A Query Language for Information Graphs

by

Sangita Betrabet

Thesis submitted to the faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications

©Sangita Betrabet and VPI & SU 1993

APPROVED:

Edward A. Fox

Dr. Edward A. Fox, Chairman

Lenwood S. Heath

Dr. Lenwood Heath

Dr. Clifford Shaffer

December, 1993

Blacksburg, Virginia

c.2

LD
5655
V855
1993
B482
c.2

A Query Language for Information Graphs

by

Sangita Betrabet

Committee Chairman: Dr. Edward A. Fox

Department of Computer Science

(ABSTRACT)

This thesis describes the Graph Object Access Language (**GOAL**) and optimization techniques to execute GOAL statements. GOAL can be used to retrieve information stored in an object-oriented database system modeled as an information graph. It is an extension of the LEND query language developed by Chen et al. The expressive power of GOAL and the effectiveness of the optimization techniques have been demonstrated using the WordNet database. Queries for the database have been identified and formulated using GOAL. The effectiveness of the optimization techniques has been demonstrated for six typical query forms with randomly selected words.

ACKNOWLEDGEMENTS

I would like to thank Prof. Edward A. Fox for his guidance and providing me with financial assistance. I would also like to thank Prof. Lenwood Heath and Prof. Clifford Shaffer for their valuable suggestions.

Thanks also to Robert France for sharing his knowledge on information retrieval systems with me and Dr. Qi Fan Chen for always having promptly answered my questions on LEND. I appreciate the useful comments and suggestions provided by the members of the Envision group.

Finally, I would like to thank my husband Chinmay for providing me with moral support and my parents and my parents-in-law for encouraging me to pursue further studies.

TABLE OF CONTENTS

1 Introduction **1**

1.1 Background 1

1.2 LEND 4

1.3 Research Summary 7

1.4 Organization of Thesis 8

2 Literature Review **9**

2.1 Important Database Models and Query Languages 9

2.1.1 Hierarchical Databases 9

2.1.2 Network Databases 10

2.1.3 Relational Databases 11

2.1.4 Object-Oriented and Extended Relational Databases 13

2.2 Query Languages based on a Graph Model 14

2.2.1 GraphLog Query Model and Language 14

2.2.2 GOOD Query Model and Language 17

2.2.3 LEND Query Model and Language 23

3 GOAL Language Description **26**

3.1 Approach 26

3.2 GOAL Data Model 27

3.3 GOAL Language Description 28

3.3.1 GOAL Set Operators 31

3.3.2 GOAL Path Operators 36

3.3.3 GOAL Node Operators 40

3.3.4 GOAL Graph Operators 43
3.4 Comparison between GOAL and other Query Languages 47

4 GOAL Implementation **52**
4.1 Architecture of the GOAL Query Processor 52
4.2 Processing of the *getPaths*, *pCompose* and the *allPaths* operators 56
4.2.1 Algorithms for processing the *getPaths*, *pCompose* and the *allPaths* operators 57
4.2.2 Query Tree Transformations 59
4.2.3 Statistical Estimation 60
4.2.4 Use of Meta-Information 60
4.2.5 Data Representation 66
4.3 Summary 66
4.4 Discussion 67

5 WordNet Database **71**
5.1 WordNet Database Structure 71
5.2 WordNet Information Graph Structure 72
5.3 WordNet Queries 77
5.3.1 List of Queries 78
5.3.2 GOAL Statements for Queries 78
5.3.3 Discussion 87
5.3.4 WordNet Command Language 89
5.4 Summary 94

6 Conclusion **95**

A Statistics for the WordNet Database **105**

LIST OF FIGURES

1.1 LEND architecture 5

2.1 Sample GraphLog database 16
2.2 A Graphlog query to find descendents of Peter who are not descendents of John (Adapted from [CM90]) 16
2.3 A GraphLog query to find the shortest path from New York to Los Angeles (Adapted from [CM90]) 17
2.4 Sample GOOD database 18
2.5 GOOD query illustrating Node Addition 19
2.6 GOOD query illustrating Edge Addition 20
2.7 GOOD query illustrating Node Deletion 20
2.8 GOOD query illustrating Edge Deletion 21
2.9 GOOD query illustrating Abstraction 22

3.1 Sample information graph 30
3.2 Graph for the common words query 46

4.1 Architecture of the GOAL query processor 53
4.2 Query tree for: `Documents.select( Title == "XXX").Print();` 54
4.3 Trace for processing query: `Documents.select(Title == "XXX").Print();` 56

5.1 WordNet database structure 74
5.2 Class hierarchy of the WordNet node objects 75
5.3 Class hierarchy of the WordNet arc objects 76
5.4 Part of WordNet database 77

LIST OF TABLES

2.1 Partial BNF syntax for the LEND query language Adapted from [Che92] . 24

3.1 Trace for processing the common words query 47

4.1 Estimation heuristics for Set operators 61
4.2 Estimation heuristics for Path operators 62
4.3 Estimation heuristics for Node operators 63
4.4 Estimation heuristics for Graph operators 63

5.1 WordNet relations adapted from [MBF+92] 73
5.2 Number of nodes traversed when processing query 2 90
5.3 Number of nodes traversed when processing query 5 90
5.4 Time required for processing query 4 91
5.5 Time required for processing query 5 91
5.6 Time required for processing query 6 92

A.1 Number of Nodes in the WordNet Database 105
A.2 Number of Relations between **Forms**, **Senses**, **Synsets** and **Frames** nodes in the WordNet Database 105
A.3 Number of Relations between Noun nodes in the WordNet Database 106
A.4 Number of Relations between Adjective nodes in the WordNet Database . . 106
A.5 Number of Relations Between Verb nodes in the WordNet Database 106
A.6 Number of Relations between Adjective and Noun nodes in the WordNet Database 106

Chapter 1

Introduction

1.1 Background

Due to the advances in software and hardware technology, computers are being used in an ever-increasing number of applications. An important software technology that has contributed to the wide use of computers is database systems. Database systems provide users with a uniform method of storing, updating and retrieving data. Uniformity in data management ensures the integrity, security and reusability of data and makes it easier to develop new applications or improve existing applications [Dat87].

The majority of the database systems currently being developed are based on the relational model proposed by E. F. Codd [Cod70, Dat87]. In the relational database model, the data is stored in relations. A relation contains information about a set of entities of the same type and it can be viewed as a table. A row in the table corresponds to an entity, and the columns correspond to the attributes that need to be recorded for the entities.

The relational model is a simple model and is easily understood. Relational database management systems (RDBMSs) are database management systems based on the relational model. Query languages for RDBMSs are based on the relational algebra or calculus. They define operations on relations and result in relations. SQL [Dat87, SQL] is the most widely used query language for RDBMSs. One key feature of query languages for RDBMSs is that they lend themselves to optimization techniques. (See [JK84, Gra93, ME92] for a review of implementation and optimization techniques.)

Though RDBMSs are useful for a wide variety of applications, they have a number of limitations such as:

- Applications that consist of complex objects with complex relationships cannot be modeled easily as a set of relations. For example, CAD (Computer Aided Design) systems are made up of a large number of components which are in turn made up of other components. The relational database model is not suitable for expressing these nested relationships. CAD systems also require that the different CAD designs and versions be represented as views of the database. RDBMSs provide little support for modeling complex objects and relationships, making it difficult to use them for CAD applications [Kim90].

- RDBMSs sometimes do not satisfy the storage and time efficiency requirements of the applications [Lyn87]. This limitation may exist for applications where large amounts of data are processed and those that require fast response times or the execution of expensive operations [Lyn87].

- Another limitation of RDBMSs is that due to the limitations in the relational algebra and calculus, they may fail to provide the functionality required by the application. For example, Aho and Ullman [AU79] describe a class of useful queries involving least fixed points that cannot be expressed using relational algebra or calculus.

Information retrieval (IR) systems are difficult to implement efficiently using an RDBMS unless suitable extensions are made [Lyn87]. IR systems store information regarding documents and are used to retrieve information that would satisfy a user's needs. There are many types of documents, e.g., books, journals and dictionaries. These documents often are complex objects that can in turn be made up of other objects. Thus, books are made up of chapters that are in turn made up of sections and so on. IR systems also employ complex relationships between objects in the database. For example, they may have information stored in dictionaries. The words in a dictionary have different types of relationships such as synonymy, antonymy and morphological variant of.

IR systems typically store large amounts of information and require fast retrieval. The optimization techniques used by standard RDBMSs often do not apply because insufficient

data is captured to allow estimation that would take into account the occurrence distributions commonly found [Lyn87].

Many IR systems also require functionality that is not provided by the RDBMSs. Thus, uncertainty, which is of importance in IR systems, cannot be suitably expressed by RDBMSs [Lyn87]. Furthermore a class of generalized linear recursive queries that is useful for IR applications cannot be expressed using relational algebra or calculus. An example of a generalized linear recursive query is to find all the words that can be reached, starting from a user supplied list, following *synonymy* links stored in an IR system containing information from dictionaries.

Due to these requirements, RDBMSs traditionally have not been used for IR applications. Extensions to RDBMSs to solve one or more of the problems listed above have been proposed. Thus, Fox [Fox81] proposed extensions to the relational database model to allow attributes of relations to be abstract data types and an extended relational algebra to manipulate such complex objects. Lynch [Lyn87] proposed extensions to improve the performance of relational systems and to support IR operations such as ranking and weighted retrieval.

Recently, object-oriented database management systems (OODBMSs) and extended relational database systems have been proposed for applications that cannot effectively use relational database technology. They are able to capture better the semantics of the objects and make use of this information to improve the modeling capabilities, efficiency and functionality provided by the database system. Furthermore, they provide features that make it feasible to develop a whole new range of applications, and are often referred to as next-generation database systems [Cat91].

OODBMSs [Cat91, Kim90, ABD+89] are based on the object-oriented paradigm popular in programming languages [Seb92]. Research in the area of OODBMSs is ongoing. Cattell [Cat93] describes a proposal for a standard object-oriented database query language.

Kim [Kim90] has defined an OODBMS as a database consisting of a set of objects instead of relations. The value of an attribute of an object can be an object in its own right. Objects

are assigned unique IDs by the system. Details about the components of the objects and the procedures (program code) required to manipulate them are stored with the objects. The properties of objects of the same type are abstracted into a class definition and a class can inherit properties of another class. Examples of database systems that satisfy this definition are ORION [K+89], O_2 [D+91], ObjectStore [LLOW91], GEMSTONE [BOS91] and LEND [Che92, FCF91].

Extended relational databases are RDBMSs that have been extended with object-oriented constructs [Cat91]. Examples of such systems are POSTGRES [SK91a] and Starburst [LLPS91].

In the following section we describe the LEND (Large External object-oriented Network Database) system [Che92, FCF91] developed at VPI&SU. LEND is an OODBMS developed primarily to provide database support to manage IR data obtained from documents, thesauri, dictionaries and hypertext systems.

1.2 LEND

LEND satisfies the definition of an OODBMS given in the previous section. LEND consists of three layers as shown in Figure 1.1.

1. The lower-most LEND layer is the *storage layer*. It provides support for storage and retrieval of data. It is composed of hashing and clustering algorithms that are very efficient [FCDH90, FCDH91, FCHD89, FCHD90, FCHD92, Che92]. It also provides some basic object types such as strings, integers, tuples and sets that can be used to build other objects. For example, an object representing a person's name can be derived from a string object.

2. The LEND *graph layer* provides support for uniform representation of the data. At this level, data is modeled as a graph. The nodes of the graph contain information about entities in the database and the arcs of the graph represent relationships between the entities. The nodes and arcs of the graph are each objects belonging to a class

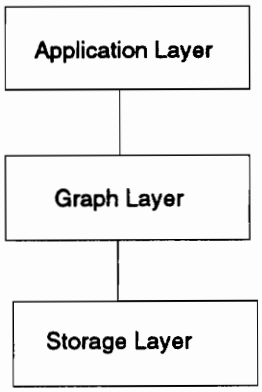

Figure 1.1: LEND architecture

and have the member variables and member functions defined for that class. A class can inherit member variables and functions of another class and an object can be composed of other objects.

3. Finally, the LEND *application layer* is built on top of the LEND graph layer. Applications provide users with different views of the data stored in the database. For example, an IR collection can be modeled as a graph [Che92, FCF91]. Objects in the application and information about their relationships are mapped to node and arc objects in the graph layer. The application layer also provides the functionality required by the application. This functionality is built using a series of graph-level operations.

The key difference between LEND and other OODBMSs is that LEND provides a graph view of the data that distinguishes between node, arc, path and graph objects. We call this database graph an *Information Graph*. An information graph is a natural representation for real-world systems containing complex relationships between objects. In a LEND information graph, the relationships between two objects are represented as arcs of the information graph. Other OODBMSs represent relationships between two objects by nesting the first

object as a component of the second and the second object as a component of the first [Cat91].

In [Che92] it has been shown that an information retrieval system can be easily built using the different components of the graph level. The clustering and hashing techniques discussed there make an efficient implementation possible. Lavinus [Lav92] studied clustering techniques that would reduce the input-output cost of searching an information graph.

LEND has successfully provided database functions required by the MARIAN system which is under development at the VPI&SU Computing Center. The MARIAN system is a state-of-the-art OPAC (On-line Public Access library Catalog) system [FFS+93]. It provides access to the University Library's MARC (MAchine Readable Catalog) bibliographic database.

A query language has been proposed for the LEND graph layer [Che92, FCF91]. Section 2.3.3 contains a detailed description of this language, that can be used to retrieve information from a LEND database. The language can be used in two ways. First, tasks required by applications can be mapped into queries. This approach helps in the development of applications because the LEND query language can be embedded in a high-level language that can access the data. Second, users can directly query an information graph, using ad-hoc questions.

The LEND query language only allows the retrieval of data and cannot be used to modify data. The decision to restrict the language to retrieval of data was made because the first few applications that used the language were static and did not allow for any database modification. A future project could be to extend the language and enable it to modify the databases.

The problem of describing the characteristics of user required information from the database is analogous to the problem of specifying the characteristics of a subgraph of the information graph. Thus, LEND query language operators can be used to specify a graph pattern that describes the characteristics of the subgraph of interest. That subgraph can be a node or a path or a larger portion of the information graph. A path is a sequence of

alternating node and arc objects. New patterns can be formed using patterns previously defined.

[Che92] gives examples of queries for IR systems that cannot be expressed using relational query languages but can be expressed using the LEND query language. For example, for a database containing information about words and the synonym relations between them, the LEND query language can be used to find all the words that can be reached by following synonym links starting from a given word.

1.3 Research Summary

The objectives of this research were to increase the expressive power of the LEND query language and demonstrate optimization techniques for efficient execution of queries for the improved language. The new improved language is called GOAL (Graph Object Access Language). Specific improvements that have been made are:

1. The expressive power has been increased, to better support retrieval of information stored in an IR system.

2. Optimization techniques that make it feasible to use the language for IR systems have been developed.

We have illustrated the expressive power of GOAL and the effectiveness of the optimization techniques by using GOAL to query the WordNet database [MBF+92], which contains information about words and their meanings. Queries for this database have been identified, GOAL query statements for those queries have been formulated and finally the effectiveness of the optimization techniques has been demonstrated by both implementing the optimization techniques and executing the queries. Also, we have shown that the optimization techniques are able to significantly lower the execution costs of frequently used and expensive operations.

We believe these improvements will make it easier to retrieve information stored in a LEND information graph, develop new IR applications and improve existing applications.

We expect that GOAL will be helpful in these ways in connection with Project ENVISION [BCF+93, Fox92, FL93, FHH91, FHN+93] and further work on the MARIAN system [FFS+93].

1.4 Organization of Thesis

Chapter 1 provides an introduction to the topic of database systems and IR systems. Chapter 2 has two sections. Section 2.1 reviews other database models such as the network and hierarchical database models, the relational database model and the object-oriented database model. It also reviews query languages based on these models. Section 2.2 reviews query languages based on graph models, and includes a description of the LEND query language.

Chapter 3 contains a detailed description of our enhanced information graph model and the GOAL query language. It gives the motivation for all GOAL constructs and examples of GOAL queries. Section 3.4 describes how GOAL differs from the approaches described in Chapter 2. Chapter 4 contains a description of the implementation of GOAL and the optimization techniques that have been used.

Chapter 5 describes how GOAL can be used with the WordNet database. In Chapter 6 we conclude with a discussion of possible extensions to GOAL.

Chapter 2

Literature Review

This chapter is divided into two sections. Section 2.1 reviews important database models and their query languages. Section 2.2 describes query languages based on the graph model. It also includes a description of LEND and the LEND query language.

The discussion in this chapter is geared towards describing the features of other database systems in order to serve as a platform for the discussion in Section 3.4, where we explain the differences between other database systems and GOAL.

2.1 Important Database Models and Query Languages

In this section we describe the important features of the hierarchical, network, relational, object-oriented and extended relational database models along with query languages based on these models.

2.1.1 Hierarchical Databases

The Information Management System/Virtual Storage (IMS) system [McG77, Dat87] developed at IBM is a classic example of a hierarchical database management system. An IMS database has an associated schema tree. The nodes of the schema tree (also called records or segments) give types for the objects or entities in the application system. These records are made up of fields, which represent the different attributes of the objects that are recorded by the system. Relationships between the entities are represented as parent-child links between the nodes.

The IMS database consists of an ordered set of trees. These trees have the same node

types and parent-child links as the schema tree. The fields of the nodes in the database trees have values associated with them. Two nodes of the same type having the same parent are called twins. The database trees are ordered. The nodes in each tree are also ordered, with the child nodes and twins following each other in the ordering.

The IMS query language can be used to perform the following operations:

- Select a database tree based on the value of the fields of any of the nodes in the tree.
- Sequentially examine the objects in the database by considering the database trees in sequence and following links between nodes for each tree.
- Add and delete nodes.
- Create views of the database by hiding nodes (and consequently the trees rooted at those nodes) and also hiding fields of nodes that are not required in the view.
- Create logical databases by changing the structure of the tree or combining two database trees into one.

The restructuring mechanisms to create logical databases do not have a good theoretical basis. As a result there are restrictions on the types of restructurings allowed and the implementation is ad-hoc. This makes systems requiring restructuring complex and difficult to maintain.

The hierarchical databases are useful for applications with an inherent hierarchical structure because they provide good performance. But many real-world applications do not fall in this category. Such application when forced into a hierarchical structure lead to non-intuitive and complex systems [Dat87].

2.1.2 Network Databases

A network database [Dat87] is also made up of nodes representing the entities in the application system and the links between the nodes representing their relationships. However, unlike hierarchical systems, a child node can have multiple parents. The nodes of the

system are called records and they are made up of fields. Most network database systems do not allow cycles, or any links between the same type of nodes.

IDMS (Integrated Database Management System) is the most widely used database system based on the network model [Dat87]. It is based on the proposals of the Data Base Task Group (DBTG) of the Programming Language Committee of the "Conference on Data Systems Languages" (CODASYL) ([DBT71, X3H84]). IDMS operators can be used to seek nodes that have specific values for their fields, and then move from one node to the next from this retrieved set or follow the links from a node to another node.

IDMS (and IMS) programs tend to get complicated because the operations supported by the systems operate on records. The operations provided by other database languages for relational or object-oriented databases (including the LEND language and GOAL) are preferred because they operate on sets. Since a set is made up of records, a set-level operation is equivalent to a sequence of record-level operations, and so can be considered as a 'higher-level' operation when compared with a record-level operation [Dat86b].

2.1.3 Relational Databases

Relational databases [Dat87, Cod70], introduced in Chapter 1, consist of relations that can be viewed as tables. The rows of the tables are called tuples and each row contains information about an entity. The columns of the tables are called attributes. They represent the different types of information regarding the entities in the table that should be recorded. A particular tuple stores the different attributes of a particular entity.

A relation has the following properties [Dat87]:

- Each tuple in a relation is distinct from other tuples in the relation; a minimal set of attributes which distinguish tuples from others is called a primary key.
- The tuples are unordered.
- Attributes are unordered.

- All attribute values are atomic, i.e., an attribute cannot be composed of a set of values.

