A Graphical Tool for *Ad Hoc* Query Generation*

Kilian Stoffel‡, John D. Davis†‡, Gerald Rottman†‡, Joel Saltz†‡, James Dick, William Merz‡, and Robert Miller‡

†Computer Science Department
University of Maryland
College Park, MD 20742

‡Johns Hopkins Hospital
Department of Pathology
Baltimore, MD 20885

Medical data are characterized by complex taxonomies and evolving terminology. Questions that clinicians, medical administrators, and researchers may wish to answer using medical databases are not easily formulated as SQL queries. In this paper we describe a graphical tool that facilitates formulation of ad hoc questions as SQL queries. This tool manages multiple attribute hierarchies and creates SQL query strings by navigating through the hierarchies. This interactive tool has been optimized using indexing to improve the overall speed of the query building and the data retrieval process. Indexed queries performed 5 to 100 times faster than query strings. However, query string generation time depends on the size of the taxonomies describing the hierarchies, while the index generation time depends on the size of the data warehouse.

INTRODUCTION

Database support of medical practice, hospital administration, and clinical research is complicated by partially standardized and evolving terminology and by complex medical taxonomies. Relational databases require the consistent use of a set of attributes describing the domain of interest. Typical medical databases have attributes with enormous name space, i.e., range of possible values. This makes it difficult to formulate many queries. In addition, attributes may have semantic dependencies. The meaning of one attribute value depends on the value of a different attribute.

To overcome this sort of difficulty, we are exploring the use of semantic knowledge, stored in the form of ontologies.[1] These ontologies are hierarchies of attributes that support efficient querying and indexing of large databases. An illustration may be given from the domain of clinical microbiology.

*This research was supported by the National Science Foundation under the Grants #ASC 9318183 and #ACI-9619020 (UC Subcontract #10152408), and DARPA under Grants #N66001-97-C-8534 and #DABT 63-94-C-0049 (Caltech Subcontract #9503).

Microbiological identification is a process of iterative refinement. There is no single battery of tests that can be applied to all specimens to obtain a precise identification. This process of identification is carried to varying degrees of refinement depending on the body source of the specimen, the current state of microbiological knowledge, and clinical considerations. Reported microbial identities run the gamut from broad, descriptive groupings to family and species designations. Gram stain, metabolic, and morphologic characteristics may also appear in the microbiology report.

For example, the organism *Neisseria gonorrhoeae* may be identified in the laboratory as an oxidase-positive, gram-negative diplococcus growing on Thayer-Martin medium or, directly, by gene probe analysis. Depending on techniques utilized and progress in identification, the same organism may be reported variously as "*Neisseria gonorrhoeae*", "gram negative diplococcus", "*Neisseria species*", etc. Thus, the traditional taxonomy structure does not support this iterative identification process because there may exist multiple paths to a single organism identification. Rather, the data form a more complex structure in which attribute values may have multiple practical definitions.

We demonstrate in the domain of clinical microbiology how efficient indexing, complex data access, and high-level querying can be supported for users untrained in the details of the underlying database forms. Our approach is general and can be applied to all types of database attributes with complex hierarchies. We present an algorithm used to formulate complex queries and an indexing scheme that improved the performance of query generation and execution in an interactive system.

Related Work

Over the last few years, much research has focused on optimization of aggregate queries.[2,3] This development was largely driven by applications for online analytic processing (OLAP) and decision support systems (DSS).[4] Data cubes are a key technology that emerged from this research.[5,6] Materialized views are another optimization approach related to our work.[7,8]

We found that even a very large number of preprocessed views is not sufficient to support the queries encountered in a medical context. The high degree of variation in the types of queries, as well as the combinatorial complexity of the hierarchical attributes, make the use of materialized views impractical.

This paper proposes a scheme in which metadata describing an attribute hierarchy is used, in effect, to support the dynamic creation of new derived attributes. The ability to support metadata-based hierarchies is important in our applications because large hierarchies are not well supported by previously proposed methods.[9] We describe a tool which allows on-the-fly creation of queries for multiple attributes, as well as an indexing mechanism that is quite different from those proposed for standard data cubes.[10,11]

APPLICATION OVERVIEW

We are analyzing information in a medical data warehouse that we are maintaining at Johns Hopkins Hospital. The warehouse includes in-patient and out-patient data from the clinical laboratories. In addition, we have access to a hospital-pharmacy database and a hospital-administration database containing demographic, treatment, and billing information. We use Microsoft SQL Server 6.5, running on an NT 4.0 platform. Laboratory data is downloaded in HL7 format nightly from an MSM laboratory information system.

