |Nodecodes(v)|=|Progenitors(v)|.

Representing Paths with Nodecodes: Given nodecodes for a graph, there are different ways to represent paths using nodecodes. Since for any node v, each nodecode of v specifies a distinct path from a vertex u in progenitors(v) to the node v itself, to

IEEE COMPUTER SOCIETY

represent a path, a straightforward way is through a pair of begin, end nodecodes. Due to space limitations, the proofs for the following lemmas stating properties of paths represented by node codes [8] are omitted.

Lemma 4. Let p be a path, and u=start(p) and v=end(p) denote start and end vertices of p. Then p can be identified by a pair of nodecodes $<n_u, n_v>$, where n_u is in Nodecodes(u) and n_v is in Nodecodes(v).

Lemma 5. Let p be a path and start(p) has m nodecodes, then for each nodecode in end(p) there are a total of m distinct pairs of begin, end nodecodes each representing p. Each of these m pairs corresponds to a distinct path from a progenitor to start(p).

Lemma 6. Let p be a path of length k. Then, there exists a string $s=a_1*a_2*\ldots a_k*$ (where a_i, $0 \leq i \leq k$, is an integer, and * is a delimiter) such that for every begin, end nodecode pair <b, e> representing p, e = b+s always holds, where + is the string concatenation.

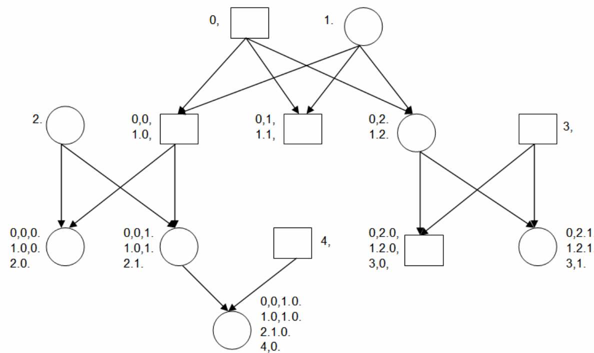

Figure 3: Pedigree labeling on a sample document

Based on this observation, we can use another way to identify a path:

Corollary 1. Let $s=a_1*a_2*\ldots a_k*$ be the path suffix of path p, $1 \leq k$, and b be a nodecode of start(p), then b and s determines p.

Thus, in order to find descendants of a given node v it is sufficient to pick any nodecode from Nodescodes(v). However, finding ancestors require considering all the nodecodes of a vertex since an arbitrary nodecode of a node is not sufficient to find its ancestors.

Let $n_{v=} a_1*a_2*\ldots a_{n-1}*a_m*$ be a nodecode of a node v, where $2 \leq m$. Then "$a_1*a_2*\ldots a_{m-1}$." and "$a_1*a_2*\ldots a_{m-1}$," are in Nodecodes(mother(v)) and Nodecodes(father(v)) respectively. Similarly, "$a_1*a_2*\ldots a_{m-1}*a_m*a_{m+1}*$" is in Nodecodes(children(v)) for each nodecode n_v of v, where a_{m+1} is an integer, and "*" is "." for daughters and "," for sons.

Given Nodecodes(v) of a node v, nodecodes of all descendants, all ancestors, siblings, and other relatives of v can be determined similarly.

IndividualID	PedigreeID	NodeCode
1	10	1,0.
1	10	2.0.
5224	489	7.0;1.
609	8425	5,0.0,0.

Figure 4: Sample NodeCodes relational table fragment

Query Evaluation with Nodecodes: Queries on pedigree structures are converted into SQL statements that utilize nodecodes for efficient evaluation of structural relationships. The SQL statement generated at this step is evaluated over the relational table representation of the pedigree structure to get the final result of the query (see Figure 4). Using nodecodes, retrieving the parents, children, siblings, ancestors or descendants of an individual is reduced to simple string comparison operations on the nodecode labels. Let nc_1 be a nodecode in Nodecodes(n). Given the definition and properties of nodecodes, node *c* is a child of node *n* if and only if *Nodecodes(c)* includes nc_1+s where *s* is a string that consists of two characters (one delimiter character, and one character that denotes the sibling order of *c*). Note that one encoded character is enough to encode the number of biological children a person can have in all but extreme cases. The following query,

Q_A:
```
SELECT n2.IndividualID
FROM NodeCodes n1, NodeCodes n2
WHERE n1.IndividualID=k
AND n1.PedigreeID=n2.PedigreeID
AND n1.nodecode+'_ _' LIKE n2.NodeCode
```

is the SQL representation of this query and returns the children of the individual with IndividualID *k*. Note that in SQL LIKE expressions, '_' is any one character and '%' is zero or more characters of any kind.