Query languages for relational databases operate on relations and result in relations. They are based on the relational algebra and relational calculus, which are formal languages for relational databases [EN89]. The relational algebra operators are procedural while the relational calculus operators are declarative. The operations can be used to create relations as follows [Dat87]:

- Create a relation containing a subset of the tuples in the given relation (*select*).
- Create a relation containing a subset of the attributes of the given relation (*project*).
- Create a relation containing the cartesian product of the tuples in two relations (*product*).
- Restrict the tuples of the relation formed by the cartesian product to those satisfying given conditions and possibly restrict the attributes in the result (*join*).
- Given two relations R_1 and R_2 create a relation R satisfying the condition that the product of R and R_1 is R_2 (*divide*).
- Create a relation that is the set union (*union*), set intersection (*intersect*), or set difference (*diff*) of the tuples in two relations.

Klug [Klu82] extended relational algebra and calculus to include the *aggregate* operator in order to make it easy to generate reports and extract information from a relational database. The *aggregate* operator operates on a relation and requires two arguments: a list of attributes of the relation and an aggregate function name. The relation generated by this operator is created by first partitioning the tuples in the given relation by the values in the columns or attributes specified as arguments and then applying the aggregate function to each partition. For example, if the query "find the total number of parts available at each location" is used to query a relation with three attributes – the location number, the part

number and the number of parts available at that location – the aggregate function can be used to group the tuples by location number and then add the number of parts available for each part to find the total for each location.

2.1.4 Object-Oriented and Extended Relational Databases

In this section we discuss object-oriented and extended relational databases. Both OODBSs and extended relational database systems can be used to manage complex objects. Complex objects that are made up of other objects can be stored as nested objects. Relationships between objects are also represented by nesting objects of one type within another object. A one-many relation between objects of type A and B is represented by nesting a set, list or array of objects of type B within objects of type A and an object of type A within objects of type B.

The behavior of an object is defined by the methods (program code) defined for the object. These database systems come with an assortment of other capabilities such as maintaining versions, knowledge management, tools to help program development, etc.

The two main differences between query languages for these database systems and those for relational database systems are:

1. The query languages for OODBSs and extended relational database systems allow objects to be retrieved based on the values of their components that are nested at any depth. For example, if an employee object (*Emp*) stores the department (*Dept*) of the employee and a department object stores the floor (*Floor*) in which the department is located, it is possible to find the employees that work on the first floor by specifying a condition such as (*Emp.Dept.Floor* == *'1'*). Of course, the syntax will vary with the query language.

2. The query languages allow objects to be retrieved by specifying conditions on values returned by methods applied to the objects. For example, if an employee object (*Emp*) stores the salary received each month (*Salary[12]*), and has an associated method

YearlySalary() that computes the yearly salary by adding the salary for a twelve month period, it is possible to retrieve the employees whose yearly salary is more than $40,000 by finding the value returned by the method *YearlySalary*().

A method can be considered as a procedure that encapsulates a property of an object. A method is actually program code; as a result query languages for object-oriented databases are linked closely with object-oriented programming languages. In fact, most OODBSs have extended object-oriented programming languages with support for database operations or are based on an object-oriented programming language. Thus, the ObjectStore [LLOW91] system is an extension of the object-oriented language C++ [Cat91]. On the other hand, the query languages for extended relational database systems have been formed by extending query languages for relational databases with object-oriented features; e.g., POSTGRES [SK91a] is an extension of the INGRES query language for relational databases.

2.2 Query Languages based on a Graph Model

This section describes three query languages for database systems based on an underlying graph model. These query languages use a *graph pattern* to describe the characteristics of the required information. The query processor finds matchings, i.e., subgraphs of the database graph that match the graph pattern.

The GraphLog (Section 2.2.1) and GOOD (Section 2.2.2) query languages described below are called graphical query languages because their systems provide the user with a graphical user interface in which the user can cut and paste parts of a graph to describe the graph of interest. The LEND query language described in Section 2.2.3 uses a command language to describe the desired results.

2.2.1 GraphLog Query Model and Language

In the GraphLog model [CM90] data is modeled as a directed graph with the nodes representing the objects and edges representing their relationships. The edges of the graph

have labels associated with them, and both the nodes and the edges can have attributes associated with them. Figure 2.1 shows a GraphLog database graph containing information about cities and the distance between them. The nodes in the figure represent cities and the edges represent the distance between the cities. The names of the cities and the distance between them are stored as attributes of the nodes and edges, respectively. The edges are labeled `Dist` to indicate that they represent the distance between the cities.

A query is expressed visually on the screen as a query graph pattern with a specially marked edge called the *distinguished edge*, that is marked with a bold line. The nodes in the query graph pattern may be labeled with constant values or variables and the edges can be either *solid edges* or *dashed edges*. Optionally, the edges can be crossed and labeled with regular expressions. (See for example Figure 2.2.)

The query processor finds matchings. The labels and the different types of lines (solid, dashed, crossed) for the edges are interpreted as follows by the query processor.

- Constant values for nodes are matched onto nodes with those values.
- Nodes with variables can match any node in the matched graph.
- Solid edges match edges of the database graph.
- Dashed edges match paths in the database graph.
- If a solid line is crossed, the matching process returns nodes that are not connected by edges.
- If a dashed line is crossed, the matching process returns nodes that are not connected by paths.
- The order of the labels of the edges in the matching are consistent with the regular expression(s) in the query graph.

For each matching, the query processor adds the distinguished edge to the database

graph. The matchings and the corresponding distinguished edges are shown to the user by either highlighting them in the database graph or presenting each matching separately.

GraphLog allows labels of the distinguished edges defined in previous queries to be used when defining new queries. But, the label of the distinguished edge can occur only once in a query; hence it is not possible to recursively define the distinguished edges.

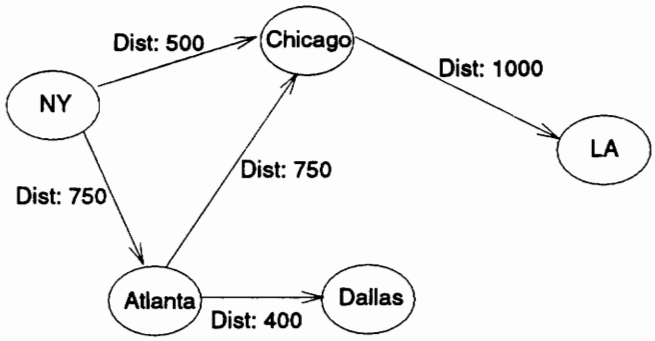

Figure 2.1: Sample GraphLog database

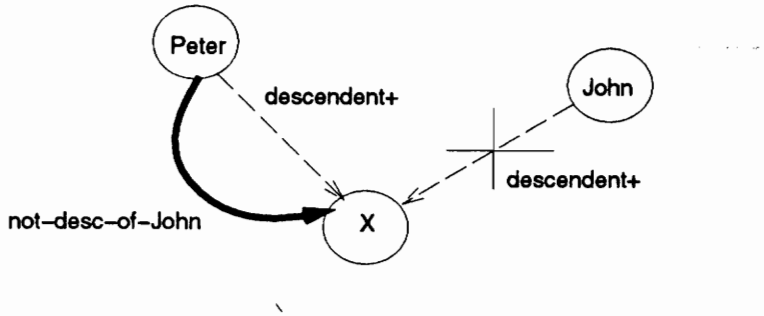

Figure 2.2: A Graphlog query to find descendents of Peter who are not descendents of John (Adapted from [CM90])

GraphLog also allows edges in the query graph to be annotated with aggregate operators (see Figure 2.3). The aggregate operators allow aggregation of information along a path

as well as multiple paths that have the same source and sink nodes. For example, the query shown in Figure 2.3 finds the shortest path between the cities New York and Los Angeles. The `Sum(Dist)` function aggregates information in a path while the `Min()` function aggregates information along multiple paths.

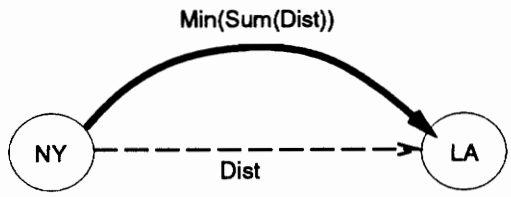

Figure 2.3: A GraphLog query to find the shortest path from New York to Los Angeles (Adapted from [CM90])

To summarize, GraphLog allows users to specify graph patterns of interest and to add (distinguished) edges to the database graph. Consens and Mendelzon [CM90] have shown that the expressive power of GraphLog is equivalent to stratified linear DataLog, first order logic with transitive closure, and non-deterministic logarithmic space (assuming ordering on the domain). The GraphLog system has been implemented using the Smalltalk-80^{TM} and a graph editor called NodeGraph90 [CM90].

2.2.2 GOOD Query Model and Language

Joint work between the University of Indiana and the University of Antwerp, Belgium has focussed on developing a model for a graph based database called GOOD (Graph Object-Oriented Database) [AGP$^+$92, PBA$^+$92, GPBG].

A GOOD database is made up of nodes linked by directed, labeled arcs. (See Figure 2.4.) The nodes can be either of two types: base nodes or abstract nodes. Base nodes have values associated with them and do not have any outgoing edges. In the figure the base nodes have been marked with ovals. Abstract nodes do not have any values associated with

them. In the figure they have been marked with rectangles.

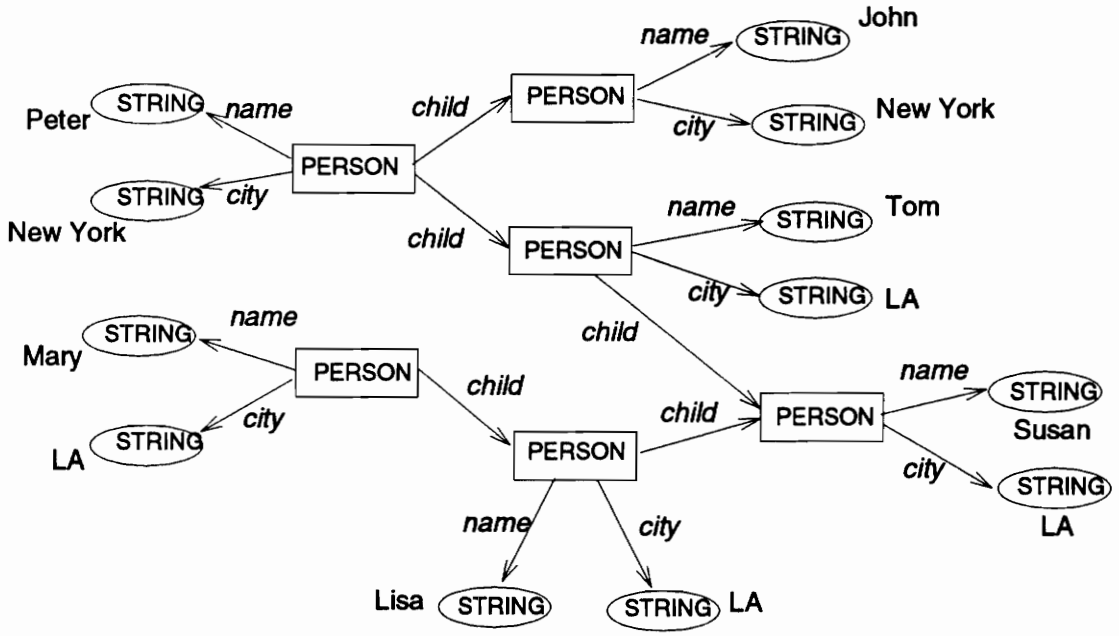

Figure 2.4: Sample GOOD database

The nodes in the database correspond to the different objects in the application system. The abstract nodes correspond to complex objects, while the base nodes correspond to simple objects that are not composed of other objects. The abstract nodes are connected to the base nodes that they are composed of. Thus, city and name edges link a node corresponding to a person with the nodes corresponding to their name and city. Abstract nodes are also connected to other abstract nodes with the links representing various relationships between the objects. For example, a person node can be linked to another person node to represent that the second person is a child of the first person.

The GOOD query is expressed visually on the screen as a graph. The query graph consists of a pattern and an action. The pattern is similar to the GOOD database graph with the exception that it need not have values for the base nodes. The query results in a number of matchings. A matching is a subset of the database graph that has the same

structure as the pattern graph with the labels in the matchings the same as those in the pattern. If a label is not present in the pattern, it can be matched by any base node.

The action consists of a GOOD operation. The action is applied on each matching. The GOOD operations are: *node addition*, *node deletion*, *edge addition*, *edge deletion* and *abstraction*. We describe below each operation in detail.

- **Node Addition:** Node addition is used to add new objects to the database. The node addition operation is frequently used to select nodes of interest by creating a pattern graph containing a node with the characteristics of interest and using the node addition operation to add a new node that is linked to the interesting node from the pattern. Thus, Figure 2.5 shows a GOOD query to find people living in New York for the database graph shown in Figure 2.4. The nodes and links in bold face are the nodes and links that are added as part of the node addition operation.

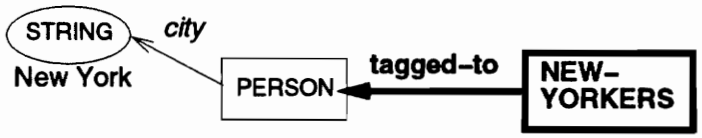

Figure 2.5: GOOD query illustrating Node Addition

- **Edge Addition:** The GOOD edge addition operation is used to define new relationships. Thus, the query shown in Figure 2.6 adds a new relationship *grandchild* (shown in bold face) between people and their grandchildren.

- **Node Deletion:** The node deletion operation, shown in a query using the dashed outline shape, is used to remove objects from the database that are no longer of interest. For example, the query shown in Figure 2.7 is used to remove all people who live in New York from the database.

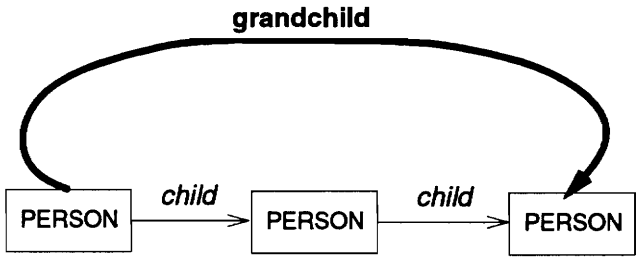

Figure 2.6: GOOD query illustrating Edge Addition

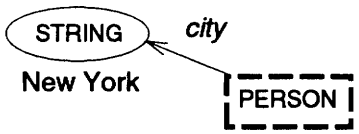

Figure 2.7: GOOD query illustrating Node Deletion

- **Edge Deletion:** The edge deletion operation is used to remove links between nodes. For example, Figure 2.8 describes a GOOD query to change the city corresponding to the person `Susan` from `New York` to `Chicago`. This is accomplished by first deleting the edge linking the person node associated with `Susan` to the city with value `New York` and then adding an edge linking the same person node to the city with value `Chicago`.

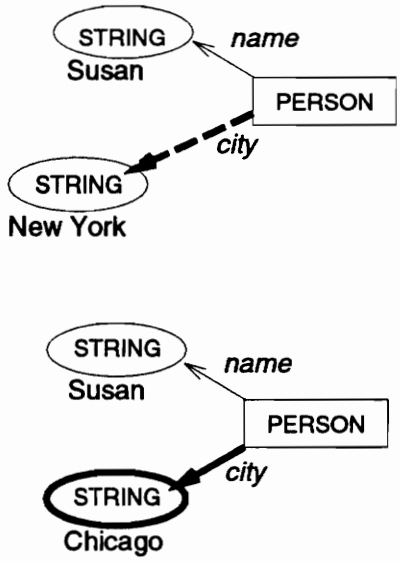

Figure 2.8: GOOD query illustrating Edge Deletion

- **Abstraction:** Abstraction is used to create a unique object that will replace a set of objects that each have the same properties. For example, Figure 2.9 describes a GOOD query to create a node called `Parent` connected to person nodes that have a child node attached. The abstraction operation consists of three parts distinguished in the figure using solid, dashed and bold lines and outlines. The solid outline represents the pattern that should be matched, the dashed line represents the condition for equality, and the bold lines and outlines specify the types of nodes and edges that

should be added as a result of the operation. Double-headed arrows are used in the figure to represent the fact that the source node of the arc can be connected to more than one sink node.

Figure 2.9: GOOD query illustrating Abstraction

GOOD also provides the *method* and *macro* facilities to group a sequence of operations. Users can use the method facility for frequently used operations. Methods can also be used to specify recursive processes. Macros differ from methods because they have special operators that make them easy to use. The macros are predefined.

Gyssens et al. [GPBG] discuss the computational power of the language and show that it can be considered as an object-oriented database language using the definition of [ABD+89]. Further, it has been shown that GOOD is relationally complete and has the computational power to simulate arbitrary Turing Machines [GPBG, BGAG92].

A GOOD query can be viewed as a specification for a graph transformation. Andries et al. [AGP+92] show the use of graph transformations as a mechanism to restructure the database schema, query the data, specify constraints, update the database, update the database schema, view the data in a feasible manner and browse the database.

The GOOD system is being implemented using two approaches. The first approach (used at University of Antwerp) is to build it on top of SQL. GOOD programs are translated into *C++* embedded SQL programs. The second approach is to implement the GOOD system using the binary relational model called the *Tarski Data Model* [SSG91]. An implementation using the *Tarski Data Model* is underway at the University of Indiana.

2.2.3 LEND Query Model and Language

The LEND query model [Che92, FCF91] is based on a graph view of the data, i.e., objects in the database are represented by nodes and the relationships between the objects are represented by directed arcs linking the nodes. The LEND database model satisfies the definition of an object-oriented database given in Chapter 1.

The LEND query language can be used to retrieve sets of nodes or paths from the database. A path is an ordered sequence of alternating nodes and arcs beginning and ending with a node. Table 2.1 gives the Backus Naur Form (BNF) definition of the language.

New sets of paths can be created using four operators: *virtualPaths*, *compose*, *iterate* and *closure*. The *virtualPaths* operator creates a set of paths that link nodes and paths from the given sequence in the given order. The *compose* operator can be used to create a set of paths that are formed by joining a path from the first set of paths to a path from the second set when the sink node of the first path is the same as the source node of the second. The *iterate* operator creates a set of paths by repeatedly joining the paths at least as many times as the third argument and at most as many times as the fourth argument under the condition that the source nodes of the paths belong to the first set of nodes and the sink nodes of the paths belong to the second set of nodes. For example, if X and Y are sets of nodes and P is a set of paths, the statement *P.iterate(X, Y, 1, 3)* finds paths that link nodes from the set X to those in the set Y. The paths are formed by repeatedly joining paths in the set P once, twice or thrice. The *closure* operator repeatedly joins the paths as above but without any restriction on the length of the paths generated.

New sets of nodes can be created using the *sinks*, *select*, *forward* and *backward* operators. The *sinks* operator finds the nodes that are connected to the given set of nodes by arcs having the specified label. The *select* operator finds the nodes that satisfy the given condition. The condition can compare a component of the object to another component of the object or a constant value. The *forward* and *backward* operators find the nodes from the given path that occur at the given position from the beginning or end of the path.

Table 2.1: Partial BNF syntax for the LEND query language Adapted from [Che92]

Paths	::=	Path_ClassName |
		virtualPaths(**Nodes**, **Label**, **Nodes** [**LabelNodeList**]) |
		Paths.compose(**Paths**) |
		Paths.iterate(**Nodes**, **Nodes**, **Num**, **Num**) |
		Paths.closure(**Nodes**, **Nodes**)
Nodes	::=	Nodes_ClassName |
		Nodes.sinks(**Label**) |
		Nodes.select(**Select_Exp**) |
		Paths.forward(**Num**) |
		Paths.backward(**Num**)
Label	::=	Label_Name
LabelNodeList	::=	, **Label**, **Nodes** [**LabelNodeList**]
Select_Exp	::=	**Select_Exp** AND **Select_Exp** |
		Select_Exp OR **Select_Exp** |
		NOT **Select_Exp** |
		(**Select_Exp**) |
		Component **OP** Value_Object
Component	::=	**attribute_name** |
		forward(**Num**) |
		backward(**Num**) |
		[.**Component**]
OP	::=	== | <

The language definition also includes the *intersect*, *union* and *difference* operators that find the set intersection, union and difference for two sets of nodes and paths.

In summary, the LEND query language can be used to specify patterns that contain either a single node or a path. The operators can be used to join existing patterns to form new patterns. However, the LEND language has the following limitations:

1. The LEND language cannot be used to describe a graph pattern containing more than one (unconnected) component. Such graph patterns are useful to answer queries such as "find pairs of people such that the first person earns less money than the second" for an information graph containing information about people and their earnings. Though it is possible to express the query using the LEND query language if the nodes representing the people are connected by arcs representing this relation, it is unreasonable to expect that the database manager has defined all such relations. It would also be infeasible to store relations of all such types.

