*A Query Language for Information Graphs**

Sangita C. Betrabet, Edward A. Fox, and QiFan Chen

TR 93-03

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

January 27, 1993

A Query Language for Information Graphs *

Sangita C. Betrabet, Edward A. Fox
Department of Computer Science
Virginia Polytechnic Institute and State University (Virginia Tech)
Blacksburg VA 24061-0106

and

QiFan Chen
HaL Computer Systems, Inc.
8920 Business Park Dr. Suite 300
Austin TX 78759

CR Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical design - *Data models* ; H.2.3 [**Database Management**]: Languages - *Query languages*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval: *Query formulation*; H.3.3 [**Information Storage and Retrieval**]: Systems and Software: *Information Networks*.

General Terms: Languages.

Additional Key Words and Phrases: Object-oriented DBMS, graph theory, recursive queries, transitive closure.

1 Introduction

In this paper we propose a database model and query language for information retrieval systems. The **information graph** model and Graph Object Access Language (**GOAL**) allow integrated handling of data, information and knowledge along with a variety of specialized objects (e.g., for geographic or multimedia information systems). There is flexible support for hyperbases, thesauri, lexicons, and both relational and object-oriented types of DBMS applications.

While developing and experimenting with the COmposite Document Expert/extended/effective Retrieval (CODER) system and extending it to better support semantic network representations and hypermedia applications, we began specification of a reference model and query language [6]. Next, the basic information graph model was described, the Large External object-oriented Network Database (LEND) system was implemented around that model, and the LEND query language was defined [4]. In this paper we give the first published account of our new, powerful model and language (GOAL), illustrate their use, and compare them with related work.

2 Approach

Our goal is to provide a framework for large integrated information systems, that will support objectives of efficiency, usability, and ease of development. We assume that a variety of simple and composite types must be managed, and so allow objects of all sizes, with multiple inheritance.

Among those objects are tuples. Hence we subsume the relational model, and include relevant operators and expressions, as well as support for sets. However, we eschew the philosophy of extending the relational model, and instead of sets and relations use graph theory as the real foundation.

The key concept is to allow links or arcs as first class objects, so that we not only have efficiency gains as in the network database model, but also can reason with or have expressions involving (binary) relationships (as

*This work was funded in part by grants from the National Science Foundation (IRI-8703580) and PRC Inc.

needed for semantic networks and hypertext, respectively). Beyond that we have paths, subgraphs, and a variety of operators to manage them.

In working on GOAL we have also accepted two main challenges. First, we strove to have a language that was not only powerful and expressive but also "intuitive" and usable. Second, we have been developing optimization and implementation methods to go along with the language, so efficiency concerns, especially those related to information retrieval would not be forgotten.

Our handling of these challenges can be clearly seen in the following sections. Of particular importance is that the query language allows us to easily and naturally express linear recursive queries that are difficult or impossible to express using traditional database languages.

In the following section we describe the information graph model and query language. In Section 4 we give examples. Section 5 relates our work to other efforts, and the Appendix gives additional details. Finally, in Section 6 we describe our future plans.

3 Query Language and Database Model

In our model, data is stored in the nodes and arcs of an information graph. GOAL variables have types *graphs*, *paths* or *nodes* and can be assigned a set of graphs, paths or nodes, respectively. A graph is a set of nodes and arcs such that an arc is present in a graph only if the two nodes it connects are present in the graph. A path is a sequence of type $node_1$, arc_1, ..., $node_n$, arc_n, $node_{n+1}$ such that arc_i links $node_i$ and $node_{i+1}$. A path can visit the same node more than once but cannot repeat an arc (i.e., $arc_i \neq arc_j$ for $1 \leq i, j \leq n$). A path of length one is an arc. The first node in a path is called its *source* and the last node is called its *sink*.

Nodes, arcs, paths, and graphs are all objects in this object-oriented database system. An object belongs to a class and has the member variables and functions defined for that class. A class can inherit the member variables and functions of another class.

The value of any object or any of its components is accessed using the dot (.) notation. A component of an object is either a member variable of the object or the value returned by a member function defined (by the database administrator) for that object (or the class it is derived from), and is an object in its own right.