The query for retrieving the parents *P* of a given node *n* directly follows this definition, with a modification only in the last line of Q_A. Let nc_2 be a nodecode in Nodecodes(p). Similar to the condition above, *p* is a parent of *n* if and only if *Nodecodes(n)* includes nc_2+s where, again, *s* is a string that consists of two characters. The last line becomes:

```
AND n1.nodecode LIKE n2.nodecode+'_ _'
```

Similarly, node *d* is a descendant of node *n* if and only if *Nodecodes(d)* includes nc_3+s where *s* is a

IEEE COMPUTER SOCIETY

nonempty string and nc3 is a nodecode in Nodecodes(n). The SQL representation of this query is very similar to Q_A, with a modification in the last line:

```
AND n1.nodecode+'_%' LIKE n2.nodecode
```

To retrieve ancestors *A* of a node *n*, we reverse this:

```
AND n1.nodecode LIKE n2.NodeCode+'_%'
```

Since the nodecode definition encodes gender information in the delimiters, including gender in these queries is trivial. For example, the condition in the last line of Q_A needs to be modified as follows to find all male descendants of the specified individual.

```
AND n1.nodecode+'%,' LIKE n2.NodeCode
```

Retrieving the siblings *B* of a node *n* involves the use of some built-in string functions. A node *b* is a sibling of node *n* if and only if *b* has at least one nodecode that differs from a nodecode of *n* by only the last two characters. Excluding self from the results is a crucial condition in finding siblings. The generated SQL statement for retrieving siblings would have the following part[1] instead of the last line of Q_A:

```
AND n2.nodecode LIKE SUBSTRING(n1.nodecode, 0, LEN(n1.nodecode)-2)+'_ _'
AND n2.nodecode <> n1.nodecode
```

More complicated relationships can be converted to SQL using these basic steps, or SQL queries similar to the ones above can be generated for the commonly used set of complex structural relationships such as uncle/aunt or cousin relationships. A more complicated SQL query example that uses the above definitions is the SQL query for Q_4 in *figure 2*:

```
SELECT DISTINCT(i1.IndividualID)
FROM FamilyMembers i1
WHERE i1.Proband = True
AND 2 <= (
        SELECT COUNT(DISTINCT n2.IndividualID)
        FROM NodeCodes n1, NodeCodes n2
        WHERE
        n1.IndividualID=i1.IndividualID
        AND n1.PedigreeID=i1.PedigreeID
        AND n2.PedigreeID=i1.PedigreeID
        AND n2.nodecode LIKE n1.nodecode + '_%'
        AND 3 <= (
        SELECT SUM(p1. PolypCount)
        FROM Polyps p1, FamilyMembers i2
        WHERE p1.IndividualID=i2.IndividualID
        AND n2.individualID=i2.IndividualID
        AND p1. PolypsExamDate < i2. DOB + 35
        )        )
```

[1] Note that this is simplified for demonstration. If the individual is a progenitor, then they will have no parent and this case must be handled as well.

Q1. ID45/Ancestor
Q2. ID75/(FirstDegreeRelative | SecondDegreeRelative) [/PatientData/Cancer [Year(@DateDiagnosed) > Year(..@DOB)+50]]
Q3. Individual [COUNT (/Father/ (Descendant|Self) [@HasCancer]) >=3 AND COUNT (/Mother/ (Descendant | Self) [@HasCancer])=0]
Q4. Proband [COUNT (/Descendant [SUM (/PatientData / Polyps [(Year(@ExamDate) > Year (..@DOB)+35)] @PolypNumber)]) >= 2]

Figure 5: PQL representations of the first four queries in Figure 1

7. Experimental Results

In this section we show the effectiveness of our framework for query evaluation by comparing the use of SQL and nodecodes with a common method used in existing systems [18,17,1]. We examine the performance of real-life queries using data from the Cleveland Clinic's Familial Polyposis Registry [5].

Experimental data: The Cleveland Clinic's (CCF) Familial Polyposis Registry [5] is the largest inherited colorectal cancer registry in the United State and the second largest in the world. At present, the Polyposis Registry database contains pedigrees of 750 families and 11,350 patient histories recorded in the past twenty-five years at CCF. It captures complex pedigree and clinical data such as demographic characteristics, pedigree relations, the distribution of polyps, cancer sites, surgical procedures, and medical treatments.