2. The LEND query language can be used to restrict the types of nodes and arcs that make up the path. Some useful queries require other restrictions on the characteristics of the paths that are not supported by the LEND query language, e.g., to restrict the paths to those that start and end at the same node (i.e., form a cycle.) Other queries require that paths that are 'better' than others be retrieved. These restrictions depend not on the characteristic of a single path, but on the relative characteristics of paths. One such application is to find the shortest path between two given nodes.

In the next chapter we describe the GOAL query language that is an extension of the LEND query language. GOAL has more expressive power than the LEND query language, and allows users to describe graph patterns that do not have the restrictions listed above.

Chapter 3

GOAL Language Description

This chapter is divided into three sections. Section 3.1 describes our design goals and approach. Sections 3.2 and 3.3 describe the GOAL data model and query language, respectively. These two sections motivate the different features of the model and language. These features are explained in the context of the design goals (Section 3.1) and are contrasted with the LEND model and language (Section 2.2.3). In addition, Section 3.3 illustrates the use of all the GOAL operators. (Chapter 4 contains more examples of GOAL queries for the WordNet database.)

Section 3.4 contains a comparison of GOAL with the query models and languages that were discussed in the previous chapter.

3.1 Approach

The objectives of this research are to increase the expressive power of the LEND query language and develop optimization techniques so that the improved language will satisfy the requirements of information retrieval (IR) systems. Other considerations are ease of use and conformance to language design guidelines.

We expect that GOAL will be used to ease the development of applications and to answer ad-hoc queries posed by users. Hence, a typical user of GOAL would be a programmer developing applications with LEND support or a person familiar with the data stored in a LEND database, and using GOAL to query it on a regular basis. Familiarity with graph constructs such as subgraphs, paths, nodes, and arcs is assumed. An attempt has been made to make the operators intuitive and easy to remember. The syntax of the language

is similar to that of the *C++* programming language [Lip92].

Though there are advantages in limiting the language to an essential set of operators (e.g., it is easier to learn, implement and extend), some additional operators were included to make it easier for users to express frequently used queries.

Conformance to language design guidelines was considered during the design. We describe in the following section design decisions to make the language orthogonal, simple and regular [Dat86a, Mac87].

3.2 GOAL Data Model

In the GOAL model, data is stored in the nodes and arcs of an information graph. GOAL variables have type – *graphSet*, *pathSet* or *nodeSet* – and can be assigned a set of graphs, paths or nodes, respectively. A graph $G = (N, A)$ is composed of a set of nodes N and a set of arcs A such that an arc a connecting nodes n_1 and n_2 belongs to the set A only if the two nodes it connects are present in the set N, i.e., n_1, n_2 belong to the set N. A path P is an alternating sequence of nodes and arcs starting and ending with a node, i.e. of form $(n_1, a_1, n_2, \ldots n_k, a_k, n_{k+1})$ such that a_i links n_i and n_{i+1} for $1 \leq i, j \leq k$. The first node in a path is called its *source* and the last node its *sink*. For example, the source of the path P given above will be the node n_1 and the sink n_{k+1}. The length of a path is the number of arcs in the path. For example, the length of the path P given above is k. A path of length one is an arc; i.e., an arc is of the form (n_1, a_1, n_2).

The LEND query language allowed retrieval of sets of paths and nodes but could not retrieve sets of graphs. The graph retrieval feature was added to GOAL because some useful queries for IR systems result in the retrieval of unconnected graphs (see Section 2.2.3).

Both the LEND language and GOAL allow retrieval of path sets. A path has ordered nodes and arcs, and so cannot be replaced by a graph because a graph simply represents a set of nodes and a set of arcs, implying that there is no ordering. A path can represent a list. Paths also are of value since many useful queries require an ordered set to be retrieved.

For example, the query "find the different routes one can take to go from NY to Chicago", for an airline database containing information about cities and flights, requires that a set of lists of cities be retrieved. A path is also a convenient tool to describe some connected graphs. For example, they are valuable to represent hypertext tours.

Sets of nodes can be considered as sets of graphs containing only one node object. The GOAL definition does not include sets of arcs because they can be represented as sets of paths each containing only one arc.

Nodes, paths, arcs and graphs are *objects*. All objects of the same type belong to a *class*. A class of objects can be derived from another class and can inherit properties defined for that class. Objects are *complex* if composed of other objects; otherwise they are *simple objects*. Paths and graphs are always complex objects because they each are made up of node and arc objects. Complex objects can be recursively composed of other complex objects and/or simple objects. The simple objects can be user defined objects or standard objects from the LEND storage layer (described in Section 1.2 and [Che92]). The value of a component of a complex object is accessed using the dot (.) notation (see Section 2.1.4). The dot notation is similar to that used in the LEND query language [Che92] and other database systems [SK91b]. Objects have associated methods (program code) that define their behavior. These methods are defined for the class when the database is created.

3.3 GOAL Language Description

GOAL language statements are of four types: *comment*, *declaration*, *assignment* and *retrieval*. Comments are statements that begin with two slash signs, just as in the *C++* language. Statements are terminated with the semicolon sign. It is possible to declare a variable and assign a GOAL expression to that variable in one statement. Variables were added to GOAL (and also to LEND) to help express complex queries.

GOAL expressions apply GOAL *operators* on sets of graphs, paths, and nodes to create objects of any of these three types. Retrieval statements consist of a GOAL expression

followed by a period and the name of a function. The retrieval statements can be used to create a set of objects corresponding to the GOAL expression and apply the given function to the resulting set of objects. Such a function is `Print()`, which displays on the screen either the objects in the set, or required parts of the objects in the set. Other functions can be defined for a set of objects, in which case the function is applied on each object in the set. If no function is specified in the retrieval statement the `Print()` function is applied to the retrieved set of objects.

Examples of possible methods are:

- `Display()`: which can be used to display multimedia objects on the screen;
- `Count()`: which can be used to display the number of objects in the retrieved set instead of displaying all the objects;
- `SortAndPrint()`: which accepts the names of components of objects as arguments, sorts the objects based on the value of the component corresponding to the first argument and prints the values of the components corresponding to the other arguments; and
- `Print()`: which accepts names of components of objects as arguments and prints only the values of those components instead of all the parts of the object.

In our implementation of GOAL, discussed later in Chapter 4, we have programmed the `Print()` and `Count()` functions; others can be added as needed.

Query language operators have arguments that are sets of graphs, paths, or nodes. In the following sections query language operators begin with lower-case letters and their arguments begin with capital letters. 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. We first describe the GOAL *set operators* and then the GOAL *type operators* to create *pathSets*, *nodeSets* and *graphSets*, respectively. The following sections

define the operators, motivate them, and illustrate their use with examples. Each subsection first describes a set of operators and then gives examples of queries illustrating the use of the operators.

The examples use a hypothetical bibliographic database to illustrate the use of the different operators and the data modeling and querying possibilities of GOAL. The bibliographic database that we consider has been modeled as an information graph made up of **Documents**, **Authors** and **Keywords** nodes (see Figure 3.1). A document node contains the title of the document and the year in which it was written. For *Authors* nodes, the names of the authors and the organization that they belong to are stored. For *Keywords* nodes the keyword string is stored. 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 indexed by the keyword. The **HasKeyword** arc contains the document weight (**DocWt**) of the document with respect to the keyword. Each keyword node may be connected to other keyword nodes that are synonyms of the keyword with **Synonym** arcs.

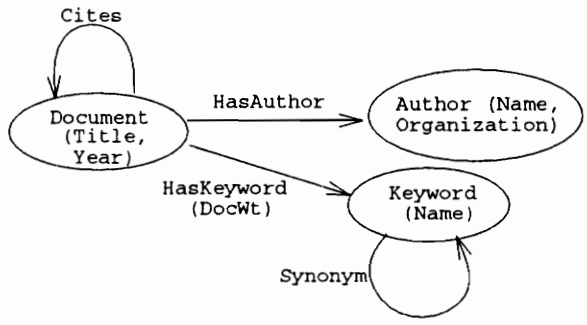

Figure 3.1: Sample information graph

3.3.1 GOAL Set Operators

Definition

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 GOAL objects of the same type.

- **union(SetList):** This operator finds the set union of the sets in the list.
- **intersect(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. The condition, usually expressed in terms of a function, can optionally have arguments that are either names of components of the objects in the set (for example, the source of a path object) or constant values (for example, the value "2") that will be used when selecting the objects.

  A function appearing in the *Condition* should be defined for one of the following two classes:

  - the class corresponding to an object in the given set, or
  - the class corresponding to the whole set of objects.

  In the first case, the function will return *true* if the object satisfies the given condition, and *false* otherwise. In the second case, the function will select from the given set that subset satisfying the condition. For example, if a function is defined for the class *path* associated with a path object, it will return *true* if the given path satisfies the condition. On the other hand, if a function is defined for the class *pathSet* associated with a set of paths, it will return the appropriate subset of the paths.

These functions will have to be defined by the database manager when designing the classes for the database. Inheritance and virtual functions can be used to ease the development process. The query processor will call the functions when processing the queries to find the set of objects retrieved using the *select* operator.

GOAL allows any arbitrary condition to be used as an argument for the *select* operator; the LEND query language allowed only conditions that compared components of objects with constant values or other components of the objects.

The select condition `partEqConst(ComponentName, Const)` is used very often. It has two arguments: a component name and a constant. It returns "`true`" if the value of the component for the object is equal to the given constant value. In order to make the condition `partEqConst` easier to use, it can also be expressed as `ComponentName == Const` (see queries listed below).

- **Set.aggregate([GroupList], AggregateFunction):** Here, *GroupList* is a comma separated list of component names of the objects in the set and *AggregateFunction* is the name of a function that can be applied to sets of objects present in the set. The aggregate operator is evaluated by first grouping the objects in the set into subsets having identical values for the objects in GroupList. The aggregate function is then applied to each subset to create one resulting object corresponding to each subset. The keyword *All* can be used in place of the GroupList in order to specify that all the objects in the set should form one group.

  The *AggregateFunction* can optionally have arguments that are either names of components of the objects in the given set or constant values. (See also the description above of the *select* operator.)

  A function corresponding to the aggregate function appearing in the query should be defined for the class corresponding to the objects in the given set. This function will be defined by the database manager when designing the classes for the database.

The function will be used by the query processor when aggregating objects. Each object in the given set will be considered in turn, and the result set will be updated if the function associated with the object returns `true`. The function will return `true` if it is necessary to update the result set because of the object being considered. When returning `true`, the function will also return an aggregated object that will have to be added to the result set and a set of objects that have to be deleted from the result set as a result of adding the aggregated object.

Aggregate operators in relational databases (see Section 2.1.3) made relational database systems more useful because they could be used to generate statistical information often required for reports. The aggregate operator provides the same function in GOAL. In addition, GOAL allows this operator to be applied on sets of paths. This makes it possible to express a large number of useful queries in GOAL.

This operator can be used to group paths having the same source and to aggregate the information in the subsets thus created. For example, the aggregate function `Count(NumInSet)` can be used to find the number of paths in the set that start from the same node and assign this number to a part called `NumInSet` of the path object. It also is possible to find the shortest or longest paths by retaining only the paths that have the minimum or maximum length using the aggregate functions `Min(length())` and `Max(length())`. Similar useful queries can be expressed by grouping the paths leading to the same sink node or by grouping paths having the same source and sink nodes.

- **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 for a graph, arc, or node object and the resulting set will be a set of graphs, paths and nodes, respectively. The result will be all of the objects in the information graph that belong to that class. The *ClassName* also can be the name of a public base class associated with the object. For example, an *employee* class is a public base class of a *salesman* class; it is possible

to retrieve only salesmen by specifying the *className* as *salesman* and retrieve all employees including salesmen by specifying the *className* as *employee*. The query processor will recursively check the hierarchy for the different types of objects in the database to see if they belong to the specified class.

This approach is different from the POSTGRES [SK91a] approach where it is possible to retrieve objects belonging to a particular class by specifying the name of the class, and to retrieve objects that belong to a class derived from a public base class by following the class name by an asterisk. For example, a search for objects of type *employee* in POSTGRES will result in retrieving all employees and not special types of employees such as salesmen. On the other hand, a search for objects of type *employee* * will retrieve salesmen and all other types of employees. We believe our interpretation is suited for LEND because we allow public inheritance, and public inheritance by definition implies an "isa" relationship. Thus, in GOAL it is possible to find employees who are not salesmen, using the query `diff(employee, salesman)`.

- **SetVariableName:** The set of objects retrieved by specifying a variable name will be those assigned to the given variable. Note that a variable must be defined and assigned a GOAL expression before it can be used.

Examples

In this section we give examples of queries illustrating the use of the GOAL Set operators.

1. Display all the keyword nodes.

   GOAL Query:

   ```
   Keywords.Print();
   ```

   Comments: This GOAL query applies the `Print()` function to all the node objects in the `Keywords` class. The `Print()` function can be omitted from this query because it is the default function that is used. Hence, we will omit it in the following examples.

2. Find the keyword node(s) corresponding to the keyword *software engineering.*

   GOAL Query:

   ```
   Keywords.select(name == "software engineering");
   ```

   Comments: The GOAL statement uses the *select* operator to select the keyword node(s) with the given name and then prints the information stored in the node.

3. If variables D_A, D_B, D_C and D_D have been assigned to the set of document nodes that are connected to the keywords A, B, C and D respectively, find the set of documents that are relevant to the query "((A but not B) or C) and D".

   GOAL Query:

   ```
   intersect(union(diff(D_A, D_B), D_C)), D_D);
   ```

   Comments: The GOAL query uses the *union*, *intersect* and *diff* operators to find the required set of documents. Document nodes that are connected to given keyword nodes can be found using path operators given in the next section.

4. Find the number of documents in the database written each year.

   GOAL Query:

   ```
   Documents.aggregate([Year], Count(NumInSet)).
                              Print(Year, NumInSet);
   ```

   Comments: The GOAL query finds the set of document nodes and uses the *aggregate* operator to partition the set into subsets of documents written the same year. The Count() aggregate function is used to find the number of documents in each subset and to assign the result to the component NumInSet of the Documents object. The number of documents is then printed together with the year, for each subset.

The following sub-sections describe the GOAL type operators.

3.3.2 GOAL Path Operators

Definition

The following operators create sets of paths.

- **getPaths(Nodes, Paths, ... Nodes):** 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.

  For example, the statement `getPaths(`N_1, P_1, N_2, P_2, ..., N_{n+1}`)`, will retrieve a set of paths P such that a path p belonging to P is formed by joining paths p_1, p_2, ... p_n belonging to the sets P_1, P_2, ... P_n respectively, if for $1 \leq i \leq n$ the following conditions hold:

  - the source node of p_i belongs to the node set N_i;
  - the sink node of p_i belongs to the node set N_{i+1}; and
  - the sink node of the path p_i is the source node of the path p_{i+1}.

  The *getPaths* operator is a generalized form of the *virtualPaths* operator of the LEND query language. The *virtualPaths* operator allowed only arc labels as arguments; the *getPaths* operator allows sets of paths as arguments. This generalization does not add to the power of the language but should make the language easier to use.

- **pCompose(Paths, ... Paths):** This operator creates paths that are formed by joining paths in the comma separated list of paths. A path in the first set 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. *PathsList* is a comma separated list of one or more paths.

  For example, the statement `pCompose(`P_1, P_2, ..., P_n`)`, will retrieve a set of paths P such that a path p belonging to P is formed by joining a path p_i belonging to the

set P_i to a path p_{i+1} in the set P_{i+1} if the sink node of the path p_i is the source node of the path p_{i+1} for all $1 \leq i \leq n$.

The *pCompose* operator is a generalized form of the *compose* operator of the LEND query language. The *compose* operator can be used to join paths in two path sets. The *pCompose* operator can be used to join an arbitrary number of path sets. This generalization does not add to the power of the language but should make GOAL easier to use.

- **allPaths(Nodes, Paths, Nodes):** This operator repeatedly joins paths in the given set of paths to find all paths that connect nodes in the first set of nodes to nodes in the second set of nodes. The resulting set of paths may be of varying lengths because first, the set of paths can be of varying length and second, there is no restriction on the number of times the paths are joined. Hence, they may be joined as many times as possible to generate paths that satisfy the conditions. A path in the resulting set cannot contain the same arc more than once, though it can contain the same node more than once.

  For example, the statement `allPaths(`N_1`,` Q`,` N_2`)`, will retrieve a set of paths P such that a path p belonging to P is formed by joining a sequence of paths $q_1, q_2, \ldots,$ q_n belonging to the path set Q, if the following conditions hold:

  - The source and the sink of the path p belong to the node sets N_1 and N_2, respectively.
  - The sink of the path q_i is the source node of the path q_{i+1} for all $1 \leq i \leq n$.
  - No arc occurs more than once in the path p.

  The *allPaths* operator is equivalent to the LEND closure operator. The *select* operator can be applied on the set of paths retrieved by the *allPaths* operator to restrict the lengths of paths retrieved. This will provide the same function as the LEND *iterate* operator.

Finding the shortest path, longest path and other similar queries can be expressed by following the *allPaths* operator with the *aggregate* operator. The *allPaths* operator is very expensive. Useful optimization techniques allow execution of the two operators together, leading to faster execution in LEND (see Chapter 4).

- **Graphs.graphGetArcs():** Given a set of graphs, this operator returns a set of paths, each made up of only one arc, that are present in the given graph set.

- **Paths.pathGetArcs([DigitsSeq]):** This operator returns the set of paths of length one that are made up of the arcs in the given set of paths, 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 [$Digit$]. For example, if the digit sequence is [2–5], it will return arcs at positions 2, 3, 4 or 5; if it is [3, 5], it will return arcs at position 3 and 5; if it is [3], it will return arcs at position 3.

  The LEND query language did not include an operator equivalent to the *pathGetArcs* operator to allow us to retrieve arcs from paths that occur at given positions. In contrast the language did allow retrieval of nodes that occurred at a given position in the path using the *forward* and *backward* operators. The *pathGetArcs* operator was added to GOAL to make it regular.

- **←(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.

  For each path $p = (n_1, a_1, n_2, \ldots, n_n)$ belonging to the path set *Paths* there will be a corresponding path in the resulting set of the form $(n_n, a_{n-1}, n_{n-1}, \ldots, n_1)$.

  The ← or *reverseArrow* operator is not present in the LEND query language. It is included in GOAL because it helps us to express queries naturally (see for example query 1 given below and Chapter 5.) It makes it possible to traverse the graph by following the arcs (or paths), starting at the sink and ending at the source.

Examples

1. Find all paths that link the document with title **XXX** to the nodes of documents that cite it.

   GOAL Query:

   ```
   pathSet P = getPaths(Documents.select(Title == "XXX"),
                                ←(Cites), Documents);
   ```

   Comments: The *getPaths* operator is used to find 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.

2. Find the paths that connect the document node representing the document with title **XXX** with the author nodes of the documents that cite it.

   GOAL Query:

   ```
   pCompose(P, HasAuthor);
   ```

   Comments: "`HasAuthor`" is the name of the class of arcs that link document nodes to the corresponding author nodes. As a result, the set of paths associated with the second argument of the *pCompose* operator will be a set of paths linking document nodes to the corresponding author nodes. The *pCompose* operator is used to join the paths created in the previous example (linking the document with title XXX to the documents that cite it), to the set of paths retrieved using *HasAuthor* as an operand.

3. Find the paths made up of `Cites` arcs that are of length at most 3 and are connected to the document node with title **XXX**.

GOAL Query:

```
pathSet Q = allPaths( Documents.select(Title == "XXX"),
                      Cites, Documents).
                      select(Length_LT( "4"));
```

Comments: In this example we use the *allPaths* operator to find all paths that start from the given document node and follow arcs of type `Cites`. We use the *select* operator to restrict the lengths of the paths to less than or equal to 3. In this example we have assumed that a path object has a member function `Length_LT()` that returns the value "`True`" if the length of the path is less than the value of the argument.

4. If `G` is a graph containing the set of document nodes, find all the `HasAuthor` arcs in `G`.

   GOAL Query:

   ```
   G.graphGetArcs().select(ClassName == "HasAuthor");
   ```

   Comments: In this query we use the *graphGetArcs* operator to find the required set of arcs from the set of graphs previously created.

5. Given the paths assigned to the variable `Q` in query 3, find the arcs that are directly connected to the document with title `XXX`.

   GOAL Query:

   ```
   Q.pathGetArcs([1]);
   ```

   Comments: In this query we use the set of paths assigned to the variable `Q` to find the required information. The required set of paths will consist of paths of length one. This set of paths is the same as that assigned to the variable `P` in query 1.