GOAL statements can be classified into three types: *declaration statements*, *assignment statements* and *retrieval statements*. It is possible to declare a variable and assign a value to that variable in one statement. The statements apply *GOAL operators* on sets of graphs, paths and nodes to create new objects of either of these three types. Retrieval statements allow us to create a set of objects and apply a member function to the set, such as *print()*, which displays on the screen the objects in the set.

Query language operators have arguments (beginning with capital letters) that are sets of graphs, paths or nodes. There are two types: *GOAL set operators*, used to create sets of any of the three GOAL types, and *GOAL type operators*, that only create sets of a particular type.

3.1 GOAL Set Operators

In the following list of GOAL set operators, "Set" can be a set of any of the three GOAL types, and a "SetList" is a comma separated list containing one or more sets of the same type.

- **union(Set, SetList):** This operator finds the set union of the sets in the list.
- **intersect(Set, SetList):** This operator finds the set intersection of the sets in the list.
- **diff(Set, Set):** This operator finds the objects that are in the first set but not in the second set.
- **Set.select(Condition):** The *select* operator returns the set of objects satisfying the given condition, which is made up of condition-terms connected by Boolean operators. A condition-term is of type (*attribute-term relational-operator attribute-value*), for relational-operators $==$, $<$, $>$, $<=$, $>=$ or $!=$. An attribute-term is an object or set of objects, while an attribute-value is either an object, a set of objects or a constant enclosed within two quotes ("s).
- **Set.aggregate([GroupList], PartName, AggregateFunction):** Here, GroupList is a comma separated list of objects. PartName is the name of a part that is added to the resulting objects, and AggregateFunction is the name of a function that can be applied to sets of objects present in Set.

The *aggregate* operator first groups the objects in the set into subsets having identical values for the objects in GroupList. It then applies the aggregate function to each subset to create a resulting object of the same type with an additional part called PartName, whose value is determined by the AggregateFunction.

- **ClassName:** It is also possible to create new sets by specifying the name of the class of an object. The class name that is specified can be either a graph, arc or node object and the resulting sets will be of type graphs, paths and nodes, respectively. The result will be all of the objects belonging to that class in the information graph.

- **SetVariableName:** If the name of a variable is specified, the resulting set will have the same set of objects assigned to that variable. Note that a variable must be defined and assigned a value before it can be used.

3.2 GOAL Type Operators

These operators create objects of one type using objects of other types.

3.2.1 The Graph Operators

The following operators create sets of graphs.

- **getGraphs(Paths):** Each graph in the resulting set of graphs created by this operator contains the nodes and arcs present in a path from the given set of paths.

- **getGraphs(Nodes):** Each graph in the set of graphs created by this operator contains only one node from the given set of nodes.

- **gUnion(Graphs, Graphs):** This operator takes one graph at a time from each of the two sets of graphs, and results in a graph containing the union of the sets of nodes and arcs in the two graphs.

- **gIntersect(Graphs, Graphs):** This operator takes one graph at a time from each of the two sets of graphs and results in a graph containing the nodes and arcs that are present in both of the graphs.

- **gDiff(Graphs, Graphs):** This operator finds the graphs containing the difference of the sets of nodes and arcs in a graph from the first set and a graph in the second set.

3.2.2 The Path Operators

The following operators create sets of paths.

- **getPaths(Nodes, PathsNodesList):** Here, PathsNodesList is a comma separated list of alternating path and node sets. This operator creates a set of paths starting from the first set of nodes, connected by paths in the first set of paths to the nodes in the next set of nodes, and so on.

- **pCompose(Paths, PathsList):** This operator creates paths that are composed of the paths in the comma separated list of paths. A path in the first set in the list is joined to a path in the next set if the sink node of the first path is the same as the source node of the following path. Similarly the paths in the second set are joined to the paths in the third set (if present in the list), and so on.

- **allPaths(Nodes, Paths, Nodes):** This operator finds all paths that connect nodes in the first set of nodes to those in the second set with paths from the given set of paths. The resulting set of paths may be of varying lengths.

- **Graphs.getArcs(ArcClassName):** This operator returns a set of paths made up of only one arc, that are of the type *ArcClassName*. The arcs are preceded and succeeded by the nodes that they link.

- **Paths.getArcs(ArcClassName,[DigitsSeq]):** This operator returns the set of paths of length one that are made up of the arcs in the given set of paths of the type ArcClassName, occurring at the designated positions in the digits sequence. The digits sequence is a sequence of digits that can be specified either as $Digit_1$–$Digit_2$ or $Digit_1$, $Digit_2$ or $Digit$.

- **←(Paths) or reverseArrow:** This operator creates a set of paths that run in the opposite direction as the paths in the given set. The number of paths in the result is the same as that in the given set.

3.2.3 The Node Operators

The following operators create sets of nodes.

- **Paths.source():** This operator returns the set of nodes that are the source nodes (first nodes) of the paths in the given set.
- **Paths.sink():** This operator returns the set of nodes that are the sink nodes (last nodes) of the paths in the given set.
- **Graphs.getNodes(NodeClassName):** This operator returns the set of nodes in the given graph that belong to the given class.
- **Paths.getNodes(NodeClassName, [DigitsSeq]):** This operator returns the set of arcs in the given paths that belong to the given class occurring at the designated positions in the digits sequence. The digits sequence is a sequence of digits that can be specified either as *Digit1–Digit2* or *Digit1, Digit2* or *Digit*.

4 Examples

Since our model and language were developed to help with information systems, we give an example of a bibliographic database that illustrates the data modeling and querying possibilities.

We use in our examples an information graph made up of `Document`, `Author` and `Keyword` nodes (see Figure 1). A document node is connected to another document node with an arc of type `Cites` if the first document has cited the second document. A document node is connected to an author node with an arc of type `HasAuthor` if the document has been written by the particular author. Finally, a document node is connected to a keyword node with a `HasKeyword` arc if the document is relevant with respect to the keyword. The `HasKeyword` arc contains the document weight (`DocWt`) of the document with respect to the keyword. The nodes in the graph are also made up of different parts as shown in the figure.

We now give examples of queries on this database, along with the corresponding GOAL query statement(s).

1. Display the document(s) with title XXX.

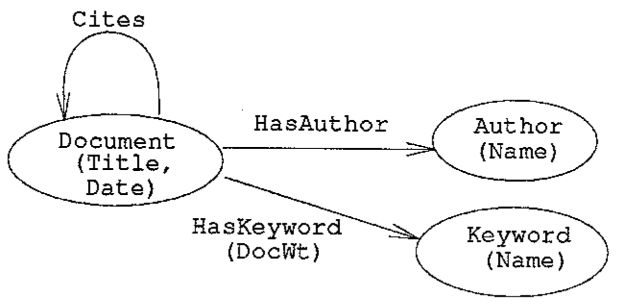

Figure 1: Sample Information Graph

```
Document.select(Title == "XXX").print();
```

This statement selects the document with the given title from the set of document nodes and prints its contents. It is possible to omit the print() function because it is the default function that will be applied on the objects that are retrieved. We will omit this function in the following examples.

2. List all documents that cite the document with title XXX.

```
paths P = getPaths(
   Document.select(Title == "XXX"),
   ← (Cites), Document);
P.sink();
```

The first GOAL statement finds all paths that start from the given document and are connected to another document node with arcs of type `Cites`. We have used the *reverseArrow* operator becauses we want to follow the links in the opposite direction of the `Cites` arcs. The sink nodes of these paths are the required documents.

3. List the documents written by the authors that have cited the document with title XXX.

```
paths P = getPaths(
   Document.select(Title == "XXX"),
   ← (Cites), Document,
   HasAuthor, Author,
   ← (HasAuthor), Document);
P.sink();
```

In this example, we find the paths starting from the given document to the documents that cite it;

following the arcs to the authors that have written those documents we then go another step further to the documents written by these authors.

4. List the title of the documents written in Jan. 1988 and the name of the corresponding author.

```
nodes D =
   Document.select(Date == "Jan.1988");
getPaths(D, HasAuthor, Author).
   print(D.Title, Author.Name);
```

Here, we find the paths linking the required documents to the authors who have written them and then print only the title and the name of the author instead of the whole path object.

5. List all documents that can be reached in at most 3 steps from the document with title XXX, following a path of arcs of type `Cites`.

```
paths P = allPaths(
   Document.select(Title == "XXX"),
   Cites, Document).
   select(Length() <= "3");
P.source();
```

In this example we start from the given document and follow arcs of type `Cites` and find all the documents that can be reached in three or fewer steps. In this example we have assumed that a path object has a member function `Length()` that returns the length of the path.

6. Find the documents that are relevant to the keywords A and B or C. This is an example of a boolean query.

```
nodes DocsA =
   getPaths(Document, HasKeyword,
      Keyword.select(Name == "A").
      source();
nodes DocsB =
   getPaths(Document, HasKeyword,
      Keyword.select(Name == "B").
      source();
nodes DocsC =
   getPaths(Document, HasKeyword,
      Keyword.select(Name == "C").
      source();
union(intersect(DocsA, DocsB), DocsC);
```

The first three statements given above find the documents that are connected to the keywords A, B and C respectively. The last statement then applies the intersect and union set operations to find the required set of documents.

7. List the documents based on their inner product similarity with respect to the keywords A, B and C with weights 0.1, 0.2 and 0.3.

```
paths PathsA =
   getPaths(Document, HasKeyword,
      Keyword.select(Name == "A");
paths PathsB =
   getPaths(Document, HasKeyword,
      Keyword.select(Name == "B");
paths PathsC =
   getPaths(Document, HasKeyword,
      Keyword.select(Name == "C");
union(PathsA, PathsB, PathsC).
   aggregate([Source], Sim, Sum(
   Product(
      pathsA.HasKeyword.DocWt, 0.1),
   Product(
      pathsB.HasKeyword.DocWt, 0.2),
   Product(
      pathsC.HasKeyword.DocWt, 0.3)
   )).print(Document.Title, Sim);
```

The first three statements for this example find the paths that link the documents to the required keywords. The next statement first finds the union of the three sets of paths and then applies the aggregate operator. While aggregating the information, it first partitions the set of paths into subsets containing paths leading to the same document, and then computes the inner product similarity by first finding the product of the document weights on the paths and the given keyword weights, and then adding the results. This similarity is assigned to a new part `Sim` that is added to the path.

The resulting set of paths after applying the aggregate operator will have one path from each document. This path will go to any of the keywords A, B or C depending on the definition of the `Sum()` function. The inner product similarity will be stored in `Sim`.

Finally, the title and the similarity are printed.

In this example, we make use of the fact that the inner product similarity is the aggregation of the document weight information on the paths. Instead of the inner product similarity we can compute any other similarity measure as long as we have defined the functions necessary to compute the similarity.

5 Related Work

The most popular databases and query languages are based on Codd's [5] relational algebra and calculus, with the extensions made by Klug [8] to include the aggregate functions, that make it easier to generate reports and extract information.

However, the relational model is not adequate when dealing with large and complex objects such as multimedia or CAD, CAM and CASE data. Object-oriented databases were introduced as a method of dealing with these complex objects, and are based on the object-oriented paradigm popular in programming language theory. An object-oriented database is made up of objects instead of tuples. Details about the components of the object and the procedures to manipulate them are stored with the object. Object-oriented DBMSs also allow objects to inherit properties of other objects, and assign unique identifiers for the objects, so users are not responsible for assigning primary keys. Our model and language subsume the capabilities of both relational and object-oriented DBMSs.

In addition, as can be seen in the Appendix, GOAL easily handles a special but important class of queries that cannot be supported by a pure relational language.

6 Summary and Future Extensions

This paper has outlined our information graph model and query language. The GOAL interpreter will be completed and demonstrable for *SIGIR '93*, since we are extending the earlier working version developed for LEND. Experimental studies with GOAL will make use of the CACM-3204 test collection, Princeton's WordNet data, and a large collection of library catalog data loaded into LEND for use with the library catalog system being developed at Virginia Tech.

APPENDIX
Linear Recursive Queries

Aho and Ullman [2] described a class of queries called least fixed point queries or general recursive queries that cannot be expressed using relational algebra and calculus. In order to process these queries we need to apply a relational algebra expression f() first on a relation R, and then on the resulting relation obtained, till the resulting relation ceases to change. General recursive queries are very expensive to evaluate because in order to express them we need the expressive power of a programming language.

Bancilhon and Ramakrishnan identified an important subclass of general recursive queries called *linear recursive* queries [3], that can be executed efficiently; the relation R occurs exactly once in the relational algebra expression for these queries. They conjectured that linear recursive queries occur more often and are more useful than other general recursive queries.

Jagadish, Agrawal and Ness [7] showed that every linear recursive query can be expressed as a transitive closure operation possibly preceded and followed by a relational algebra expression.

A transitive closure operation can be defined as follows: Assume R is a relation having two attributes k_1 and k_2 belonging to the same domain K, and a graph G_R corresponding to the relation R has nodes belonging to the domain K, such that an arc $k_i - k_j$ exists in the graph if and only if there is a tuple (k_i, k_j) in the relation R. Then the resulting relation R^* of the transitive closure operation on R will correspond to a graph G_{R^*}, which will have arcs $k_i - k_j$ if and only if it is possible to reach k_j from k_i following arcs in G_R.

The *allPaths(Nodes, Paths, Nodes)* operator in GOAL finds all the paths connecting the nodes in the first set of Nodes to those in the second set of Nodes, considering only links of type Paths.

It is difficult to extend the relational algebra and calculus to find relations that store the path information as we do using the *allPaths(Nodes, Paths)* operator, because the paths can be of variable length and the nodes and arcs in the paths are ordered. The definition of a relation expects the number of columns in the relation to be fixed and unordered.

A number of attempts have been made to extend relational algebra or calculus to include linear recursive queries (see [1]). These attempts either limit the types of queries that can be executed or limit the type of operations that can be performed on the resulting paths obtained. These limitations are imposed primarily to retain the initial definitions of relations and ensure that the result is normalised.

GOAL uses a path to store the result of the transitive closure operation. A path object captures the semantics of the result and hence is a natural representation.

References

[1] Agrawal, R. "ALPHA: An Extension of Relational Algebra to Express a Class of Recursive Queries", *Proceedings of IEEE 3rd International Conference on Data Engineering*, Los Angeles, California, Feb. 1987, 580–590. Also in *IEEE Transactions on Software Engineering*. 14, 7 (July 1988), 879–885.

[2] Aho, A. V. and Ullman, J. D. "Universality of Data Retrieval Languages", *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, San-Antonio, Texas, Jan 1979, 110–120.

[3] Bancilhon, F., and Ramakrishnan, R., "An Amateur's Introduction to Recursive Query Processing Strategies", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington DC, May 1986, 16–52.

[4] Chen, Q. F. "Object-Oriented Database System for Efficient Information Retrieval Applications", Computer Science Dept., Virginia Polytechnic Institute and State University, 1992. Ph.D. Dissertation.

[5] Codd, E. F. "A Relational Model of Data for Large Shared Data Banks", *Communications of the ACM* 13, 6 (June 1970), 377–387.

[6] Fox, E. A., Chen, Q. F. and France, R. K. "A General Reference Model for Hypermedia and Information Retrieval and its Implementation in CODER / LEND". In Emily Berk and Peter Morley, editors, *Hypertext / Hypermedia Handbook* 329–355. McGraw-Hill, Inc., 1991.

[7] Jagadish, H. V., Agrawal R., and Ness, L. "A Study of Transitive Closure As a Recursion Mechanism", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Francisco, 331–344., May, 1987.

[8] Klug, A. "Equivalence of Relational Algebra and Relational Calculus Query Languages having Aggregate Functions," *Journal of ACM* 29, 3 (July 1982), 699–717.