We performed our experiments on the available good pedigrees in this dataset, which consisting of 655 pedigrees containing 8381 individuals. These pedigrees had a maximum size of 118 individuals with an average size of 12 individuals. In total, 3862 individuals were progenitors, with at most 67 progenitors per pedigree and 5 on average. The maximum number of generations in a pedigree was 8 and the maximum number of children for any one individual was 15.

Experimental setup: We tested the effectiveness of our method using C# 2005 and SQLServer 2000. We implemented the nodecode labeling algorithm and used strings to store nodecodes with the sibling numbers

0-7695-2590-3/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

encoded in a base-64 representation. For typical human pedigrees, this means that all steps could be represented in two characters (one for encoded sibling number, one for delimitor), with the exception of possibly progenitors (only three of which required three character labels in our data). We implemented Q1, Q2, Q3, and Q4 by transforming the PQL into both SQL utilizing nodecodes as well as the näive method of iterative querying in C# using SQL to find parents/children by simple SELECT statements using the MotherID/FatherID stored for each individual. For the descendant axis step, it finds all children of the starting point then finds their children, making another simple SQL query for each individual visited under all descendants are explored. Results from axis steps are stored in C# lists and combined with other steps in C# code to implement more complex queries.

For Q2, which consisted of the union of seven steps with a definite length (i.e. parent, child, sibling, grandparent, grandchild, aunt/uncle, and niece/nephew, but no ancestor/descendant), could also be implemented in plain SQL using joins on MotherID/FatherID and UNION between the different axes. For this query, we evaluated it with nodecodes in a two step process: 1) retrieve the individual and their nodecodes and 2) from the nodecodes generate a SQL expression that evaluates all relevant steps. For example, if the query is a progenitor (which have a single label containing their progenitor number within the pedigree instead of their sibling order), then clearly the only steps that can possibly apply are child and grandchild (as their parent is not known). For ancestor-related queries, all labels are examined and relevant steps are added as conditions to the SQL query, but only one label needs to be used for descendant-related steps.

For Q1 and Q2, which take a single individual as input, we generated 10 random individuals from the largest pedigree, repeated the query 5 times each for each individual and then averaged the results over all individuals. For Q3 and Q4, which operate on all pedigrees, we ran each one 10 times and took the average. The database cache was cleared before each query was started. In addition to timing the queries, we were able to obtain the number of read operations and the bytes read from SQLServer. The experiments were run on a 2Ghz Pentium 4 machine with 1GB ram running Windows XP.

Results: The performance results can be seen in Figure 6. For Q1, the randomly chosen individuals had a low average number of ancestors, and even in this case where the iterative implementation required few queries (i.e. best case for iterative), we can see that nodecodes performed equally well. For Q2, we can see that all the additional joins and reads slows down the SQL only implementation (4 times slower than iterative), but using nodecodes reduced the number of reads and bytes accessed for a 30% improvement (1.4 times faster) over the iterative method. However, it is not until the more complex Q3 and Q4 where we can see the full potential of nodecodes. Even though more bytes are read by nodecodes in Q3, there are less read operations and only a single SQL query is required (versus 33562 queries for iterative), resulting in nodecodes being 1.8 times faster. Finally, for Q4, which contains a SUM condition instead of an attribute check as in Q3 to identify qualifying individuals, we can see the nodecodes/SQL evaluation performs 8.4 times faster than the iterative method.

Query - Method	# Rows	Time (ms)	# Reads	Bytes Read	# SQL Queries
Q1 - Iter (avg)	0.3	34	4	33587	2.3
Q1 - Sql/NC (avg)	0.3	34	5	36864	2.0
Q2 - Iter (avg)	1.5	119	12	101581	25.0
Q2 - Sql only (avg)	1.5	486	110	2587853	1.0
Q2 - Sql/NC (avg)	1.5	84	10	82739	3.0
Q3 - Iter	50	20656	320	2809856	33562
Q3 - Sql/NC	50	11473	255	4112384	1
Q4 - Iter	4	14002	354	3121152	2065
Q4 - Sql/NC	4	1663	321	4358144	1

Figure 6: Performance results

We also need to examine the storage costs associated with this performance improvement. For our dataset of 8381 individuals, the labeling program generated 15548 labels in total, for a very reasonable average of only 1.86 labels per individual. The maximum number of labels per individual was 14. The database tables used in our experiments are shown with their sizes in Figure 7. From this you can see that the storage requirements for using nodecodes on pedigrees are quite reasonable given the performance improvements for complex queries.