3.3.3 GOAL 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.graphGetNodes():** This operator returns the set of nodes that belong to the graphs in the given set.

- **Paths.pathGetNodes([DigitsSeq]):** This operator returns the set of arcs in the given paths 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*]. See also the description above of the *pathGetArcs* operator.

  Note that the LEND language has *forward* and *backward* operators that allow us to select nodes from paths that occur at given positions starting from the source or sink respectively. The GOAL *pathGetNodes* operator provides the same function as the *forward* operator. There is no operator equivalent to the LEND *backward* operator because its function is satisfied by applying $\leftarrow$, the *reverseArrow* operator, and then the *pathsGetNodes* operator on the resulting path.

The *source* and *sink* operators are not really required because their function is satisfied by the *pathGetNodes* and the $\leftarrow$ (*reverseArrow*) operator. They have been added nevertheless because they are frequently used operations. The GOAL *sink* operator allows a set of paths as arguments and hence can be considered as a generalized form of the LEND *sinks* operator which allowed arc labels to be used as an argument.

Examples

1. Find the authors that have published papers in the field of software engineering.

GOAL Query:

```
nodeSet K_SW = Keywords.select(Name == "Software Engineering");
pathSet A_SW = getPaths(K_SW, ←(HasKeyword), Documents,
                        HasAuthor, Authors).sink();
```

or

```
pathSet A_SW = getPaths(Authors, ←(HasAuthor), Documents,
                        HasKeyword, K_SW.)source();
```

Comments: The first query statement uses the *select* operator to find the keyword node(s) corresponding to **Software Engineering** (see Query 2 in Section 3.3.1). The second statement uses the *getPaths* operator to find the paths linking these keyword node(s) to the author nodes, and then applies the *sink()* operator on the resulting set of paths to select only the author nodes from the paths.

The third statement finds the same set of nodes, but does so by specifying the path pattern starting from the author nodes. The required set of author nodes for this statement will be the source nodes of the paths created.

Since the result of the operation is a set of nodes, the query processor will ensure that the result does not contain the same node more than once.

2. If **G** is a graph containing the set of document nodes, find the **Authors** nodes in **G**.

GOAL Query:

```
G.graphGetNodes(Authors);
```

Comments: In this query we use the *graphGetNodes* operator to find the required set of nodes from the set of graphs previously created.

3. Given the paths assigned to the variable `P` in query 1 Section 3.3.2, find the nodes that are directly connected to the document with title **XXX**.

   GOAL Query:

   ```
   P.pathGetNodes([2]);
   ```

   Comments: The given document is the source node of all the paths belonging to the set assigned to the variable `P`. The nodes directly connected to the given document will be at position 2 in the paths.

3.3.4 GOAL Graph Operators

The following operators create sets of graphs.

Definition

- **getGraphs(Paths):** Given a set of paths, this operator returns a set of graphs which contain the nodes and arcs corresponding to a path from the given set.
- **getGraphs(Nodes):** Given a set of nodes, this operator returns a set of graphs, each containing one node from the given set.
- **gUnion(Graphs, Graphs):** Given two sets of graphs, this operator returns a set of graphs formed as follows:
  - Consider all pairs of graphs (g_1, g_2), such that the first graph g_1 is from the first set of graphs and the second graph g_2 belongs to the second set.
  - For each such pair, return a graph g containing the nodes and arcs present in either g_1 or g_2.
- **gIntersect(Graphs, Graphs):** Given two sets of graphs, this operator returns a set of graphs formed as follows:

- Consider all pairs of graphs (g_1, g_2), such that the first graph g_1 is from the first set of graphs and the second graph belongs to the second set.
- For each such pair, return a graph g containing the nodes and arcs present in both g_1 and g_2.

- **gDiff(Graphs, Graphs):** Given two sets of graphs, this operator returns a set of graphs formed as follows:
  - Consider all pairs of graphs (g_1, g_2), such that the first graph g_1 is from the first set of graphs and the second graph belongs to the second set.
  - For each such pair, return a graph g containing the nodes and arcs present in g_1 but not in g_2.

The *gUnion*, *gIntersect* and the *gDiff* operators are expensive because they require that each graph in the first set be compared with all graphs in the second set. If the first set of graphs contains m graphs and the second contains n graphs then there will be $m \times n$ comparisons. We believe that these operators will be used frequently with the select operator applied to the resulting set of graphs, to restrict their number. When the two operators are applied in sequence it is possible to apply the selection function first and then perform the set operations for those graphs that satisfy the criteria.

Examples

1. Find pairs of authors such that one author has published papers in the field of software engineering and the second author has published papers in the field of information retrieval, where both authors belong to the same organization.

   GOAL Query:

```
nodeSet K_SW = Keywords.select(Name == "Software Engineering");
nodeSet K_IR = Keywords.select(Name == "Information Retrieval");
```

```
nodeSet A_SW = getPaths(K_SW, ←(HasKeyword),
    Documents, HasAuthor, Authors).sink();
nodeSet A_IR = getPaths(K_IR, ←(HasKeyword),
    Documents, HasAuthor, Authors).sink();

gUnion(getGraphs(A_SW), getGraphs(A_IR)).
    select((NumNodes_EQ("2") AND
        CheckSameNodes(Authors.Organization) == "True");
```

Comments: The first two statements are used to find the keyword nodes corresponding to "Software Engineering" and "Information Retrieval". The next two statements are used to find the author nodes that are connected to the two keyword nodes. The last statement uses the *gUnion* operator to find graphs containing nodes from the two sets of author nodes. The select operator is used to restrict the graphs to those that have two nodes and have the same organization for each node. Here we make use of two functions defined on graphs. `NumNodes_EQ()` will return `true` if the number of nodes in the graph is equal to the value of the argument and `false` otherwise. The function `CheckSameNodes()` returns `true` if all the `Authors` nodes in the graph have the same organization and `false` otherwise.

2. If a variable `X_Y` is assigned the set of paths that can be followed to reach keyword `Y` from keyword `X` following `synonym` arcs, and variable `U_V` has been assigned the set of paths that can be followed to reach keyword `V` starting from keyword `U`, find common words that lie on these paths. For example, Figure 3.2 shows a sample graph where the required keyword nodes have been highlighted.

GOAL Query:

```
graphSet G_XY = getGraphs(X_Y);
graphSet G_UV = getGraphs(U_V);
gIntersect(G_XY, G_UV).Print(Words);
```

Comments: To express this query we use the *getGraphs* operator to create sets of graphs corresponding to each path. We then use the *gIntersect* operator to find graphs that contain nodes and arcs in a graph in the first set as well as some other graph in the second set.

Table 3.1 illustrates the processing of this query for the graph shown in Figure 3.2.

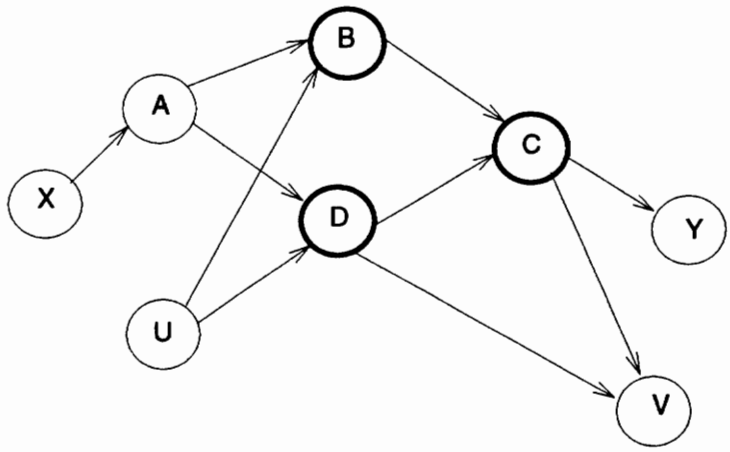

Figure 3.2: Graph for the common words query

3. If the variable `P_Q` has been assigned the set of paths linking the documents in set `P` to those in set `Q` and similarly the variable `R_S` has been assigned the set of paths linking the documents in set `R` to those in set `S`, find the nodes and arcs in the graph that lie on paths in the set `P_Q` but do not lie on the paths `R_S`.

Table 3.1: Trace for processing the common words query

Type	Set	Value
pathSet	X_Y	{X-A-B-C-Y, X-A-D-C-Y}
pathSet	U_V	{U-B-C-V, U-D-V} U-D-C-V}
graphSet	G_XY = getGraphs(X_Y)	{[X, A, B, C, Y], [X-A, A-B, B-C, C-Y]}, {[X, A, D, C, Y], [X-A, A-D, D-C, C-Y]}
graphSet	G_UV = getGraphs(U_V)	{[U, B, C, V], [U-B, B-C, C-V]}, {[U, D, V], [U-D, D-V]} {[U, D, C, V], [U-D, D-C, D-V]}
graphSet	gIntersect(G_XY, G_UV)	{[B, C], [B-C]}, {[C], [ ]}, {[D], [ ]} {[C, D], [C-D]}

GOAL Query:

```
gDiff(getGraphs(P_Q), getGraphs(R_S));
```

Comments: A query of this form will be useful if the sets `P`, `Q`, `R` and `S` have been assigned the set of documents written in the years 1981, 1984, 1982 and 1983 respectively. The retrieved set of graphs will contain document nodes that will be those paths that link documents written in the year 1981 to those in the year 1984 but do not contain any nodes for documents that were written in the years 1982 and 1983.

3.4 Comparison between GOAL and other Query Languages

One of the main differences between GOAL and the other query languages described in Chapter 2 is that GOAL only allows us to retrieve data from the database. Other languages are designed to allow users to modify the database by adding, changing and deleting information. GOAL could be extended similarly, but that would be the subject

of another study. Such a study could build upon the fact that GOAL allows us to create intermediate objects such as paths and graphs that are composed of other objects, though these objects cannot be saved.

Other differences are as follows:

- **Hierarchical and Network**
  - The GOAL database model is similar to the hierarchical and network models and especially to the network model because of the underlying graph structure. But this is where the similarity ends.
  - GOAL does not have restrictions on the types of links. As explained in Section 1.2.1 and 1.2.2, a node in the hierarchical model cannot have more than one parent, and most of the database systems based on the network model do not support graphs with cycles.
  - The arcs in the GOAL query model can be objects with properties associated with them. For example, in the airline database, an arc linking two cities can be associated with the time taken to fly between the two cities. This is not possible in hierarchical and network models.
  - Arcs in GOAL also can belong to different class hierarchies, which is not possible for the arcs in the hierarchical and network database systems. This feature increases the data modeling capabilities of the language.
  - The query languages based on the hierarchical and network models have operators that can be applied on nodes (or records) whereas the GOAL operators are set-oriented. As explained in Section 2.1, a set consists of a number of records and hence a set operation is equivalent to a sequence of record-level operations. Such set-operations should be easier to use.

- **Relational**
  - GOAL allows us to manipulate complex objects and the information graph captures much more semantics of the application than does a relational database.
  - GOAL also allows us to express generalized linear recursive queries in an intuitive manner. As explained in Section 2.3, these queries cannot be expressed using traditional query languages for relational databases.
  - A relation in a relational database can correspond to a node in the information graph. The relations represented by foreign keys in the relational database can be represented as arcs linking pairs of nodes.
- **Object-Oriented and Extended Relational**
  - The nodes and arcs of a GOAL information graph can be objects made up of other objects (see Section 3.1.1); thus complex objects are represented as nested objects in the GOAL information graph model. The information graph model also allows relationships between objects to be represented by directed arcs linking those objects.
  - The arcs in the information graph can be powerful structures because they are objects that can belong to class hierarchies and have member variables and member functions. Thus the arcs can be used to capture information about the relationships.
  - Traditional object-oriented databases and the extended relational databases such as those described in Section 2.3 provide nesting of objects as a method to represent complex objects as well as other relations between objects. We believe nesting of objects is a natural representation for modeling complex objects but it is not a natural representation for representing other relationships.

- **GraphLog**
  - GraphLog provides a graphical query language while GOAL provides a command language that can be embedded within a program.
  - GraphLog cannot be considered as an object-oriented database because it does not provide for methods on objects to be defined.
  - Both GraphLog and GOAL allow aggregation of information along paths as well as multiple paths leading to the same source and sink nodes. GOAL interprets the aggregation of information along a path as a property of the path while GraphLog creates a new edge with the aggregated information stored as an attribute of the edge.
  - It is not possible to use GraphLog to find the paths that connect two given nodes. The GOAL *getPaths*, *pCompose* and *allPaths* operators retrieve sets of paths. These paths may be useful to express queries such as "Find all the paths that can be taken to reach New York from Chicago".
- **GOOD**
  - GOOD provides a graphical query language while GOAL provides a command language that can be embedded within a program.
  - The GOOD model is powerful because it has the computational power of a Turing machine while the GOAL language has been designed to be limited in its power. This has been done in order to ensure that the GOAL queries can be executed efficiently.
  - The GOOD model stores information about the class hierarchy as parts of the database graph. It also assumes that the nodes are not made up of other objects. In the LEND information graph model, class hierarchy and the nesting relationships are stored using *C++* constructs and not as relations between objects.

- **LEND Query Language**

  We have described the differences between the LEND query language and GOAL in Sections 3.1 and 3.2. Here we summarize the differences by describing how we have overcome the limitations of the LEND query language given in Section 2.2.3.

  - GOAL can be used to retrieve sets of unconnected graphs. As can be seen from the examples in Section 3.2.3, there are many queries that correspond to unconnected graph patterns.
  - Recall that the LEND query language allows retrieval of paths based on restrictions on the source nodes, sink nodes and the lengths of the paths. GOAL provides the *select* operator that can be used to select paths based on any characteristic of the path. Users are free to define any conditions on paths that they believe will be useful for their application.

    Also, GOAL provides the *aggregate* operator that can be used to aggregate information in a set of objects.
  - Other extensions to the LEND query language were made in order to make GOAL easy to use. These extensions were the generalization of the *select*, *getPaths* and *pCompose* operators and the addition of the $\leftarrow$ (*reverseArrow*) operator.

Chapter 4

GOAL Implementation

This chapter describes the implementation of GOAL. It contains three sections. The first section describes the architecture of the GOAL query processor. The second section describes the processing of the *getPaths*, *pCompose* and *allPaths* operators. It also describes the optimization techniques used to execute queries containing these operators. The last section summarizes the first two sections, lists conditions under which the optimizations are not effective and gives recommendations to overcome such situations as well as to improve performance of other operators.

4.1 Architecture of the GOAL Query Processor

The GOAL query processor has been written in *Gnu G++*. It assumes that the database is a LEND database [Che92] and it makes use of the LEND libraries to retrieve the data.

The database should be loaded into LEND before the query processor starts executing the queries. Also, the GOAL dictionary should be supplied with the name of the information graph and the names of the different node and arc objects in the information graph. This GOAL dictionary will be maintained by the query processor and will contain information about words occurring in the queries that are not GOAL operator names. Examples of such types of words are: variable names, component names of complex objects, function names, and constants used in the selection expression or as arguments to functions.

The query processor accepts GOAL query statements and executes them. In the current implementation it works as an interpreter. As do most interpreters, the query processing is done in three steps [ASU86] (see Figure 4.1):

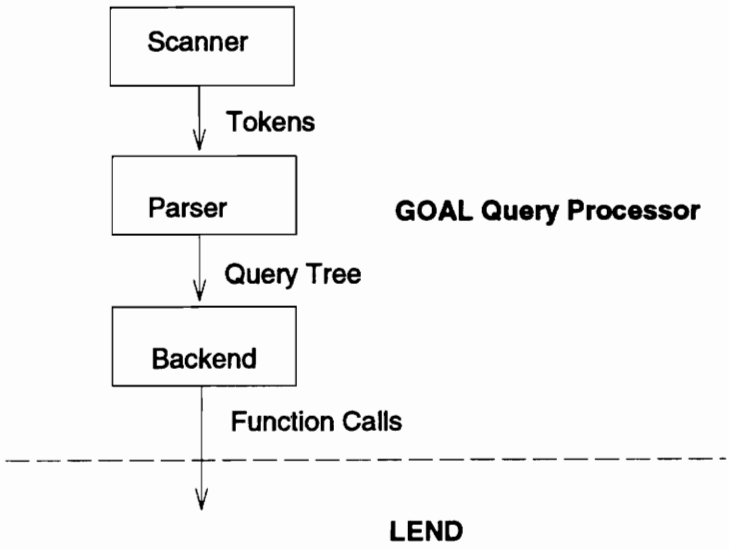

Figure 4.1: Architecture of the GOAL query processor

1. The query is first parsed by a scanner which is a lexical analyzer generated using the *Gnu flex* lexical analyzer generator [Pax90]. The scanner recognizes keywords and words present in the dictionary and converts the incoming stream of GOAL query statements to a corresponding stream of tokens. The scanner adds all unrecognized words to the GOAL dictionary.

2. The tokens identified by the scanner are fed into a parser that recognizes query language statements. The parser has been generated using *bison* [DS92] which is the *Gnu* version of *yacc*. When the parser recognizes a declaration statement which defines a *graph*, *path* or *node* variable, it updates the type of the variable in the dictionary. For each retrieval statement it creates a query tree corresponding to the statement. The query tree is formed using *C++* objects corresponding to the operators and operands.

   A node of the query tree corresponding to an operator is connected to the nodes corresponding to its operands. An operand can in turn be another GOAL operator, and hence may be connected to other nodes. For example, the query tree shown in Figure 4.2 corresponds to the statement:

"`Documents.select(Title == "XXX").Print();`.

The root of the query tree is the *nodesRet* operator because the query corresponds to a node retrieval statement. The two nodes connected to the root correspond to the two operands of the node retrieval statement. The first operand corresponds to the GOAL operator used to retrieve the required set of nodes, which in this case is the *select* operator, and the second operand is the name of the method that must be applied on the retrieved set, in this case `Print()`.

The *select* operator has two operands (see Section 3.3.1): a GOAL operator that retrieves a set of nodes, and a condition. In this case the first operand is the name of the collection, that is `Documents`, which is stored as part of the node. The condition for this query is of the form *partName == constant*; hence the node is labeled `partEqConst` and the values `title` and `XXX` are stored with the node.

When the query processor recognizes an assignment statement it creates a query tree corresponding to the right hand side (RHS) of the assignment statement and updates the dictionary by storing a pointer to the root of the query tree in the left hand side (LHS) variable entry.

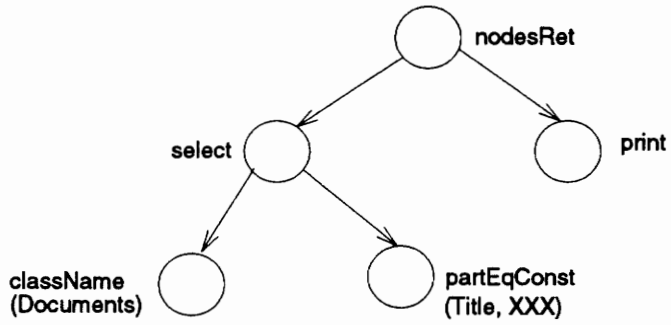

Figure 4.2: Query tree for: `Documents.select( Title == "XXX").Print();`

Variables in retrieval statements are replaced by the corresponding RHS values assigned to them. The parser also recognizes sequences of operators that can be executed more efficiently if executed simultaneously. It replaces such sequences by single operators. The sequences of operators that can be optimized in this way have been listed in the next section.

Any syntax errors in the queries are recognized by the parser. On finding such an error, the query processor reports the line number where the error was found and stops executing.

3. Declaration and Assignment statements result only in dictionary updates. Retrieval statements are processed by the back-end by calling the overloaded *exec()* member function of the object corresponding to the root of the query tree for the statement. The root of the query tree corresponds to a graph, path or node retrieval operator. Recall that a retrieval statement consists of a sequence of GOAL operators to create the required set of objects followed by the name of the method that should be applied on the retrieved set. The *exec()* functions of the retrieval statements first retrieve the objects and then apply the given method on the retrieved set. If no method is present the `Print()` method is applied on each object that is retrieved.

   In order to retrieve the objects created by the GOAL operators, the overloaded *get_collections()* member function is applied to the object corresponding to each operator. The *get_collections()* member function is defined for all objects corresponding to the GOAL operators. The function either recursively applies the *get_collections()* function to its operands and uses the result to create the desired set, or uses the operands to call the appropriate LEND routines to retrieve the desired set of objects. Figure 4.3 shows a trace of the execution of the query shown in Figure 4.2.

   Semantic errors in the queries are reported by the query processor when they are found, and after an error has been found, the query is not processed any further.

The sets of objects created during the execution of the queries are stored in main mem-

```
nodesRet.exec() {
    requiredNodes = select.get_collections() {
        allNodes = className.get_collections(Documents);
        return (partEqConst.findMatches(allNodes));
    }
    requiredNodes.print();
}
```

Figure 4.3: Trace for processing query: `Documents.select(Title == "XXX").Print();`

ory. The objects are deleted after the retrieval statements have been processed. Indexes that are used for faster processing of the queries are also created as required.

The next section describes the processing of the *getPaths*, *pCompose* and the *allPaths* operators.

4.2 Processing of the *getPaths*, *pCompose* and the *allPaths* operators

A study of different applications was conducted in order to find bottlenecks for processing GOAL queries. The applications considered were real-world applications from different research projects at VPI&SU and from the literature that was reviewed. This study was not restricted to IR applications; rather information about a wide range of applications was collected to ensure that the study was more complete. Graph schemas were then designed for the applications and the queries were expressed using GOAL language statements. (See [Bet92] for the list of queries.)

The GOAL language statements for the queries were studied and it was found that the *getPaths*, *pCompose* and the *allPaths* operators are frequently used as well as expensive to process. Hence, optimization techniques for these three operators were developed. Section 4.2.1 describes the basic algorithms that can be used for processing the three operators and the following sub-sections describe the optimization techniques used to improve performance.

4.2.1 Algorithms for processing the *getPaths*, *pCompose* and the *allPaths* operators

Recall that the *getPaths*, *pCompose* and *allPaths* operations have the following operands:

- *getPaths*:

  An alternating sequence of sets of nodes and paths N_1, P_1, N_2, P_2, ..., N_n, P_n, N_{n+1} starting and ending with sets of nodes.

- *pCompose*:

  A sequence of sets of paths P_1, P_2, ..., P_n

- *allPaths*:

  Two sets of nodes and a set of paths: N_1, N_2 and P

To process the *getPaths* operator, the query processor has to find 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. To find this set of paths, the query processor starts from either the first or the last set of nodes and creates paths starting / ending from the nodes in the selected set. These paths are created by following arcs from the node, one step at a time, using a depth-first search technique [Tar72] until it is not necessary or possible to extend the paths any further.

The *pCompose* operator requires that paths in the given sets of paths be joined in the given order to form new paths. This operator is processed using the same technique used for processing the *getPaths* operator, i.e., a path from the first (or last) set of paths is extended in a depth-first manner till it is not necessary or possible to extend it further.

To find the paths retrieved by the *allPaths* operation, it is necessary to repeatedly join the paths in the set P so that the resulting paths start and end at nodes from the first and last set of nodes (N_1 and N_2) respectively. The resulting set of paths can contain the same node more than once but cannot contain the same arc more than once.

In order to find the paths generated by the *allPaths* operator we use the algorithm described in [Pur70]. To use this algorithm we first need to do the following.

1. Find the strongly connected components (SCCs) in the graph using Tarjan's [Tar72] depth-first technique.
2. Assign IDs to each SCC.
3. Reduce the original graph to one containing only the SCCs. Store pointers to the nodes within each SCC along with the SCC ID.
4. Map the arcs in the graph to those in the SCCs, storing only the arcs that link different SCCs. Store arcs that map nodes within the SCC separately.

The graph containing only the SCCs is a directed acyclic graph (DAG). Since a DAG does not have cycles, it is possible to use depth-first-search to retrieve the required set of paths [Pur70]. Note that the SCCs can be found once while loading the database and then stored for all subsequent query processing.

In the following discussion we describe the algorithm to find all the paths for a DAG. Each such path can be easily converted to a set of paths of the original graph by following pointers to the nodes and arcs within the SCCs.

To find all the paths, we again use the depth-first search technique as we did when processing the *getPaths* and *pCompose* operators. But for the *allPaths* operator we store all intermediate paths found. We improve performance by searching the retrieved sets of paths to see if paths starting from a given node have been found, and in that case join the paths that have been found to the intermediate path instead of traversing the successor nodes and arcs again.

To implement the above algorithms we have defined the *get_pathsStarting()* member functions for all the operators that retrieve sets of paths. These member functions retrieve only the set of paths starting from a given node and are used to extend a path.

It is possible that the *allPaths* query will generate an exponential number of paths. The query processor has been set up to stop generating paths after the number of paths generated exceeds a predefined limit. Other techniques that can be used to overcome such situations are discussed in Section 4.4.

In the following sections we describe the optimization techniques to process queries with these three operators.

4.2.2 Query Tree Transformations

A common query optimization technique is to transform a given query into an equivalent query which can be executed more efficiently [JK84]. Transformations can be performed in order to reduce the amount of redundancy in the queries or to replace a sequence of operators by another sequence of operator(s) that calls for less processing and/or fewer disk accesses.

The list of queries [Bet92] revealed that the *getPaths*, *pCompose* and the *allPaths* operators are frequently followed by the *select* and/or the *aggregate* operators. In such cases, instead of first evaluating one of the *getPaths*, *pCompose* and *allPaths* operators and then applying the selection condition and aggregate function to the retrieved set, it was found that it is possible to reduce the space required for storing the paths by applying the selection condition and the aggregate function on the paths as they are retrieved. It was also found that for certain selection conditions and aggregate functions, it is possible to use information about the aggregate function, selection condition and the paths already generated to trim the search space (see Section 4.2.4). In this way we can save both execution time as well as the space required for storing the retrieved paths.

As a result of these observations it was decided to combine each of the *getPaths*, *pCompose* and *allPaths* operations optionally followed by either the *select* and/or the *aggregate* operations to *getPathsAggSel*, *pComposeAggSel* and *allPathsAggSel* operations, respectively.

The *getPathsAggSel*, *pComposeAggSel* and *allPathsAggSel* operators have the corresponding operands listed in Section 4.2.1 as well as the operands of the aggregate operator:

the aggregate function A together with a list of parts of the path G, and the condition C which is an operand of the *select* operator. If the aggregate operator and/or the select operator do not follow the *getPaths*, *pCompose* and the *allPaths* operators, the operands corresponding to them in the *getPathsAggSel*, *pComposeAggSel* and *allPathsAggSel* operators will have null values.

4.2.3 Statistical Estimation

Statistical estimation is the basis for a variety of useful optimization techniques used in database system implementations [JK84]. If the required set of objects has to satisfy more than one condition, then it is possible to use statistical estimation to guess the number of objects that satisfy each condition, evaluate the condition that is expected to retrieve the least number of objects and apply the other conditions on this (hopefully) small set. This technique will require fewer objects to be retrieved and hence will reduce the I/O time and space required for the computation.

We assume that the algorithms described in Section 4.2.1 will require the traversal of fewer paths if the graph is traversed starting from the set of nodes / paths that is expected to have fewer objects. This assumption is based on the AI heuristic for search problems which states: *it is easier to move from a smaller set of states to a larger set of states* [Ric83].

At this time simple heuristics to estimate the size of the retrieved sets have been implemented. Tables 4.1 – 4.4 summarize the heuristics used for each type of GOAL operator and also give the reasoning behind the particular heuristics. The heuristics try to capture the difference in magnitudes of the set sizes. Improved heuristics will improve the performance of the optimizer.

4.2.4 Use of Meta-Information

A useful optimizing technique is to use information about the objects in the database, queries and / or objects already retrieved, to optimize processing of queries [JK84]. It is possible to use this "meta-information" about the selection condition, the aggregate function

Table 4.1: Estimation heuristics for Set operators

- est($union(S_1, S_2, \ldots, S_n)$) = est($S_1$) + est($S_2$) + ...+ est($S_n$)

  If the sets S_1, S_2, ..., S_n do not have any common elements then the resulting set will have all the elements in each of the sets.

- est($intersect(S_1, S_2, \ldots, S_n)$) = MIN(est($S_1$), est($S_2$), ..., est($S_n$))

  The resulting set cannot have more objects than the number of objects in the smallest set.

- est($diff(S_1, S_2)$) = est(S_1)

  If the sets S_1 and S_2 do not have any common elements then the resulting set will have all the elements in set S_1.

- est($S.select()$) = Select_Est

  The constant Select_Est has been defined to have value 1. This is because experience has led us to believe that the database will have collections with hundreds or thousands of objects. The number of objects retrieved by the *select* operator will be $o(1)$ when compared to the objects in the set S. The value of this constant can be changed as our experience with the queries and databases grows.

- est($S.aggregate([groupList], aggregateFunction())$) = est($S$)

  The number of objects retrieved by the *aggregate* operator is the same order of magnitude as the number of objects in the set S. This will be true if there are very few objects in the set S having the same values for the parts in groupList.

- est($ClassName$) = number of objects in the class

  This is the only operator that has the exact number of objects retrieved. The number of objects in the class is very easy to compute because it is stored together with the collection.

Table 4.2: Estimation heuristics for Path operators

- $\text{est}(getPathsAggSel(N_1, P_1, N_2, P_2, \ldots, P_n, N_{n+1}, GroupList, Agg_Function, Sel_Condition))$
  $= \frac{MIN(est(N_1), est(N_{n+1})) \times Get_Constant}{1 + Agg_Constant + Sel_Constant}$

  Get_Constant is a constant that estimates the number of paths starting or ending at a node in N_1 and N_{n+1} respectively. Applying the aggregate function and the selection condition reduces the number of paths retrieved. The constants *Agg_Constant* and *Sel_Constant* approximate the percentage of paths that will not be retrieved because of the aggregate function and the select condition. The value of the constant *Get_Constant* has been set to 100. The values of both the constants *Agg_Constant* and *Sel_Constant* will be set to 10 if the *Agg_Function* and the *Sel_Condition* are not *Null*, otherwise their values will be set to zero because it is not possible to use the aggregate function and select conditions to reduce the number of paths retrieved.

- $\text{est}(getPathsAggSel(N_1, P_1, N_2, P_2, \ldots, P_n, N_{n+1}, GroupList, Agg_Function, Sel_Condition))$
  $= \frac{MIN(est(N_1), est(N_{n+1})) \times PComp_Constant}{1 + Agg_Constant + Sel_Constant}$

  PComp_Constant is a constant that estimates the number of paths starting or ending with a path in P_1 and P_n respectively. Applying the aggregate function and the selection condition reduces the number of paths retrieved. The constants *Agg_Constant* and *Sel_Constant* approximate the percentage of paths that will not be retrieved because of the aggregate function and the select condition. The value of the constant *PComp_Constant* has been set to 100. The values of both the constants *Agg_Constant* and *Sel_Constant* will be set to 10 if the *Agg_Function* and the *Sel_Condition* are not *Null*, otherwise their values will be set to zero because it is not possible to use the aggregate function and select conditions to reduce the number of paths retrieved.

- $\text{est}(allPathsAggSel(N_1, P_1, N_2, P_2, \ldots, P_n, N_{n+1}, GroupList, Agg_Function, Sel_Condition))$
  $= \frac{MIN(est(N_1), est(N_{n+1})) \times Get_Constant}{1 + Agg_Constant + Sel_Constant}$

  All_Constant is a constant that estimates the number of paths in the retrieved set starting or ending at a node in N_1 or N_{n+1} respectively. Applying the aggregate function and the selection condition reduces the number of paths retrieved. The constants *Agg_Constant* and *Sel_Constant* approximate the percentage of paths that will not be retrieved because of the aggregate function and the select condition. The value of the constant *All_Constant* has been set to 1000 because the *allPaths* operator generally retrieves a large number of objects. The values of both the constants *Agg_Constant* and *Sel_Constant* will be set to 10 if the *Agg_Function* and the *Sel_Condition* are not *Null*, otherwise their values will be set to zero because it is not possible to use the aggregate function and select conditions to reduce the number of paths retrieved.

- $\text{est}(\text{G}.getArcs()) = \text{est}(G)$

  The number of objects expected to be in the result is expected to have the same order of magnitude as the number of graphs in the set G. This is because experience with the queries has lead us to believe that each graph will have a small number of arcs.

- $\text{est}(\text{P}.getArcs()) = \text{est}(P)$

  The number of arcs in the resulting set is expected the same order of magnitude as the number of paths in the set P. This is because experience with the queries leads us to believe that each path will have a very small number of arcs.

- $\text{est}(\leftarrow (P)) = \text{est}(P)$

  The $\leftarrow$ (reverse arrow) operator has the same number of objects as the argument.

Table 4.3: Estimation heuristics for Node operators

- est($P.source()$) = est(P)

  The number of objects in the result is expected to have the same order of magnitude as the number of paths in the set P.

- est($P.sink()$) = est(P)

  The number of objects in the result is expected to have the same order of magnitude as the number of paths in the set P.

- est($G.getNodes()$) = est(G)

  The number of objects in the result is expected to have the same order of magnitude as the number of graphs in the set G. This is because experience with the queries has led us to believe that each graph in the graph set G will have a small number of nodes.

- est($P.getNodes()$) = est(P)

  The number of objects in the result is expected to have the same order of magnitude as the number of paths in the set P. This is because experience with the queries has led us to believe that each path in the path set P will have a small number of nodes.

Table 4.4: Estimation heuristics for Graph operators

- est($getGraphs(P)$) = est(P)

  The number of objects in the result will be the same as the number of objects in the set P.

- est($getGraphs(N)$) = est(N)

  The number of objects in the result will be the same as the number of objects in the set N.

- est($gUnion(G_1, G_2)$) = est(G_1) x est(G_2)

  The $gUnion$ operator requires that the cartesian product of the graphs in the two set be found, and the expression in the RHS is the number of objects in the cartesian product.

- est($gIntersect(G_1, G_2)$) = est(G_1) x est(G_2)

  The $gIntersect$ operator requires that the cartesian product of the graphs in the two set be found, and the expression in the RHS is the number of objects in the cartesian product.

- est($gDiff(G_1, G_2)$) = est(G_1) x est(G_2)

  The $gDiff$ operator requires that the cartesian product of the graphs in the two set be found, and the expression in the RHS is the number of objects in the cartesian product.

and the paths that have been retrieved to save processing time and space when executing queries with the *getPaths*, *pCompose*, and the *allPaths* operators. We achieve this savings by trimming the search space as early as possible during the processing. This reduction reduces the time required to traverse the nodes and arcs as well as reduces the space required to store the paths. Similar techniques have been described in [DAJ91, CN89].

The selection condition can be used to make the following decision.

- Extend? When an intermediate path is found, decide if the path needs to be extended any further. If the selection condition is to find paths with *the sum of the weights stored on the paths less than 10*, and an intermediate path (linking a node in set N_1 to one in set N_j where $2 < j < n + 1$), that has been created has weight 20, and we know that the weights on an arc is always positive, it is not necessary to extend this intermediate path any further.

The aggregate function can be used to restrict the search space as follows.

- Extend? When a path is found, compare the path with other paths created so far to decide if it needs to be extended. For example, consider the query *find the shortest path connecting two given nodes*, and the intermediate path joining some two nodes is longer than the shortest path found between them so far, it is not necessary to extend the long path any further.

- FindMore? When a path is found, decide if any other paths starting from the same source / sink node need to be found. This will be useful to limit to only one path if the query requires that only the existence of a path from the source needs to be found.

To summarize, it is possible to use meta-information associated with the selection condition and the aggregate function to decide whether the intermediate path needs to be extended and whether any other paths starting from the same node need to be considered.

This information should be supplied by the database manager when defining functions associated with the conditions and aggregate functions for path objects (see Section 3.3.1). The functions associated with path objects will return additional arguments called *selExtendMore* (for the select functions), *aggExtendMore* and *aggFindMore* (for the aggregate functions). The query processor will use these values when processing the queries.

If the meta-information is not supplied, the query processor first evaluates all the possible paths and then applies the selection condition and/or the aggregate function.

The following are examples of selection conditions and aggregate functions that can be used to reduce the search space.

- Selection Conditions

  In the following list, **ComponentName** refers to the name of a component of the path. It can be the name of a part of the path such as the weight on an arc or the name of a member function defined for the path such as **length()**

  1. ComponentName == Value: Returns "**True**" if the value of the component of the given path is equal to the given value.
     - selExtendMore? Returns "**True**" if the value of the component of the given path is an integer, cannot be negative, and is less than the given value.
  2. ComponentName <= Value: Returns "**True**" if the value of the component of the given path is less than or equal to the given value.
     - selExtendMore? Returns "**True**" if the value of the component of the given path is an integer, cannot be negative, and is less than the given value.
  3. ComponentName >= Value: Returns "**True**" if the value of the component of the given path is greater than or equal to the given value.
     - selExtendMore? Always returns "**True**".

- Aggregate Conditions

  1. Exist(): Finds only the existence of a path.
     - aggExtendMore? Always returns "**True**"
     - aggFindMore? Always returns "**False**"
  2. Min(ComponentName): Stores only the paths that have the least value for the given component compared to the other paths in the grouplist.
     - aggExtendMore? Returns "**False**" if the value of the component of the given path is an integer, cannot be negative, and is greater than the least value found so far. Returns "**True**" otherwise.
     - aggFindMore? Always returns "**True**"

4.2.5 Data Representation

An important consideration when retrieving sets of paths is the space required to store the retrieved paths. This is important because the number of nodes and arcs that can be present in each path can be very large. Also, each node or arc can be a large structure (for example, a multimedia object).

To alleviate these problems, only the LEND IDs of the nodes and arcs of the paths that are created are stored during query processing.

The optimization techniques described in the earlier sections also help reduce the number of paths that are stored when the *aggregate* and the *select* operators are used. The *aggregate* operator when used will result in only one path being stored for paths having the same values given in *groupList*.

4.3 Summary

To summarize, the current implementation has focussed on optimizing the *getPaths*, *pCompose* and *allPaths* operators because they are frequently used and expensive operations. Queries containing these operators are optimized as follows:

- **Query transformation:** When either of the three operators are followed by the *aggregate* and/or the *select* operator, the operations are combined into one step. This is done so that the *select* and the *aggregate* operations can be applied on the paths as they are retrieved to try to omit useless paths and reduce the search space.

  Also, it is sometimes possible to use information about the selection condition, aggregate function and the paths retrieved so far, to decide if the new path that has been retrieved should be extended and other paths starting from the same node need to be considered.

- **Statistical Estimation:** The numbers of nodes and paths in the first and last operands of the three path creation operators are estimated. The queries are then

processed by traversing the paths starting from the node / path set that is expected to retrieve the least number of objects.

- **Use of Meta-information about the database:** It is possible to reduce the number of paths that are traversed by using information about the selection condition, aggregate function and the paths retrieved till the given time.

  As described in Section 4.2.4, the database manager can choose to supply meta-information for the selection conditions and aggregate functions associated with the paths when defining the database.

- **Data Representation:** The query processor stores the minimum information (the IDs of the nodes and edges) in order to reduce the amount of space required to store the paths and graphs.

4.4 Discussion

This chapter described the processing of GOAL queries. The architecture of the GOAL processor was first described. Optimization techniques used to process queries with the *getPaths*, *pCompose* and the *allPaths* operators were given.

We give below a list of conditions under which the optimizations cannot be applied or will fail to be effective and then a list of recommendations for future work to overcome these failings.

Conditions under which the optimizations are not effective: The optimization techniques will not be effective if any of the following conditions hold.

1. The optimization techniques do not reduce the size of the graph that needs to be traversed.

2. The information graph contains nodes with large in-degree and / or out-degree; as a result queries require the traversal of a large number of paths.

3. The GOAL query results in long paths being retrieved.

Recommendations for overcoming the ineffectiveness of the optimizations for all the conditions listed above:

1. The query processor can use heuristics to foresee the problems listed above by estimating the number of paths generated and ask the user if the query needs to be processed. The user can at this point decide to modify the GOAL statement so that it is executed in a reasonable amount of time.

   The number of paths retrieved by the *allPathsAggSel* operator can be estimated by using the method described by Lipton and Naughton in [LN89]. Similar techniques can be developed to estimate the number of paths generated by the *getPathsAggSel* and the *pComposeAggSel* operations.

2. The problems listed above can lead to the retrieval of a large number of paths or long paths. The problem can be further worsened if the objects on the paths lie on different pages. This will increase the I/O cost and can lead to thrashing. It is possible to alleviate such problems by using clustering techniques to store node and arc objects occurring in the same paths on the same page. Lavinus [Lav92] describes heuristics for clustering as applied to information graphs.

3. Another technique to solve the problems listed above is to evaluate partial results and display them to the users. This technique, also called lazy evaluation, is very useful for information retrieval applications because many times the users are interested in browsing and after seeing some results may decide that it is not worth pursuing the query any further, or may decide to modify the query.

Other optimization techniques that can be added: In this thesis we identified query expressions that would have served as bottlenecks to the effective use of the query language and we proposed optimization techniques to remove these bottlenecks. Further study of the

language and applications using the language will reveal more bottlenecks. Techniques to remove these bottlenecks can then be implemented.

The following is a list of techniques which may be useful.

- In Section 4.2 we described some useful query tree transformations. Other transformations can be added to improve performance. [JK84] is a survey of optimization techniques used in relational databases. Many techniques described there can be adapted and used for GOAL. [SG85, Hal74] also discuss this problem.

  One such technique is to optimize the execution of the *gUnion()*, *gIntersect()* and the *gDiff()* operations when they are followed by a *select()* operation. These can be combined using query transformation to perform the two operations simultaneously. The *gUnion()*, *gIntersect()* and the *gDiff()* operations are very expensive because they involve the creation of the cartesian product of the two graph sets and the resulting set has the potential of being very large.

- The heuristics described in this chapter to estimate the size of the retrieved set were very simple. Sophisticated heuristics if used will give better estimates and can lead to better optimizations. [LN89, Lyn88, Dem80] describe heuristics that can be adapted for GOAL operators.

- In our implementation we used the technique described in [Pur70] to traverse the graph when evaluating an *allPaths* query. Techniques and algorithms such as those described in [DJ92, DAJ91, AJ90, ADJ90, Ban85, Jia90, Lu87, IR88, UY90, IRW93] reduce the I/O cost of the computation. They need to be studied and adapted for GOAL.

- As we had noted earlier, the space required to store the sets of retrieved paths is large because there is no limit on the lengths of the paths. Optimization techniques to reduce this space need to be developed. [Jag90, DJ92] consider this problem for

the generalized transitive closure problem. These techniques need to be studied and adapted for GOAL.

Chapter 5

WordNet Database

This chapter illustrates the use of GOAL by describing how it supports queries against the WordNet database. Section 5.1 describes the structure of the WordNet database. Section 5.2 describes the structure of the WordNet information graph that will be used in the rest of the chapter. The expressive power of GOAL is illustrated in Section 5.3 by showing how some useful queries can be formulated using GOAL. The section also describes the processing of the queries using the optimization techniques described in Chapter 4. Sub-section 5.3.4 describes how the commands in the WordNet command language can be expressed by modifying the queries described earlier in the section. The last section summarizes the other sections in the chapter.

5.1 WordNet Database Structure

The discussion in this section is a synopsis of the reference [MBF+92]. WordNet contains a set of words, organized according to their meaning and their relationships to each other. This organization is based on psycho-linguistic principles. WordNet can be used as a dictionary or a thesaurus and also can be used to test psycho-linguistic theories.

A 'word' in WordNet has both a syntactic form determined by its spelling and a meaning associated with it. In order to avoid ambiguity, the syntactic form of the word is called the *word form* and the meaning is called the *word meaning* or *word sense*. A word form can have different meanings associated with it.

A word form in WordNet can be a single word or a sequence of words taken together to represent a concept (e.g., "fountain pen" and "take in"). Such words are called *collocations*.

WordNet stores collocations by replacing the space separating the words by the underscore character "_".

A set of word senses having the same meaning (i.e., a set of synonyms) are grouped together into a synonym set referred to as a *synset.*

Also, the words are categorized as nouns, verbs and adjectives. WordNet stores the relations between word senses. These relationships help define the meaning of the word. Table 5.1 summarizes the different relations between word senses that are stored in WordNet. Although the relationships listed in the table are defined for word senses, many relationships are valid between all the words in one synset and all the words in another synset. Such relationships can be considered as relationships between synsets instead of between individual words.

A verb in WordNet can be associated with a *frame.* A frame is a template illustrating the usage of the verb. For example, "Something – – –s" and "It is – – –ing". A verb is associated with a frame if the template with the verb replacing the blanks can be used as part of a sentence. For example, the verb `run` will be associated with the two frames given above because the phrases "`Something runs`" and "`It is running`" can be parts of a sentence.

If all the verbs in a synset are related to a frame, the relationship can be considered as a relationship between the verb synset and the frame instead of between the individual verbs and the frames.

5.2 WordNet Information Graph Structure

Figure 5.1 shows the structure of the information graph for the WordNet database that will be used in the following discussion. This structure was designed by the author. Other structures are possible also. This structure has been selected because it models well the relationships between the entities in the database. We describe below the different types of node and arc objects in the information graph, the class hierarchy of the objects and the

Table 5.1: WordNet relations adapted from [MBF+92]

- **Antonymy:** Word X is an antonym of word Y if the two words have opposite meanings. Antonymy is a symmetric relation, i.e., if word X is an antonym of word Y then word Y will be an antonym of word X. For example, "rich" is an antonym of "poor" and "poor" is an antonym of "rich".
- **Causal Relation:** Causal relations exist between verb synsets. Word X has a causal relationship between word Y if X causes Y. For example, the word "give" has a causal relationship with the word "have".
- **Entailment:** Entailment is a relation between verbs. Word X entails word Y if X cannot be done unless Y is, or has been done. For example, the words "hit" and "miss" entail "aim".
- **Hyponymy and Hypernymy:** Word X is a hyponym of word Y if X is a (kind of) Y. Hypernymy is the inverse relation of Hyponymy. These two relations exist only between noun and verb word senses. For example, "conifer" is a hyponym of "tree" and "tree" is a hypernym of "conifer".
- **Meronymy and Holonymy:** Meronymy is a part-whole relationship between nouns and holonymy is the inverse relation.
  - **Member Meronymy and Holonymy:** Word X is a member meronym of word Y if an X belongs to a Y. For example, a "tree" is a member meronym of "forest" because a tree is a part of a forest and belongs to a forest; the word "forest" is a member holonym of the word "tree".
  - **Part Meronymy and Holonymy:** Word X is a part meronym of word Y if X is a component part of Y. For example, a "branch" is a part meronym of "tree" and "tree" is a part holonym of "branch".
  - **Substance Meronymy and Holonymy:** Word X is a substance meronym of word Y if Y is made of X. For example, "aluminium" is a substance meronym of "airplane" and "airplane" is a substance holonym of "aluminium".
- **Pertains To:** Many adjectives are used in association with a noun. Relations between adjectives and the nouns that they are related to are stored as "pertains to" or pertainymy relations. For example, "astral" and "stellar" are pertainyms of the noun "star".
- **See Also:** A See Also relationship between word X and word Y implies that information about word X can be found from word Y. For example, the word "alert" has a See Also relationship to the word "awake". This relation exists between adjective synsets, verb synsets and verb word senses.
- **Similar To:** The similar to relation is present between adjectives. Some adjectives do not have direct antonyms but their antonyms can be derived by finding the antonyms of the synonyms of these words. For example, the word "moist" does not have a direct antonym. But it is similar to the word "wet" which has an antonym "dry". Thus, "dry" can be considered as an antonym of "wet".

information stored in the objects.

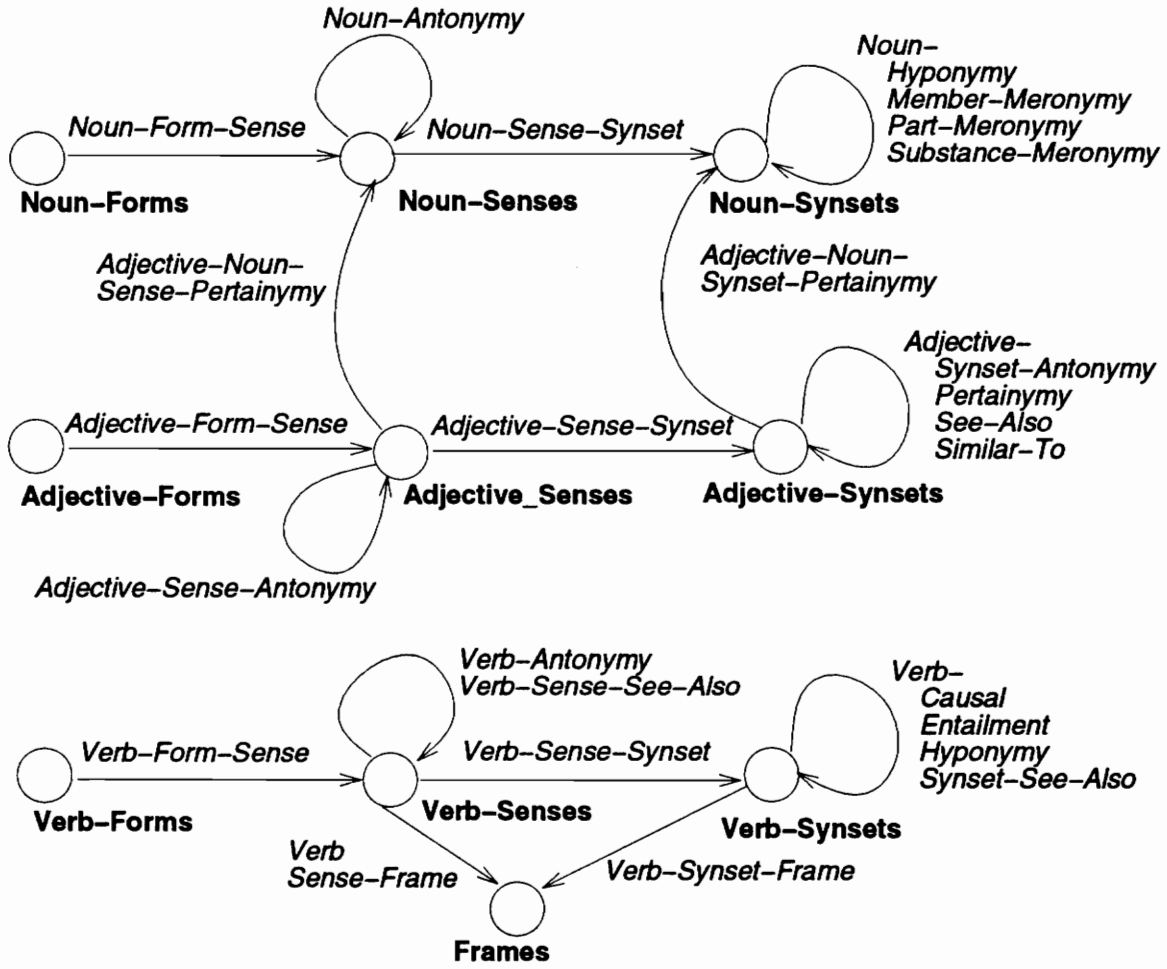

Figure 5.1: WordNet database structure

The figure shows three types of nodes – **Forms**, **Senses** and **Synsets** – corresponding to the word forms, word senses and synsets, respectively. These nodes can correspond to either nouns, adjectives or verbs. The nodes for each part of speech are derived from the basic node. For example, the nodes **Noun_Forms**, **Adjective_Forms** and **Verb_Forms** are derived from **Forms**. In addition, the WordNet information graph contains nodes of type **Frames** which are connected to the **Verb_Senses** and **Verb_Synsets** nodes. Figure 5.2 shows the hierarchy for the WordNet node objects.

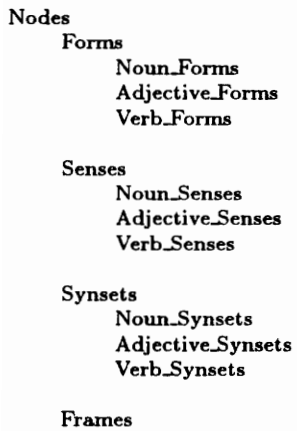

Figure 5.2: Class hierarchy of the WordNet node objects

The **Forms** nodes are connected to the **Senses** nodes with arcs of type **Form_Sense** if the word form associated with the **Forms** node corresponds to the word sense associated with the **Senses** node. The **Senses** nodes are connected to the **Synsets** nodes with arcs of type **Sense_Synset** if the word associated with the **Senses** node belongs to the synset associated with the **Synsets** node.

The relationships between different word senses and synsets for the nouns, adjectives and verbs as well as the relationships between the verb word senses and verb synsets and the frames that exist in the WordNet database are shown in Figure 5.1. Some relations (such as hyponymy and hypernymy, meronymy and holonymy, etc.) between the word senses and the synsets are reflexive. In such cases only one relation is stored since it is possible to derive the other relation. This reduces the space required to store the relations. Figure 5.3 shows the hierarchy of the WordNet arc objects.

The number of word senses of a word in the *Collins English Dictionary* is available as part of WordNet. This information is stored with the word form. It called the *polysemy count* of the word. It is not necessarily the same as the number of word senses of the word in WordNet. The **Forms** nodes also store the string corresponding to the word form. The **Senses** nodes store the sense number and the **Synsets** nodes store a short description of

- Edges
  - Relations
    - Synset_Synset
      - Noun_Synset_Synset
        - Noun_Hyponymy
        - Noun_Meronymy
          - Noun_Member_Meronymy
          - Noun_Part_Meronymy
          - Noun_Substance_Meronymy
      - Adjective_Synset_Synset
        - Adjective_Synset_Antonymy
        - Adjective_Pertainymy
        - Adjective_See_Also
        - Adjective_Similar_To
      - Verb_Synset_Synset
        - Verb_Causal
        - Verb_Entailment
        - Verb_Hyponymy
        - Verb_Synset_See_Also
    - Sense_Sense
      - Noun_Sense_Sense
        - Noun_Antonymy
      - Adjective_Sense_Sense
        - Adjective_Sense_Antonymy
      - Verb_Sense_Sense
        - Verb_Antonymy
        - Verb_Sense_See_Also
      - Adjective_Noun_Sense_Sense
        - Adjective_Noun_Sense_Pertainymy
      - Adjective_Noun_Synset_Synset
        - Adjective_Noun_Synset_Pertainymy
  - Verb_Sense_Frame
  - Verb_Synset_Frame

Figure 5.3: Class hierarchy of the WordNet arc objects

the words in the synset called a `gloss`. The `Frames` nodes store the template associated with it. The arcs in the WordNet database do not store any additional information.

Figure 5.4 shows an example from the WordNet database and Tables A.1 through A.6 in Appendix A provide various statistics about the database. The WordNet database contains a total of 251,024 nodes and 327,920 arcs.

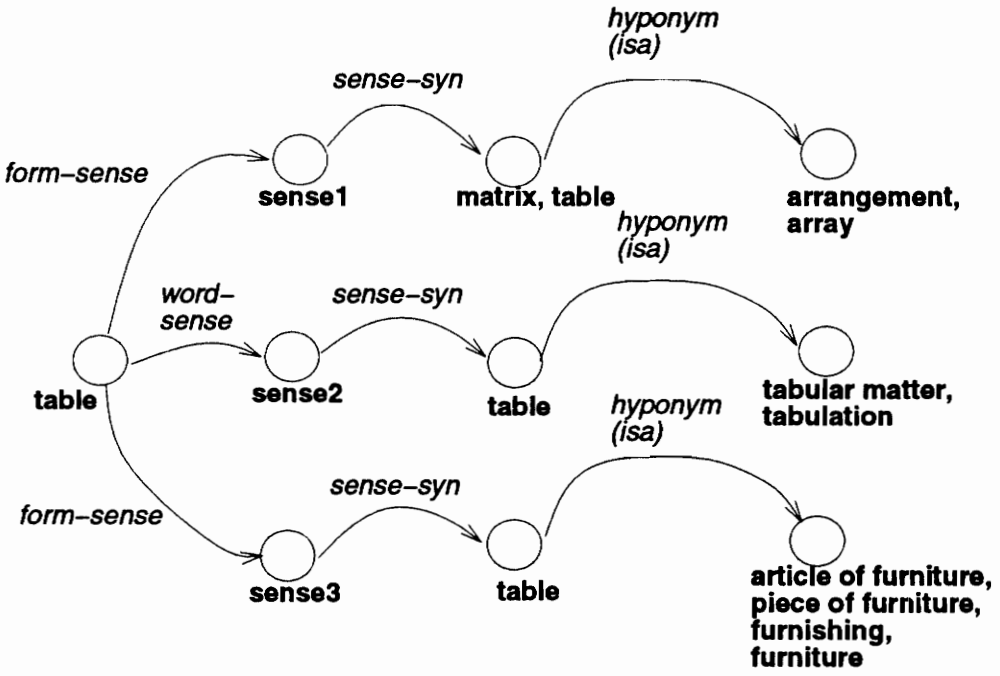

Figure 5.4: Part of WordNet database

5.3 WordNet Queries

This section first lists some queries for the WordNet database. The GOAL statements to express the queries and the optimization techniques that will be used when executing the queries are then described.

5.3.1 List of Queries

The following is the list of queries that will be considered in this section.

1. Find the synonyms of the noun "**table**".
2. Find all the types of information available for the noun "**table**". To process this query, we need to find the names of all the **Relations** arcs starting or ending at the **Synsets** or **Senses** nodes that are connected to the **Noun_Forms** nodes corresponding to word "**table**".
3. Find the immediate hyponyms of the noun "**table**".
4. Recursively find the hyponyms of the noun "**table**".
5. Find the hyponyms of the noun "**table**" that are at most two levels away from it.
6. Find the words corresponding to the synsets that are sister nodes of the word "**table**" with respect to the hyponymy relation. A sister node of a given node with respect to the hyponymy relation is connected with a hyponymy arc to the hypernym of the given node. (Recall that hypernymy is the inverse relation of hyponymy.)

The queries listed above have been selected by studying the command line interface to the WordNet system. We describe in Section 5.3.4 the WordNet command line interface and the modifications that have to be made to the GOAL statements corresponding to the queries listed above in order to use them in place of the WordNet commands.

5.3.2 GOAL Statements for Queries

This section describes the GOAL statements that can be used to retrieve information required by the queries listed above. The optimization techniques that will be applied to process the queries are also described.

Before giving the GOAL statements to express the given queries, we give two GOAL statements that create sets of paths. The sets of paths can be assigned to path variables as shown below and the variables can be used in the query statements instead of repeating the expressions when expressing the queries given below.

```
pathSet Table_Synset =
    getPaths(Noun_Forms.select(String == "table"), Form_Sense, Senses,
        Sense_Synset, Synsets);
pathSet Form_Synset =
    getPaths(Forms, Form_Sense, Senses, Sense_Synset, Synsets).
        aggregate([Sink],
        SetPrintString(Forms.String, SynsetsList));
```

The variable **Table_Synset** is assigned the set of paths that start from the noun forms node corresponding to the word "**table**" and end at the synset nodes connected to it.

The variable **Form_Synset** is assigned the set of paths formed by first finding all the paths that link the **Forms** nodes to the **Synsets** nodes, and then applying the *aggregate* operator **SetPrintString()** to the set of paths created, to form one path per synset node containing a part called **SynsetsList** containing a comma separated list of the word forms that make up the synset. This list of word forms is stored as a part of the path that is generated.

The two variables **Table_Synset** and **Form_Synset** are first defined and then used in the other GOAL statements. As described in the previous chapter, the set of paths assigned to the two variables are not evaluated when the query processor recognizes the statements, rather, the GOAL expression in the RHS of the assignment statements replaces the variable name when it is recognized.

The number of paths that are expected to be retrieved by the GOAL expression assigned to the variable **Form_Synset** is going to be very large because it is equivalent to all the paths that link a **Forms** node to a **Synsets** node. As will be evident from the description given below, all these paths will not be retrieved because of the optimization techniques that are used to process the queries.

We now proceed to list the GOAL statements to express the given queries. Following each GOAL statement we describe briefly the optimization techniques that will be used to

process the query.

1. Find the synonyms of the word "**table**".

```
pCompose(Table_Synset, ←(Form_Synset)).  Print(SynsetsList);
```

This GOAL statement is used to find the paths from the **Noun_Forms** node corresponding to the word "**table**" to the word forms of its synonyms. The list of word forms corresponding to each synset is then printed. It uses the variables **Table_Synset** and **Form_Synset** instead of the GOAL expressions assigned to the variables.

When executing this query statement, the variables **Table_Synset** and **Form_Synset** are replaced by the GOAL expressions they were assigned, and the query is transformed to the following expression.

```
pCompose(
    getPathsAggSel(Noun_Forms.select(String == "table"),
        Form_Sense, Senses, Sense_Synset, Synsets,
        Null, Null, Null);
    ← (getPathsAggSel(Forms, Form_Sense, Senses, Sense_Synset, Synsets,
            [Sink],
            SetPrintString(Forms.String, SynsetsList),
            Null))).
    Print(SynsetsList);
```

As described in Chapter 4, *Null* values are used in place of the *groupList*, *aggregateFunction* and the *selectCondition* when their corresponding operators are not specified.

When executing this transformed query, the query processor estimates the number of paths in the two sets of paths generated by the two operands of the *pCompose* operator. The estimated number of paths retrieved by the ← (reverse arrow) operator is equal

to the number of objects retrieved by the operand (see Table 4.2). Since both of the operands use the *getPathsAggSel* operator, to evaluate the estimated number of paths retrieved by both of the operands, the following formula for the *getPathsAggSel* operator will be used (see Table 4.2).

$$est(getPathsAggSel(N_1, P_1, N_2, P_2, \ldots, P_n, N_{n+1}, GroupList, Agg_Function, Sel_Condition)) = \frac{MIN(est(N_1), est(N_{n+1})) \times Get_Constant}{1 + Agg_Constant + Sel_Constant}$$

The values of the constants *Get_Constant*, *Agg_Constant* and *Sel_Constant* will be 100, 10 and zero respectively. The first operand uses a *select* operation to find the first set of nodes; therefore the value returned by the MIN function will be 1, while the value returned by the MIN function for the second operand will be the minimum of the number of objects in the **Forms** and the **Synsets** collections. The number of objects in both the collections is more than 1 (see Table A.1). As a result the *pCompose* operator will be executed by first finding the set of paths starting from the noun form "**table**" and then extending the paths using the *getPathsStarting()* member functions of the second operand. The *getPathsStarting()* member function retrieves only the paths starting from a given node that are retrieved by the path operator. As a result, instead of finding all the paths retrieved by the expression assigned to the variable *Form_Synset*, the query processor will find only those connected to the synsets of the given word form.

2. Find all the types of information available for the noun "**table**". First, it is necessary to find the different types of **Relations** arcs starting or ending at the **Senses** or **Synsets** nodes connected to the **Noun_Forms** node corresponding to the word "**table**".

```
pathSet Table_Sense =
    getPaths(Noun_Forms.select(String == "table"), Form_Sense, Senses);
```

```
pathSet Table_Synset =
    getPaths(Forms.select(String == "table"),
    Form_Sense, Senses, Synset_Synset, Synsets);
pathSet Table_SenseSynset = union(Table_Sense, Table_Synset);
pCompose(Table_SenseSynset, Relations).
        aggregate([Relations.ClassCode], Exist()).
        Print(Relations.ClassName);
```

The first three statements are used to find all the paths starting from the **Noun_Forms** node corresponding to the word "**table**" that are connected to the **Senses** and **Synsets** nodes.

The next statement is used to find the set of paths that link the paths assigned to the **Table_SenseSynset** variable to the **Senses** and **Synsets** nodes with arcs of type **Relations**. The *aggregate* operator is used to restrict the traversal to exactly one arc of each type of **Relations** arc.

The query statement given above is transformed to the query given below.

```
pComposeAggSel (
    union(getPathsAggSel (Forms.select(String == "table"),
            Form_Sense, Senses,
            Null, Null, Null),
        getPathsAggSel (Forms.select(String == "table"),
            Form_Sense, Senses, Sense_Synset, Synsets,
            Null, Null, Null)),
    Relations, [Relations.ClassCode], Exist(), Null).
    Print(Relations.ClassName);
```

This query is also processed starting from the **Noun_Forms** nodes corresponding to the word "**table**" because the first operand of the *pComposeAggSel* operator is expected to retrieve fewer paths than the second. The aggregate function **Exist()** is used to restrict the traversal to only one arc of each type of **Relations** arcs connected to the paths retrieved by the first *getPathsAggSel* operator. Optimization of the *pCompose* operator and the *aggregate* operator restricts the number of paths formed. Once one path of one type has been formed, the **Exist()** function will always return *No* to the question **findMore?**. This helps restrict the search. The names of all the **Relations** arcs in the path are printed.

3. Find the immediate hyponyms of the noun "**table**".

```
pCompose(Table_Synset, Noun_Hyponym, ←(Form_Synset)).
    Print(SynsetsList);
```

This GOAL statement is used to find the paths from the **Noun_Forms** nodes for the word "**table**", to the **Synsets** nodes it is connected to, the **Synsets** nodes of the hyponyms of the synsets and then the **Forms** nodes of the synsets. The list of hyponyms is then printed.

This query is transformed to the following query expression.

```
pCompose(
    getPathsAggSel(Noun_Forms.select(String == "table"),
        Form_Sense, Senses, Sense_Synset, Synsets,
        Null, Null, Null)),
    Noun_Hyponym,
    ← (getPathsAggSel(Forms, Form_Sense, Senses, Sense_Synset, Synsets,
            [Sink], SetPrintString(Forms.String, SynsetsList),
            NULL))).Print(SynsetsList);
```

Again, the query processor finds paths starting from the node corresponding to the word "table".

4. Recursively find the hyponyms of the words starting from the word "table".

```
pathSet All_Hyponyms = allPaths(Synsets, Hyponym, Synsets).
pCompose(Table_Synset, All_Hyponyms, ←(Form_Synset)).
    Print(Form_Synset.PrintString);
```

The statement given above is used to find the paths that link the **Noun_Forms** node corresponding to word "table" to the **Synsets** nodes that it is connected to. These paths are then extended to **Synsets** nodes that can be reached via **Hyponym** arcs are further extended to the word forms of the **Synsets**.

The query is converted to the following query expression.

```
pCompose(
    getPathsAggSel(Forms.select(String == "table"),
        Form_Sense, Senses, Sense_Synset, Synsets,
        Null, Null, Null)),
    Noun_Hyponym,
    allPaths(Synsets, Noun_Hyponym, Synsets),
    ← (getPathsAggSel(Forms, Form_Sense, Senses, Sense_Synset, Synsets,
            [Sink],
            SetPrintString(Forms.String, SynsetsList),
            NULL))).
    Print(SynsetsList);
```

This query is also processed starting from the Noun_Forms node corresponding to the noun "table". The *allPaths* operation used in this query is very expensive. The

query optimization helps in reducing the number of paths generated by finding only the paths that start from the **Synsets** nodes connected to the **Noun_Forms** nodes corresponding to the word "**table**".

5. Find the hyponyms of the noun "**table**" that are at most two levels away from it.

```
pathSet Two_Hyponyms = allPaths(Synsets, Hyponym, Synsets).
    select(length_LE("2"));
pCompose(Table_Synset, Two_Hyponyms, ←(Form_Synset)).
    Print(Form_Synset.PrintString);
```

The statement is used to find the paths that link the **Noun_Forms** node corresponding to word "**table**" to the **Synsets** nodes that it is connected to. These paths are then extended to **Synsets** nodes that can be reached via **Hyponym** arcs of length less than or equal to two. The resulting paths are further extended to the word forms of the **Synsets**.

The query is converted to the following query expression.

```
pCompose(
    getPathsAggSel(Forms.select(String ==  "table"),
        Form_Sense, Senses, Sense_Synset, Synsets,
        Null, Null, length_LE("2"))),
    Noun_Hyponym,
    allPaths(Synsets, Noun_Hyponym, Synsets),
    ← (getPathsAggSel(Forms, Form_Sense, Senses, Sense_Synset, Synsets,
            [Sink],
            SetPrintString(Forms.String, SynsetsList),
            NULL))).
    Print(SynsetsList);
```

The *allPaths* operator used in this query is a very expensive operator, but the selection condition `length_LE("2")` is used to restrict the number of paths that will be considered.

6. Find the co-ordinates of the word "`table`" with respect to the hyponymy relation. These are the words that belong to the synonym sets that are the hyponyms of the hypernyms of the word "`table`".

```
pCompose(Table_Synset, ←(Noun_Hyponym), Noun_Hyponym, ←(Form_Synset)).
    Print(SynsetsList);
```

For this query, first the paths to the hypernyms of the given word are found; recall that hypernymy is the inverse relation of hyponymy. The paths are then extended to the hyponyms of the hypernyms and then the corresponding `Forms` nodes.

This query expression is transformed as follows.

```
pCompose(
    getPathsAggSel(Noun_Forms.select(String == "table"),
        Form_Sense, Senses, Sense_Synset, Synsets,
        Null, Null, Null)),
    ← (Noun_Hyponym), (Noun_Hyponym),
    ← (getPathsAggSel(Forms, Form_Sense, Senses, Sense_Synset, Synsets,
            [Sink],
            SetPrintString(Forms.String, SynsetsList),
            NULL))).
    Print(SynsetsList);
```

This query is also evaluated starting from the first set of nodes.

5.3.3 Discussion

All the queries listed above employed query tree transformation to transform the queries into those having the *getPathsAggSel*, *pComposeAggSel* or *allPathsAggSel* operators. The number of nodes (or paths) was estimated for the first and the last operands retrieving sets of nodes (or paths). The estimated number of objects expected to be retrieved was used to decide the set that needed to be evaluated first. The paths were then generated by following arcs starting from the selected set until it was not possible or necessary to follow any more arcs. All queries were processed starting from the first set of nodes (or paths).

The processing of query 5 was optimized by using information about the selection condition to reduce the number of nodes that needed to be traversed, while that of query 2 was optimized by using information about the aggregate function to reduce the number of nodes that needed to be traversed.

The statistical estimation optimization technique functions by deciding whether to generate paths starting from the first or the last set of nodes (or paths). This is because we expect that if the paths are generated starting from the (better) selected set, fewer nodes will be traversed (see Section 4.2.3). Since there are only two options for selecting the set, if the optimization is not employed, the query processor will be forced to start generating paths starting from either one of the two sets. The query processor will traverse fewer nodes if the user specifies the query in the same order used by the query processor.

As stated earlier, all the GOAL queries that we have given require fewer node traversals if the paths are generated starting from the first set of nodes (or paths). Therefore, to test the effectiveness of the statistical estimation optimization technique, the query processor was forced to start generating paths starting from the last set of nodes (or paths).

Three types of test runs were conducted on a NeXT workstation using 20 randomly selected words instead of the noun "`table`". The three types are:

1. Using all the optimization techniques,

2. Not using the meta-information available for the selection condition and the aggregate

function but using the other optimizations.

3. Not using the statistical estimation optimization technique and forcing the query processor to start traversing paths starting from the last set of nodes (or paths).

The number of nodes traversed during the depth-first-search (see Section 4.2.1) of the information graph when processing a query was noted. The amount of time required to process a query is expected to be proportional to this number. We consider both, the number of nodes and time in the discussion below.

Tables 5.2 and 5.3 show the number of nodes traversed during run1 (*Opt*) and run2 (*Un-opt*) for queries 2 and 5 respectively. The number of paths retrieved (Size) for a query are also listed. This number is the same for both the runs. The following observations can be made from Tables 5.2 and 5.3.

- The tables show that the optimization helps reduce the search space for both queries. This is because, as described earlier, the query processor uses information about the aggregate function and the selection condition to reduce the search space when processing queries 2 and 5.

- The optimization gives better performance for query 5 than for query 2. This is because the selection condition (for query 5) is more effective than the aggregate function (for query 2) in trimming the search space for the queries and words used.

It was found that for queries 1, 3, 4 and 6, the first two runs required the same number of node traversals for all words. This is because for these queries the meta-information is not helpful in restricting the search space.

Tables 5.4, 5.5 and 5.6 show the time required to process queries 4, 5 and 6, respectively, during run1 (*Opt*) and run2 (*Un-opt*). The time is measured in 1/100ths of a second. The same query for each word was processed 10 times and the average time has been given.

The following observations can be made from Tables 5.4, 5.5 and 5.6.

- The time required to process a query is related to the number of paths retrieved (Size).

- As the number of paths retrieved by a query increases, the optimization is more effective (see for example, word **member** in Tables 5.4 and 5.5). This can be attributed to the fact that when optimizing, the query processor examines a path only once to see if it satisfies the selection condition and to aggregate the paths. Whereas, when the optimization is not used, the query processor first retrieves all the paths and then examines each path again to check if it satisfies the selection condition and to aggregate the set.

The time required to process queries 1, 2 and 3 did not differ much between the two runs and the difference between the time required for the optimized and unoptimized runs was less than 0.2 seconds for each word.

Run 3 (see page 88) required more than 2 minutes for each query. Runs 1 and 2 were all completed in less than one minute (except for 70 seconds in one case). Therefore we can conclude that the statistical estimation optimization technique is very effective.

To summarize, runs 1 and 2 showed that the meta-information is very useful for some queries, but may not be used or be less useful when processing other queries. Also, the optimization is more effective in reducing the time required to process a query when a large number of paths are retrieved.

5.3.4 WordNet Command Language

A WordNet command has the following form:

`wn searchstr [-d] [searchoption...]`

Here, `searchstr` is the word that should be searched for. Zero, one or more search options can be present and are applied in the given order.

- If no search option is specified, WordNet prints a list of valid options. This list depends on the types of links connected to the **Senses** and **Synsets** nodes connected to the **Forms** nodes corresponding to the given string. Query 2 given above finds these links. The valid options can be derived from the links by mapping the links to the options

Table 5.2: Number of nodes traversed when processing query 2

Word	Size	Opt. (# nodes trav.)	Un-opt. (# nodes trav.)	Diff (Un-opt - Opt) (# nodes trav.)	% Improvement (Diff/Opt.)
day	7	17	59	42	247
town	2	14	21	7	50
member	17	18	71	53	294
Thursday	1	7	8	1	14
force	1	20	56	36	180
polls	1	7	8	1	14
student	2	12	36	24	200
plaza	4	12	16	4	33
coordinator	1	7	8	1	14
revenue	1	12	15	3	25
aid	1	14	33	19	135
problem	1	11	19	8	72
volunteer	2	12	18	6	50
fan	2	14	21	7	50
Satan	1	7	8	1	14
contract	3	16	42	26	162
theater	3	9	26	17	188
item	1	9	13	4	44
dinner	1	11	16	5	45
debt	1	9	15	6	66

Table 5.3: Number of nodes traversed when processing query 5

Word	Size	Opt. (# nodes trav.)	Un-opt. (# nodes trav.)	Diff (Un-opt - Opt) (# nodes trav.)	% Improvement (Diff/Opt.)
day	43	241	308	67	27
town	3	21	21	0	0
member	133	602	1153	551	91
Thursday	0	4	4	0	0
force	60	323	370	47	14
polls	0	4	4	0	0
student	32	174	194	20	11
plaza	1	11	11	0	0
coordinator	0	4	4	0	0
revenue	1	11	11	0	0
aid	31	152	200	48	31
problem	5	36	36	0	0
volunteer	2	17	17	0	0
fan	4	26	27	1	3
Satan	0	4	4	0	0
contract	23	138	141	3	2
theater	4	32	32	0	0
item	2	20	20	0	0
dinner	2	18	18	0	0
debt	5	29	30	1	3

Table 5.4: Time required for processing query 4

Word	Size	Opt. (secs.)	Un-opt. (secs.)	Diff (Un-opt - Opt) (secs.)	% Improvement (Diff/Opt.)
day	99	13.00	13.40	0.40	3.1
town	3	0.60	0.60	0.00	0.0
member	595	68.80	69.80	1.00	1.5
Thursday	0	0.10	0.10	0.00	0.0
force	77	9.40	9.60	0.20	2.1
polls	0	0.10	0.10	0.00	0.0
student	41	4.80	5.00	0.20	4.2
plaza	1	0.40	0.40	0.00	0.0
coordinator	0	0.10	0.10	0.00	0.0
revenue	1	0.40	0.40	0.00	0.0
aid	64	7.70	7.80	0.10	1.3
problem	5	1.10	1.10	0.00	0.0
volunteer	2	0.40	0.40	0.00	0.0
fan	4	0.80	0.80	0.00	0.0
Satan	0	0.20	0.20	0.00	0.0
contract	23	3.30	3.50	0.20	6.1
theater	4	1.10	1.10	0.00	0.0
item	2	0.70	0.70	0.00	0.0
dinner	2	0.50	0.60	0.10	20.0
debt	5	0.70	0.80	0.10	14.3

Table 5.5: Time required for processing query 5

Word	Size	Opt. (secs.)	Un-opt. (secs.)	Diff (Un-opt - Opt) (secs.)	% Improvement (Diff/Opt.)
day	43	5.00	6.90	1.90	38.0
town	3	0.40	0.40	0.00	0.0
member	133	12.90	29.50	16.60	128.7
Thursday	0	0.10	0.10	0.00	0.0
force	60	6.90	8.30	1.40	20.3
polls	0	0.10	0.10	0.00	0.0
student	32	3.50	4.20	0.70	20.0
plaza	1	0.20	0.20	0.00	0.0
coordinator	0	0.10	0.10	0.00	0.0
revenue	1	0.20	0.20	0.00	0.0
aid	31	3.20	4.50	1.30	40.6
problem	5	0.70	0.80	0.10	14.3
volunteer	2	0.30	0.30	0.00	0.0
fan	4	0.50	0.50	0.00	0.0
Satan	0	0.10	0.10	0.00	0.0
contract	23	2.90	3.10	0.20	6.9
theater	4	0.70	0.70	0.00	0.0
item	2	0.50	0.50	0.00	0.0
dinner	2	0.40	0.40	0.00	0.0
debt	5	0.60	0.60	0.00	0.0

Table 5.6: Time required for processing query 6

Word	Size	Opt. (secs.)	Un-opt. (secs.)	Diff (Un-opt - Opt) (secs.)	% Improvement (Diff/Opt.)
day	95	8.00	8.20	0.20	2.5
town	19	1.90	2.00	0.10	5.3
member	31	3.90	4.10	0.20	5.1
Thursday	7	0.50	0.50	0.00	0.0
force	70	7.50	7.80	0.30	4.0
polls	5	0.50	0.50	0.00	0.0
student	19	2.30	2.40	0.10	4.4
plaza	25	3.10	3.40	0.30	9.7
coordinator	2	0.20	0.20	0.00	0.0
revenue	8	1.60	1.70	0.10	6.3
aid	30	3.80	4.00	0.20	5.3
problem	9	1.70	1.80	0.10	5.9
volunteer	31	3.70	3.90	0.20	5.4
fan	80	7.60	7.90	0.30	4.0
Satan	11	1.50	1.60	0.10	6.7
contract	41	4.20	4.30	0.10	2.4
theater	61	6.60	7.00	0.40	6.1
item	6	0.90	0.90	0.00	0.0
dinner	32	3.40	3.50	0.10	3.0
debt	4	0.50	0.50	0.00	0.0

which use the links. Hence query 2 can be considered to be equivalent to a WordNet command without search options.

- If the `-d` option is specified in the query, the **gloss** present in the **Synsets** nodes is printed together with the words present in the synset. The queries listed above can be reworded to require that the gloss also be included in the result and the GOAL statements can be easily modified to print the gloss.

- If the `-famln`, `-famla`, `-famlv` options are specified, the familiarity and polysemy information for the given word is printed. The familiarity is derived using the polysemy count for the word. Different ranges of polysemy count are assigned different degrees of familiarity with the highly polysemous words being highly familiar and words with low polysemy count having low familiarity.

  The following GOAL statement prints the polysemy information for the noun "**table**".

```
Noun_Forms.select(name == "table").  Print(polysemy);
```

- The following options can be expressed by suitably modifying query 3.

Option	Description
-hypon	immediate hyponyms of the given noun
-hypov	immediate hyponyms of the given verb
-subsn	substance meronyms of the given noun
-partn	part meronyms of the given noun
-membn	member meronyms of the given noun
-meron	all meronyms of the given noun
-sprtn	part holonyms of the given noun
-ssubn	substance holonyms of the given noun
-smemn	member holonyms of the given noun
-holon	all holonyms of the given noun
-entav	entailment relations of the given verb
-perta	pertainyms of the given verb
-antsn	antonyms of the given noun
-antsv	antonyms of the given verb

- The following options can be expressed by suitably modifying the GOAL query 4 given above.

Option	Description
-treen	recursive hyponym tree of the given noun
-treev	recursive hyponym tree of the given verb
-hypen	recursive hypernym tree of the given verb
-hypev	recursive hypernym tree of the given verb
-hholn	recursive holonym tree of the given noun

- The **-coorn** and the **-coorv** options display the co-ordinate or sister nodes of the given word with respect to the hyponym relation. They print the given word's immediate hypernym and the hypernym's immediate hyponyms. Query 6 given in the previous section finds this information.

5.4 Summary

This chapter described the structure of the WordNet database. Useful queries for the WordNet application were identified and expressed using the GOAL query language. The processing of the queries using the optimization techniques was also described.

It can be observed from the query statements that most of the queries use the *getPaths*, *pCompose*, *allPaths*, *select*, *aggregate* and *ClassName* operators. This shows that the operators that have been chosen for optimization are frequently used. The *union*, *intersect* and *diff* set operators are also useful but not as frequently used. The graph operators (*gUnion*, *gIntersect*, *gDiff* and the two *getGraphs* operators) and the *getArcs* and *getNodes* are not used very often.

The list of queries [Bet92] shows that the *source* and the *sink* operators are also used frequently for certain applications. Optimization techniques for these operators should be developed. The $\leftarrow$ operator also is used frequently, but the processing of this operator is closely tied to the processing of its operand.

Chapter 6

Conclusion

In this thesis we proposed the GOAL language (Chapter 3) as an extension to the LEND query language. GOAL can be used to retrieve IR (information retrieval) data stored in an information graph. In Chapter 3 we gave motivations for the query language constructs and examples to illustrate the use of the language.

The following extensions were made to the LEND query language.

1. **Addition of the graph operators:** The graph operators make it possible to use GOAL to describe unconnected graph patterns. For example, it is possible to formulate a query to find pairs of authors belonging to the same organization. This query could not be formulated using the LEND query language if arcs linking authors belonging to the same organization were not part of the information graph.

2. **Addition of the select operator for paths:** The select operator for paths allows us to retrieve paths that satisfy a condition. A function associated with the condition should be defined by the database manager for path objects of the given type (see Section 3.3.1).

3. **Addition of the aggregate operator for nodes and paths:**

   The aggregate operator allows aggregation of information present in a set of objects having similar characteristics. A function associated with the aggregate function should be defined by the database manager for the objects of the given type (see Section 3.3.1).

4. **Other extensions to make GOAL easier to use such as:**

- The generalization of the *getPaths* and the *pCompose* operations.
- The addition of the $\leftarrow$ (*reverseArrow*) operator.
- The generalization of the *select* condition.

These extensions have been described in Chapter 3.

The addition of the select and aggregate operator not only increases the expressive power of the language but also helps in the efficient execution of queries because, as described in Chapter 4, optimization methods can be used when processing the queries.

Chapter 4 described the optimization techniques that can be used to process the queries. The optimization techniques focussed on the optimization of the *getPaths*, *pCompose* and the *allPaths* operators when they were followed by the *select* and / or the *aggregate* operator. The optimization techniques used were: *Query Tree Transformations*, *Statistical Estimation*, *Use of Meta-Information* and *Data Representation*. In order for the query processor to use meta-information for the objects in the database it was necessary for the database manager to supply additional information when defining the database (see Section 4.2.4).

In Chapter 5 we used the WordNet system and a list of frequently used operations for that system to demonstrate the expressive power of the language and the effectiveness of the optimization techniques.

In Chapter 2 we described other database models and query languages and in Section 3.3 we compared them with the GOAL query model and language.

The following work can be carried out to further explore the area.

- GOAL can be used to retrieve subgraphs from the existing information graph. These retrieved subgraphs cannot be saved for later use. GOAL cannot be used to modify the objects in the database. An extension of GOAL for these functions would be very useful.
- In this thesis we explored some optimization techniques for efficient implementation of the queries. This area can be further explored to get better performance using the

techniques described in Section 4.3.

- It would be interesting to experiment using the query language with more databases. The results of the experiments can be used to find bottlenecks in the system. We expect the query language to be used in two modes: first, to query the database online and second, by embedding GOAL queries in a program that uses the database. The first approach supports ad-hoc queries for the database. Experiments using this approach will test the robustness of the system.

  If the second approach is used, we expect that GOAL could be a high level development language to access the database, deriving benefit from the optimization techniques used in GOAL. Such queries will require fast response time because they will be executed whenever the application is run. In that case, a compiler might be needed to supplement the current interpreter.

- GOAL being a command language may be difficult to use for novice users. An interesting project would be to build a graphical user interface on top of GOAL that would make it easy to use. The work with the GOOD model and GraphLog described in Section 2.3 are some other approaches of this type.

REFERENCES

[ABD+89] M. Atkinson, F. Bancilhon, D. Dewitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proceedings of International Conference on Deductive and Object Oriented Databases*, pages 223–240, Kyoto, Japan, Dec. 1989.

[ADJ90] Rakesh Agrawal, Shaul Dar, and H. V. Jagadish. Direct transitive closure algorithms: Design and performance evaluation. *ACM Transactions on Database Systems*, 15(3):427–458, 1990.

[AGP+92] Marc Andries, Marc Gemis, Jan Paredaens, Inge Thyssens, and Jan Van den Bussche. Concepts for graph-oriented object manipulation. In *Advances in Database Technology - Proceedings of the 3rd International conference on extending Database Technology*, pages 21–38, Vienna, Austria, March 1992.

[AJ90] Rakesh Agrawal and H. V. Jagadish. Hybrid transitive closure algorithms. In *Proceedings of the 16th International Conference on Very Large Databases*, pages 326–334, Brisbane, Australia, Aug. 1990.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[AU79] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pages 110–120, San-Antonio, Texas, Jan. 1979.

[Ban85] F. Bancilhon. Naive evaluation of recursively defined relations. In M. Brodie and J. Mysopoulos, editors, *On Knowledge Base Management Systems - Integrating Database and AI Systems.* Springer-Verlag, 1985.

[BCF+93] Dennis J. Brueni, Bailey T. Cross, Edward A. Fox, Lenwood S. Heath, Deborah Hix, Lucy T. Nowell, and William Wake. What if there were desktop access to the computer science literature? In Stan C. Kwasny, editor, *Proceedings of 21st Annual ACM Computer Science Conference: CSC '93*, pages 15–22, Indianapolis, IN, February 1993. ACM Press, New York, NY.

[Bet92] S. C. Betrabet. List of queries, 1992. Working paper, unpublished, Virginia Polytechnic Institute and State University.

REFERENCES

[BGAG92] Jan Van den Bussche, Dirk Van Gucht, Marc Andries, and Marc Gyssens. On the completeness of object-creating query languages. In *Proceedings of the 33rd Symposium on Foundations of Computer Science*, pages 372–379, Pittsburgh, PA, October 1992. IEEE Computer Society Press.

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[Cat91] R. G. G. Cattell. *Object Data Management: Object-oriented and Extended Relational Database Systems.* Addison-Wesley Publishing Co., Reading, MA, 1991.

[Cat93] R. G. G. Cattell. *The Object Database Standard: ODMG-93.* Morgan-Kaufmann Publishers, San Mateo, California, 1993.

[Che92] QiFan Chen. *Object-Oriented Database System for Efficient Information Retrieval Applications.* PhD dissertation, Computer Science Dept., Virginia Polytechnic Institute and State University, March 1992.

[CM90] Mariano Consens and Alberto O. Mendelzon. GraphLog: A visual formalism for real life recursion. In *Proceedings of the 9th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 404–416, 1990.

[CN89] I. F. Cruz and T. S. Norvell. Aggregative closure: An extension of transitive closure. In *Proceedings of the IEEE 5th International Conference on Data Engineering*, pages 384–391, Los Angeles, California, Feb. 1989.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[D+91] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10):34–48, October 1991.

[DAJ91] Shaul Dar, Rakesh Agrawal, and H. V. Jagadish. Optimization of generalized transitive closure queries. In *Proceedings of IEEE 7th International Conference on Data Engineering*, pages 345–354, 1991.

[Dat86a] C. J. Date. An introduction to the Unified Database Language UDL. In *Relational Database: Selected Writings*, pages 203–240. Addison-Wesley, Reading, MA, 1st edition, 1986.

[Dat86b] C. J. Date. Some principles of good language design (with especial reference to the design of database languages). In *Relational Database: Selected Writings*, pages 261–268. Addison-Wesley, Reading, MA, 1st edition, 1986.

[Dat87] C. J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley Publishing Co., Reading, MA, 4th edition, 1987.

REFERENCES

[DBT71] DBTG. Data Base Task Group of CODASYL programming language committee: Report. April 1971.

[Dem80] Robert Demolombe. Estimation of the number of tuples satisfying a query expressed in predicate calculus language. In *Proceedings of the 6th International Conference on Very Large Databases*, pages 55–63, Montreal, Oct 1980.

[DJ92] Shaul Dar and H. V. Jagadish. A spanning tree transitive closure algorithm. In *Proceedings of IEEE 8th International Conference on Data Engineering*, pages 2–11, 1992.

[DS92] Charles Donnelly and Richard Stallman. *The Bison reference manual: The YACC-compatible parser generator Ver. 1.20.* Free Software Foundation, 675 Massachusetts Avenue, Cambridge, MA 02139 USA, Dec. 1992.

[EN89] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings Publishing Co. Inc., Redwood City, California, 1st edition, 1989.

[FCDH90] Edward A. Fox, QiFan Chen, Amjad M. Daoud, and Lenwood S. Heath. Order preserving minimal perfect hash functions and information retrieval. In Jean-Luc Vidick, editor, *Proceedings of the 13th International Conference on Research and Development and Development in Information Retrieval*, pages 279–311, Brussels, Belgium, Sept. 1990.

[FCDH91] Edward A. Fox, QiFan Chen, Amjad M. Daoud, and Lenwood S. Heath. Order preserving minimal perfect hash functions and information retrieval. *Communications of the ACM*, 34(7):281–308, July 1991.

[FCF91] Edward A. Fox, QiFan Chen, and Robert K. France. A general reference model for hypermedia and information retrieval and its implementation in CODER / LEND. In Emily Berk and Joseph Devlin, editors, *Hypertext / Hypermedia Handbook*, pages 329–355. McGraw-Hill, Inc., New York, 1991.

[FCHD89] Edward A. Fox, QiFan Chen, Lenwood S. Heath, and S. Datta. A more cost effective algorithm for finding perfect hash functions. In *Proceedings of the 1989 ACM 17th Annual Computer Science Conference*, pages 114–122, Louisville, Kentucky, Feb. 1989.

[FCHD90] Edward A. Fox, QiFan Chen, Lenwood S. Heath, and Amjad M. Daoud. A minimal perfect hash functions finding algorithm for large databases. Technical Report TR-90-41, Department of Computer Science, Virginia Polytechnic Institute & State University, Blacksburg, Virginia, 1990.

REFERENCES

[FCHD92] Edward A. Fox, QiFan Chen, Lenwood S. Heath, and Amjad M. Daoud. Practical perfect hash function finding algorithms for large databases. *Communications of the ACM*, 35:606–611, Jan. 1992.

[FFS+93] Edward A. Fox, Robert K. France, Eskinder Sahle, Amjad M. Daoud, and B. E. Cline. Development of a modern OPAC: from REVTOLC to MARIAN. In *Proceedings of the ACM SIGIR International Conference on Information Retrieval*, pages 248–259, Pittsburgh, PA, June 1993.

[FHH91] Edward A. Fox, Lenwood S. Heath, and Deborah Hix. A User Centered Database for the Computer Science Literature. Technical report, Virginia Polytechnic Institute and State University, Blacksburg, 1991.

[FHN+93] Edward A. Fox, Deborah Hex, Lucy T. Nowell, Dennis J. Brueni, William C. Wake, and Lenwood S. Heath. Users, user interfaces, and objects: Envision, A digital library. *Journal of the American Society for Information Science*, 44(5), Sept. 1993.

[FL93] Edward A. Fox and L. Lunin. Introduction and overview to perspectives on digital libraries. *Journal of the American Society for Information Science*, 44(5), Sept. 1993.

[Fox81] Edward A. Fox. Implementing SMART for minicomputers via relational processing with abstract data types. In *Joint Proceedings of SIGSMALL Symposium on Small Systems and SIGMOD Workshop on Small Database Systems*, Orlando, Florida, 1981.

[Fox92] Edward A. Fox. Building a user-centered database from the ACM literature. In *Proc. Symposium on Document Analysis and Information Retrieval*, pages 235–246, Tropicana Hotel, Las Vegas, Nevada, March 1992.

[GPBG] Marc Gyssens, Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. A graph-oriented object database model. *to appear, IEEE Transactions on Knowledge and Data Engineering.* Revised version of Technical Report no 327, Computer Science Department, Indiana University, and of UIA Technical Report 91-27.

[Gra93] Goetz Graefe. Query optimization in database systems. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Hal74] P. A. V. Hall. Optimization of single expressions in a relational data base system. *IBM Journal of Research and Development*, 20(3):244–257, 1974.

[IR88] Y. E. Ioannidis and R. Ramakrishnan. An efficient transitive closure algorithm. In *Proceedings of the 14th International Conference on Very Large Databases*, pages 403–411, Los Angeles, California, Aug.–Sept. 1988.

REFERENCES

[IRW93] Yannis Ioannidis, Raghu Ramakrishnan, and Linda Winger. Transitive closure algorithms based on graph traversal. *ACM Transactions on Database Systems*, 18(3):512–576, Sept. 1993.

[Jag90] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Transactions on Database Systems*, 15(4), Dec. 1990.

[Jia90] B. Jiang. A suitable algorithm for computing partial transitive closures in databases. In *Proceedings of the IEEE 6th International Conference on Data Engineering*, pages 264–271, Los Angeles, California, Feb. 1990.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

[K+89] W. Kim et al. Features of the ORION object-oriented database system. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Applications, and Databases*. Addison-Wesley, Reading, MA, 1989.

[Kim90] W. Kim. Object-oriented databases: Definition and research directions. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):327–341, Sept. 1990.

[Klu82] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of ACM*, 29(3):699–717, July 1982.

[Lav92] Joseph W. Lavinus. Heuristics for laying out information graphs. Master's thesis, Computer Science Dept., Virginia Polytechnic Institute and State University, August 1992.

[Lip92] Stanley B. Lippman. *C++ Primer, 2nd edition.* Addison-Wesley Publishing Co., April 1992.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[LLPS91] Guy M. Lohman, Bruce Lindsay, Hamid Pirahesh, and K. Bernhard Schiefer. Extensions to Starburst: Objects, Types, Functions and Rules. *Communications of the ACM*, 34(10):94–109, October 1991.

[LN89] Richard J. Lipton and Jeffrey F. Naughton. Estimating the size of generalized transitive closures. In *Proceedings of the 15th International Conference on Very Large Databases*, pages 165–171, Amsterdam, Holland, 1989.

[Lu87] H. Lu. New strategies for computing the transitive closure of a database relation. In *Proceedings of the 13th International Conference on Very Large Databases*, Brighton, England, Sept. 1987.

REFERENCES

[Lyn87] Clifford Lynch. *Extending Relational Database Management Systems for Information Retrieval Applications.* PhD thesis, Computer Science Dept., University of California, Berkeley, Nov. 1987.

[Lyn88] Clifford A. Lynch. Selectivity estimation and query optimization in large databases with highly skewed distributions of column values. In *Proceedings of the 14th International Conference on Very Large Databases*, pages 240–251, Los Angeles, California, Aug.–Sept. 1988.

[Mac87] Bruce J. MacLennan. *Principles of Programming Languages: Design, Evaluation and Implementation, Second Edition.* Holt, Rinehart and Winston, New York, 2nd edition, 1987.

[MBF+92] George A. Miller, Richard Beckwith, Christiane Felibaum, Derek Gross, and Katherine Miller. Introduction to WordNet: An on-line lexical database. 1992.

[McG77] W. C. McGee. The IMS/VS system. *IBM Systems Journal*, 16(2), June 1977.

[ME92] Priti Mishra and Margaret H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[Pax90] Vern Paxson. *FLEX - fast lexical analyzer generator*, May 1990.

[PBA+92] Jan Paredaens, Jan Van den Bussche, Marc Andries, Marc Gemis, Marc Gyssens, Inge Thyssens, Dirk Van Gucht, Vijay Sarathy, and Lawrence Saxton. An overview of GOOD. *SIGMOD RECORD*, 21(1):25–31, March 1992.

[Pur70] P. Purdom. A transitive closure algorithm. *BIT*, 10:76–94, 1970.

[Ric83] Elaine Rich. *Artificial Intelligence.* McGraw-Hill Book Company, New York, 1983.

[Seb92] Robert W. Sebesta. *Concepts of programming languages.* The Benjamin / Cummings Publishing Co. Inc., Redwood City, CA, 2nd edition, 1992.

[SG85] David E. Smith and Michael R. Genesereth. Ordering conjunctive queries. *Artificial Intelligence*, 26:171–215, 1985.

[SK91a] Michael Stonebraker and Greg Kemnitz. The POSTGRES next-generation database management system. *Communications of the ACM*, 34(10):78–92, October 1991.

[SK91b] Michael Stonebraker and Greg Kemnitz. The POSTGRES next-generation database management system. In *Communications of the ACM*, volume 34, pages 78–92. October 1991.

[SQL] SQL. Database language SQL, document ANSI X3-135-1986 committee. Also, ISO/TC97/SC21/WG3 N143.

REFERENCES

[SSG91] Vijay M. Sarathy, L. V. Saxton, and D. V. Gucht. Translating query graphs into Tarski algebra expressions. Technical Report 342, Indiana University, Bloomington, Indiana, Dec. 1991.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[UY90] J. D. Ullman and M. Yannakakis. On the input/output complexity of transitive closure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, NJ, May 1990.

[X3H84] X3H2. American national standards database committee: Draft proposed Network Database Language NDL. Document X3H2-84-100. August 1984.

Appendix A

Statistics for the WordNet Database

Table A.1: Number of Nodes in the WordNet Database

Number of Noun Word Forms	57586
Number of Adjective Word Forms	14873
Number of Verb Word Forms	11340
Total Number of Word Forms	83799
Number of Noun Word Senses	63384
Number of Adjective Word Senses	15922
Number of Verb Word Senses	28956
Total Number of Word Senses	108262
Number of Noun Synsets	41263
Number of Adjective Synsets	10741
Number of Verb Synsets	6925
Total Number of Synsets	58929
Number of Verb Frames	34

Table A.2: Number of Relations between **Forms**, **Senses**, **Synsets** and **Frames** nodes in the WordNet Database

Number of Noun Word Form to Noun Word Sense Links	71575
Number of Noun Word Sense to Noun Synset Links	71575
Number of Adjective Word Form to Adjective Word Sense Links	21009
Number of Adjective Word Sense to Adjective Synset Links	21009
Number of Verb Word Form to Verb Word Sense Links	18285
Number of Verb Word Sense to Verb Synset Links	18285
Number of Verb Synsets to Verb Frames Links	11547
Number of Verb Word Senses to Verb Frames Links	11678

Table A.3: Number of Relations between Noun nodes in the WordNet Database

Relations Between Noun Synsets	
Number of Hyponymy and Hypernymy relations	41706
Number of Member Meronymy and Holonymy relations	6144
Number of Part Meronymy and Holonymy relations	4022
Number of Substance Meronymy and Holonymy relations	112
Relations Between Noun Words	
Number of Antonymy relations	1272

Table A.4: Number of Relations between Adjective nodes in the WordNet Database

Relations Between Adjective Synsets	
Number of Antonym relations	5
Number of Pertains relations	98
Number of See Also relations	2913
Number of Similar To relations	14215
Relations Between Adjective Words	
Number of Antonym relations	2403

Table A.5: Number of Relations Between Verb nodes in the WordNet Database

Relations Between Verb Synsets	
Number of Cause relations	159
Number of Entailment relations	402
Number of Hyponymy relations	6522
Number of See Also relations	2
Relations Between Verb words	
Number of Antonymy relations	850
Number of See Also relations	855

Table A.6: Number of Relations between Adjective and Noun nodes in the WordNet Database

Relations Between Adjective Synsets and Noun Synsets	
Number of Pertains To relations	1296
Relations Between Adjective Words and Noun Words	
Number of Pertains To relations	1

VITA

Sangita Betrabet (nee Sangita Sirur), daughter of Krishnanand and Nalini Sirur, was born on 21st August 1966 in Bombay, India. She graduated from St. Columba High School in 1981. She then studied at Wilson College, Bombay and obtained a Bachelor's degree in Mathematics from Bombay University in 1986. She obtained a diploma in Computer Science from S.N.D.T. University, Bombay in 1987.

From October 1987 to December 1988 she worked at Johnson & Johnson, Bombay as a Programmer. In January 1989, she married Chinmay S. Betrabet. She enrolled in the Master's program at Virginia Tech in Fall 1990.

Sangita C Betrabet