Motivating Example

One study using this tool evaluated two treatment regiments for immunosuppressed cancer patients. This study compared the use of different prophylactic antibiotics to prevent infection in these patients. The first task was to formulate a query defining the patient population. This query would identify all oncology patients with cultures growing gram-negative, aerobic rods. Using this patient population, the next set of queries would provide all the patients resistant to the two treatment regiments.

A query such as "Find all patients with cultures growing gram negative rods" seeks not only patients for whom *Organism* = "gram-negative rod" but also patients where *Organism* = "subcategory of gram-negative rod". Our intended meaning is the set of all microbial identifications (nodes) that can be reached by following category-subcategory relations starting from gram negative rods. Our patient population query requires a disjunctive expression as in "Select all patients where *Organism* = gram-negative rod OR *Organism* = Pseudomonas aeruginosa OR ..." Moreover, for most queries, aggregate functions (especially the count function) are applied to these select statements (e.g., "How many patients were infected with gram negative bacteria?").

Queries of medical databases entail two fundamental problems. First, it is very time consuming to generate the large disjunctive expressions, and second, questions change regularly; it is not possible to predefine all queries. Therefore, we created a tool that allows us to manage multiple, large hierarchical attribute spaces to generate queries.

Data Structures

In general, the list of possible microbial identifications can be ordered, i.e., "xxx" is a subcategory of "yyy", which is a subcategory of "zzz". Note, that the attribute classification is not a traditional taxonomy, but rather what is termed a directed acyclic graph (DAG).[12] In the example given above, “xxx” could also be a subcategory of “www” and “www” is not a subcategory of “yyy” or “zzz”.

A DAG representing the ordered relationships between attributes must be constructed in order to generate queries or indices for hierarchical attributes. We have used an object-oriented approach to construct these hierarchies, which simplifies the manipulation of attributes when constructing complex queries.

The DAG is comprised of four types of nodes, root nodes, intermediate nodes, alias nodes, and duplicate-name nodes. Root nodes are the starting points of the DAG and the starting points of the scan operations. Usually, these nodes are present only to provide a more user-friendly structure to the DAG and have no corresponding values in the database. Intermediate nodes are guaranteed to have a predecessor. In most cases, these nodes have associated values in the database.

Alias nodes were introduced to make management of the name space simpler. Nomenclature in microbiology is dynamic. Alias nodes provide a means to link old terminology with new terminology. Furthermore, aliases can also be used when a single value in the database is recorded under multiple attribute names. This provides an embedded thesaurus for the taxonomy. Duplicate-name nodes are distinct nodes in the DAG that share the same name. The DAG, in general, is built under the unique name assumption, which means that every node has a unique name. However, the entire name space is not unique and it is necessary to provide a mechanism to differentiate nodes with the same name. This can be accomplished by specifying the direct predecessors of the nodes in question.

The data structure has the following declaration:

```
class Node : public Persistent_Object {
  public:
  String name;              // Node name
   Int node_type;           // Node type: alias, root, etc.
```

```
    Node **successors;      // List of child nodes
    Node **constraints;     // List of constraining nodes
    Void insert_successor(Int ID);   //build DAG
    String generate_query();
    Int *generate_index();}
class Node_with_Value : public Node {
   public:
   String attribute;          // Name of attribute
    String value;             // Value in database
    Int value_type;           // Data type
    Iptr Index_Pointer;}      // Indices of all other like
                              // records in the database
```

The information stored in the Node class defines the hierarchical structure of the domain for an attribute. The Node_with_Value class extends the Node class. This class extension applies to attributes that occur in the database and specifies the format of the node in the database. This extension was made because there are nodes in the hierarchy, which are needed to build a meaningful hierarchy of attributes, but do not have associated values in the database. An example is the family of Enterobacteriaceae. Many isolates are Enterobacteiaceae but none are reported simply as "Enterobacteriaceae".

Operations

The following operations are defined for the Node data structure:

Instance retrieves all the *direct* instances of an attribute from the database.

Successor retrieves the *successor-list*.

Transitive_Successor returns all sub-categories directly or indirectly connected to an attribute by the directed links.

Constraint retrieves the *constraint-list* of an attribute.

Transitive_Instance is a combination of the transitive successor operation and the instance operation. In the first step, the set of all successor nodes is ascertained. In the next step, all instances of these attributes are gathered.

Programming Methods

There are two main methods that were the focus of our performance study. **Generate_query** generates queries from the user interface. **Generate_index** creates an index in order to increase performance.

Generate_query. The Generate_query method uses a simple language that we have previously described[13] to generate SQL queries from high-level expressions formulated by the user. The user selects attributes from a graphical representation of the hierarchy by selecting individual nodes or alternatively, by selecting a node and all its transitive successors (see Figure 1). These operations generate SQL queries typified by (single node selection) SELECT * FROM PatientData WHERE Organism = '*Escherichia species*' or (Transitive node selection) SELECT * FROM PatientData WHERE Organism IN ('*Escherichia species*',' *Escherichia coli*','*Escherichia coli* O157:H7'). Arbitrarily complex queries may be generated by disjunction and conjunction of basic queries such as these.

Several optimizations have been implemented. The most important of these eliminates redundant information. If multiple branches of the DAG are selected in a conjunctive statement, then only the intersection of the branches has to be considered. The number of times a node is selected in a complex query is recorded. In the conjunctive case, if B branches of the DAG are selected, all nodes that were selected B or more times are appended to the query string. In disjunctive queries a node is included in the query string if it has been selected at least once. Aggregate functions have also been implemented in a manner similar to that used in refined Data Cube operations.[9]

Generate_index. Because of the structure of the attribute hierarchies, indexing schemes described in the literature[5,9,11,14] were unacceptable, Thus, we developed a novel indexing method for speeding data retrieval from large databases. We use two data structures to generate an initial index. First we extended the Node_with_Value class, as shown in the Data Structure section, by adding the variable *Index_Pointer*.

Index_Pointer is actually an array of pointers into the database. Each pointer in this array points to a unique occurrence of a specified attribute value in the database. The arrays for each unique attribute are initialized statically to improve the program initialization time. The attribute index arrays are of size (1 + the number of records). The index arrays for each attribute are stored as metadata that list the attribute name, the number of records for that attribute, and the list of indices. This format enables easy updates of the metadata.

We also added to the application a database-index array of size (1 + the total number of records in the database). Entries in this array point to one record in the database and store the number of times a record has been included in various parts of a query. A conjunction-constraint variable is used to track the number of conjunctions and the number of unique constraints. This number is used to decide on the inclusion of a given attribute in a conjunctive query. For inclusion, the value stored in the array must be equal to the number of conjunctions and constraints encountered. For a disjunction, we build the union and then remove all duplicates. The solution is the set of all non-zero indices in the database-index array.

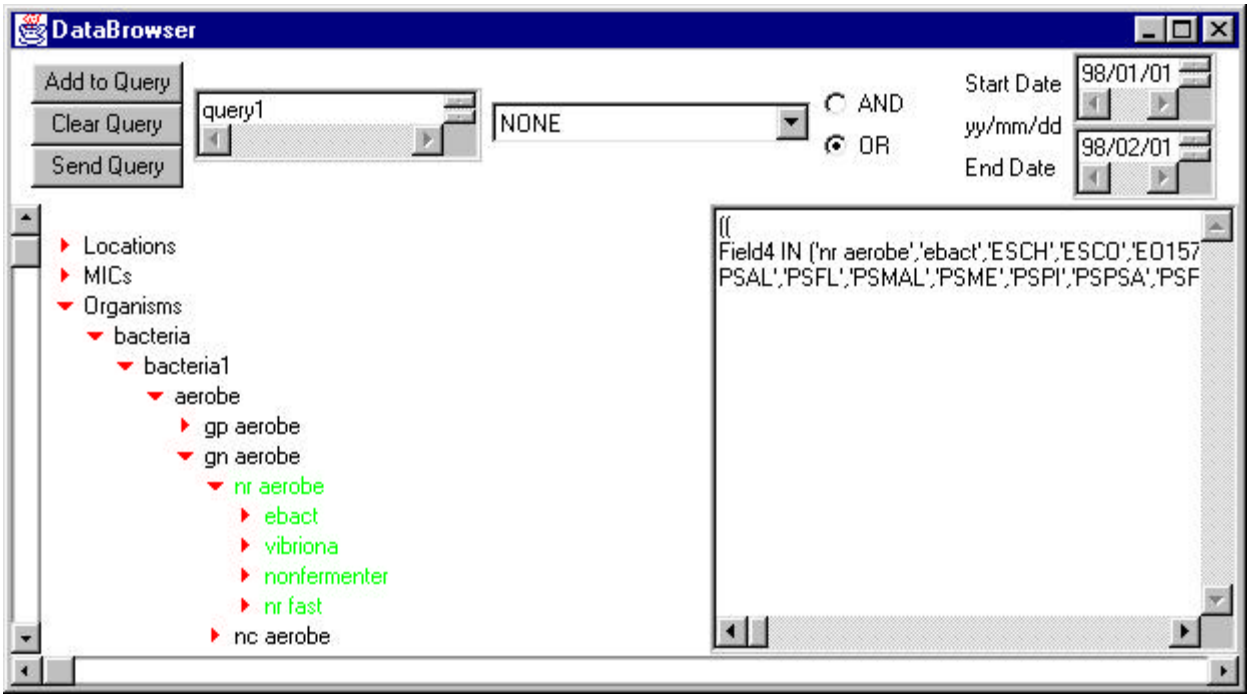

Figure 1. Java implementation of query tool.

Constraints

Queries involving antibiotic susceptibility of microorganisms impose an additional complication on the query formulation process. In the laboratory, antibiotic susceptibility is measured in terms of the minimal concentration of an antibiotic necessary to inhibit bacterial growth (MIC). However, clinicians simply want to know the bottom line. Is the organism susceptible or resistant to the antibiotic in question? MIC values must therefore be translated into terms of susceptibility and resistance by setting threshold values of MIC at which an antibiotic can be considered useful. In practice the procedure is complicated by a need to use different thresholds for different groups of bacteria.

Because MIC thresholds depend on the microorganism in question, antibiotic nodes will be multivalued. In order to restrict the name space of queries produced by these multivalued attributes constraints have to be imposed. We store the MIC values for antibiotics and use constraints to narrow the search to the particular groups of organisms for that particular antibiotic. This enables us to give each attribute multiple dimensions and hide some of the complex query generation details from the user. The constraints can be specified as being conjunctive or disjunctive. The constraints can also be specified for individual attributes or groups of attributes. Each node can have a list of constraints associated with that attribute, which allows for n additional dimensions that could be used to characterize that attribute.

Implementation

Java was selected as the implementation language because of portability and functionality. An object-oriented language was needed to manage and store the metadata used to build the DAGs. We also wanted the flexibility to implement this project both as an application and an applet to be used on the Internet. The Java application interfaces to a database using Microsoft (MS) data access objects (DAO 3.5), giving the application the ability to open databases and manipulate records.

The ontologies are stored in a database in the form of tables, which describe the hierarchy. The Java application reads in this information and produces a preorder DAG data structure for each table. This object contains the name of the database attribute along with multiple modifiers including, the long name, list of children nodes, list of constraining nodes, some status flags, etc. This structure is used to construct the graphical hierarchy displayed using Java's AWT class (Figure 1).

The user is able to formulate queries by clicking on appropriate nodes in the graphical representation of the hierarchy and by entering additional conditions in the dialogue boxes provided. The SQL query generated is displayed in a window for any further editing.

DISCUSSION

The evaluation of this tool was based on ease of use compared to designing the query by hand, completeness of the query, and speed. The user would

have to include over 300 values to design the patient population query discussed in the Motivating Example section. Selecting one value from two attribute hierarchies (Location and Microorganisms) is much easier. There are no organisms or locations in the database that do not appear in the associated hierarchy. This guarantees that the query is complete.

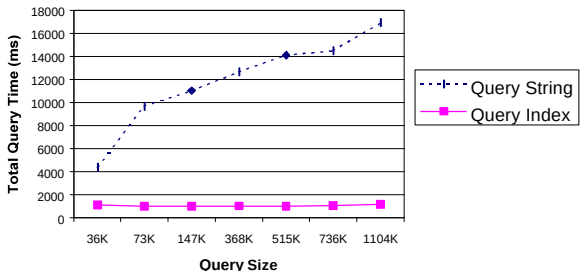

Figure 2. Patient population query performance comparison.

Finally, experiments showed that the indexing scheme offered increasing gains in the retrieval speed as the database size increased as compared with the SQL query strings. As shown in Figure 2, retrieval from a database of 1.1 million records with 35 fields was 100 times faster with indexing than just using SQL strings.

CONCLUSION AND FUTURE WORK

This paper proposes a scheme in which an attribute hierarchy defined by metadata is used to, in effect, support the dynamic creation of new derived attributes. We demonstrated the usefulness of these ideas by implementing a tool that supports the formulation of complex queries that involve attributes with internal structure described by complex hierarchies. In our approach, these hierarchies are maintained as metadata in an object-oriented fashion using a DAG. The information stored in the object hierarchy can then be used to create complex queries. The system produced correct SQL query strings for a variety of medical studies.

This system is currently used by microbiologists and infectious disease specialists at Johns Hopkins Hospital to assess changes in patterns of antibiotic susceptibility and to formulate strategies for empiric antibiotic coverage of neutropenic oncology patients. We are also targeting several other medical application domains whose data elements are characterized by complex taxonomies with dynamically changing terminology and multiple ways of organizing the name space.

References

1. UMLS. "Unified Medical Language System". National Library of Medicine, 1994

2. S. Chaudhri and Kyuseok Shin. Including Group-By in query Optimization. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB*), pp. 354-366, Santiago, Chile, 1994.

3. A. Gupta, V. Harinarayan, and D. Quass. Aggregate-Query Processing in Data Warehousing Environments. In *Proceedings of the 21st Conference on Very Large Databases (VLDB)*, pp. 358-369, 1995.

4. Karl Heiny Hess. Very Large Databases in a Commercial Application Environment. . In *Proceedings of the 22nd Conference on Very Large Databases (VLDB)*, 1996.

5. Jim Gray, Adam Bosworth, Andrew Layman, Hamid Pirahesh: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *Proceedings of the 12th International Conference on Data Engineering (*ICDE), pp. 152-159, 1996.

6. Sameet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, Sunita Sarawagi: On the Computation of Multidimensional Aggregates. In *Proceedings of the 22nd Conference on Very Large Databases (VLDB),*: pp. 506-521,1996.

7. Elena Baralis, Stefano Paraboschi, Ernest Teniente. Materialized Views Selection in a Multidimensional Database. In *Proceedings of the 23rd Conference on Very Large Databases (VLDB)*, pp. 156-165, 1997.

8. Latha S. Colby, Akira Kawaguchi, Daniel F. Liewen, Inderpal Singh Mumick, Kenneth A. Ross. Supporting Multiple View Maintenance Policies, In *Proceedings of SIGMOD '97*, pp. 405-416, 1997.

9. Venky Harinarayan, Anand Rajaraman, Jeffrey D. Ullman, Implementing Data Cubes Efficiently, In proceedings of SIGMOD '96, pp. 205-216, 1996.

10. Nick Roussopoulos, Yannis Kotidis, and Mema Roussopoulos. Cubetree: Organization of and Bulk Incremental Updates on the Data Cube, In *Proceedings of SIGMOD '97*, 1997.

11. T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+ tree: a dynamic index for multi-dimensional objects. In *Proceedings of the 13th Conference on Very Large Databases (VLDB)*, 1987.

12. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, pp. 86-91, 1990

13. Kilian Stoffel, Joel Saltz, Jim Hendler, Jim Dick, William Merz and Robert Miller. Semantic Indexing for Complex Patient Grouping. In *Proceedings of the Annual Conference of the American Medical Informatics Association*, 1997.

14. Timos K. Sellis, Nick Roussopoulos, Christos Faloutsos. Multidimensional Access Methods: Trees Have Grown Everywhere. In *Proceedings of the 23rd Conference on Very Large Databases (VLDB)*, pp. 13-14, 1997.