Name	Rows	Data	Index
FamilyMembers	8381	5776 KB	1664 KB
NodeCodes	15548	656 KB	576 KB
Cancer	565	176 KB	24 KB
Polyps	3188	1376 KB	24 KB

Figure 7: Database table sizes

8. Conclusion

We have proposed a framework that will allow queries on pedigree data to be expressed effectively and provided a way to evaluate these queries

IEEE COMPUTER SOCIETY

efficiently. For this, we introduced PQL, an XPath style query language for pedigree structures, which can express complex pedigree queries on both the pedigree structure and the associated data intuitively. We provided a comparison of pedigree and XML in terms of structure and querying methods, and defined a directed acyclic graph model of pedigree data that includes gender as well. In order to evaluate PQL queries on this model, we described a labeling scheme for the pedigree structure and an efficient query evaluation methodology using these labels (nodecodes). We experimentally demonstrated that query evaluation using this scheme is much more efficient than the previously used methods, especially for complex queries involving ancestor or descendant retrieval.

Our future work includes developing a full-scale system for storing, querying and visualizing pedigree data, which would also allow the visual construction of PQL queries. We are also working on improving the efficiency of nodecodes further by using an encoding scheme that represents these labels in a more compact way.

References

[1] R. Agarwala, L.G. Biesecker, K.A. Hopkins, C.A. Francomano, and A.A. Schäffer. Software for Constructing and Verifying Pedigrees Within Large Genealogies and an Application to the Old Order Amish of Lancaster County. *Genome Research*, 98(8):211-221, 1998.

[2] S.F. Akgul, B. Elliott, and Z.M. Ozsoyoglu. Insert-friendly XML Labeling System for Efficient Query Processing. *CWRU Technical Report*, 2005.

[3] T. Bozkaya, N. Balkir, T. Lee. Efficient Evaluation of Path Algebra Expressions. *CWRU Technical Report*, 1997.

[4] J.M. Church. A scoring system for the strength of a family history of colorectal cancer. *Dis Colon Rectum*, 48(5):889-896, 2005.

[5] http://colorectal.ccf.org

[6] Cyrillic Software, http://www.cyrillicsoftware.com/

[7] S. M. Dintelman, and A.T. Maness. An implementation of a query language supporting path expressions. In *Proceedings of the 1982 ACM SIGMOD international Conference on Management of Data*, 87-93, June 2002.

[8] B. Elliott, S.F. Akgul, Z.M. Ozsoyoglu, E. Manilich. Modeling and Querying Pedigree Data. *CWRU Technical Report*, 2006.

[9] J.R.Garbe, and Y. Da. Pedigraph 2.0, a software tool for the graphing and analysis of large complex pedigrees. *Abstract book, ADSA-ASAS-PSA Joint Annual Meeting*, 04:25-29, 2004. http://animalgene.umn.edu/pedigraph.html

[10] H.T.Lynch, A.D.L. Chapelle. Hereditary Colorectal Cancer. *N Engl J Med*, 348: 919-932, 2003.

[11] Pedigree and Population Resource: Utah Population Database. http://www.hci.utah.edu/groups/ppr/

[12] Progeny Software, http://www.progeny2000.com/

[13] S. Abiteboul, P. Buneman, D. Suciu. Data on the Web: From Relations to Semistructured Data and XML. Morgan Kaufmann, 1999.

[14] L. Sheng, Z.M. Ozsoyoglu, and G. Ozsoyoglu. A Graph Query Language and Its Query Processing. *Proceedings of 1999 ICDE Conference*, 1999.

[15] T. Kameda. On the vector representation of the reachability in planar directed graphs. *Information Processing Letters*, 3(3), January 1975.

[16] Technical report/formal language specification website.

[17] R.V. Berloo, and R.C.B. Hutten. Peditree: Pedigree Database Analysis and Visualization for Breeding and Science. *Journal of Heredity* 96 (2005): 465-468.

[18] E.A. Wernert, and J. Lakshmipathy. PViN: a scalable and flexible system for visualizing pedigree databases. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, 05:115-122, 2005.

[19] A. Yount. The Use of Relational Database Commands in Retrieval of Pedigree Information. *Journal of Medical Systems*, 87(11):Nos. 2/3, 1987.

[20] XPath Language specification http://www.w3.org/TR/xpath

[21] Glosary of Genetic Terms, National Human Genome Research Institute http://www.genome.gov/glossary.cfm?key=pedigree

0-7695-2590-3/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY