INTELLIGENT QUERY ANSWERING BY KNOWLEDGE DISCOVERY TECHNIQUES

by

Yue Huang

B.S., Tsinghua University, Beijing, China, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in the School
of
Computing Science

SIMON FRASER UNIVERSITY
August 1993

APPROVAL

Name:	Yue Huang
Degree:	Master of Science
Title of thesis:	**Intelligent Query Answering By Knowledge Discovery Techniques**

Examining Committee: Dr. Louis Hafer , Chairman

Dr. Jiawei Han,
Senior Supervisor

Dr. Nick Cercone,
Supervisor

Dr. Fred Popowich,
External Examiner

Date Approved: August 6, 1993

Title of Thesis/Project/Extended Essay

Intelligent Query Answering by Knowledge Discovery Techniques

Author: [illegible]
(signature)

Yue Huang
(name)

August 11, 1993
(date)

ABSTRACT

Knowledge discovery in databases facilitates querying database knowledge, cooperative query answering and semantic query optimization in database systems. In this thesis, we investigate the application of discovered knowledge, concept hierarchies, and knowledge discovery tools for intelligent query answering in database systems. A knowledge-rich data model is constructed to incorporate discovered knowledge and knowledge discovery tools. Queries are classified into data queries and knowledge queries. Both types of queries can be answered directly by simple retrieval or intelligently by analyzing the intent of query and providing generalized, neighborhood or associated information using stored or discovered knowledge. Techniques have been developed for intelligent query answering using discovered knowledge and/or knowledge discovery tools, which includes generalization, data summarization, concept clustering, rule discovery, query rewriting, lazy evaluation, semantic query optimization, etc. Our study shows that knowledge discovery substantially broadens the spectrum of intelligent query answering and may have deep implications on query processing in data- and knowledge-base systems. A prototyped experimental database learning system, DBLEARN, has been constructed. Our experimental results on direct answering of data and knowledge queries are successful with satisfactory performance.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude and appreciation to my senior supervisor, Dr. Jiawei Han for his valuable guidance, support and encouragement since I came to SFU. I should also be thankful to the members of my examining committee, Dr. Nick Cercone, Dr. Fred Popowich, and Dr. Lou J. Hafer for reading this thesis carefully and making thoughtful suggestions.

I would also like to thank all my friends in the School of Computing Science, Engineering Science who certainly contributed to the work I have done in SFU.

I owe sincere thanks to my parents and my sister for their unfailling support. I would like to dedicate to them whatever I have achieved in this research.

I am grateful to Simon Fraser University and the Natural Science and Engineering Research Council of Canada for the financial support.

Finally, my special thanks go to a special girl for her love, understanding, and support throughout the entire effort.

CONTENTS

CHAPTER 1

INTRODUCTION

Knowledge discovery in databases creates a new frontier for intelligent query answering and query optimization in database systems. It has been estimated that the amount of information collected by human beings in the world doubles every 20 months. The size and number of databases probably increases even faster. The growth in the size and number of existing databases far exceeds human abilities to analyze such data, thus creating both a need and an opportunity for extracting knowledge from databases.

William Frawley and his colleagues[18] give a definition of knowledge as follows.

"Given a set of facts(data) F, a language L, and some measure of certainty C, a *pattern* is defined as a statement S in L that describes relationships among a subset F_s of F with a certainty c, such that S is simpler (in some sense) than the enumeration of all facts in F_s. A pattern that is interesting (according to a user-imposed interest measure) and certain enough (again according to the user's criteria) is called *knowledge*."

Although these definitions about the language, the certainty, and the simplicity and interestingness measure are intentionally vague to cover a wide variety of approaches. Collectively, these terms capture our view of the fundamental characteristics of discovery in databases.

The computer science community is responding to both the scientific and practical challenges presented by the need to find the knowledge adrift in the flood of data. Some research methods are already well enough developed to have been made part of commercially available software. Several expert system shells use variations of ID3[46] for inducing rules from examples. Other systems use inductive[41] or genetic learning approaches to discover patterns in personal computer databases[17]. A number of discovery algorithms have been developed. Conceptual clustering works with nominal and structured data and determines clusters both by attribute similarity and by conceptual cohesiveness, as defined by background information. Recent examples of this approach include AutoClass[8], the Bayesian Categorizer[3], and Cobweb[16]. Once identified, useful pattern classes usually need to be described rather than simply enumerated. Empirical learning algorithms, the most common approach to this problem, work by identifying commonalities or differences among class members. Well-known examples of this approach include decision tree inducers[50], neural networks[52], and attribute-oriented induction methods[24][25].

With the rapid development of knowledge discovery techniques, it is natural to study the applications of the technology in querying database knowledge and processing queries in database systems[20][9]. At Simon Fraser University, a prototyped experimental database learning system, DBLEARN[5][26], has been constructed. The system, DBLEARN, takes learning requests as inputs, applies the knowledge discovery algorithm(s) on the data stored in a database with the assistance of the concept hierarchy information stored in a concept hierarchy base. The outputs of the system are knowledge rules extracted from the database.

In this thesis, we investigate the application of discovered rules, concept hierarchies and knowledge discovery techniques to intelligent query answering in database systems. To study query processing in a knowledge-rich database associated with knowledge discovery tools, it is often necessary to distinguish data, knowledge and queries defined at the primitive data level from those defined at a relatively high concept level. Data in a knowledge-rich database are classified into primitive data and high-level data. The former are actual data stored in data relations and, if appearing in some concept hierarchies, correspond to the primitive level (i.e., leaf) nodes of the hierarchies; whereas the latter are nonprimitive data subsuming primitive ones and residing at the nonprimitive level of concept hierarchies. Correspondingly, a primitive-level query is a query whose constants involve only primitive data; whereas a high-level query is a query whose constants involve high-level data. Similarly, rules (notice that integrity constraints can be viewed as a special kind of rules) can be classified into primitive-level and high-level rules, based on their reference to high-level data.

In many cases, a database user may not be able to distinguish between primitive and high-level data and between information that is data and information that is knowledge. A knowledge query can often be viewed as a follow-up to a data query when the answer to a data query requires further explanation, reasoning or summarization. Therefore, it is important to provide a single, coherent framework to handle data and knowledge queries and to handle direct query answering and intelligent query answering. There have been some interesting studies on querying database knowledge and intelligent query answering [45][54][49][34][11]. Previous studies emphasize the application or inquiry of deduction rules and integrity constraints in relational or deductive databases. With the availability of generalized knowledge and knowledge discovery tools, queries can be posed and answered at levels higher than that of primitive concepts, and knowledge about general characteristics of data can be inquired or utilized in the processing of data or knowledge queries[27].

In this thesis, A knowledge-rich data model is constructed which consists not only of the components from a deductive database (including database schemas expressed by an extended deductive entity-relationship data model, data relations, deduction rules and integrity constraints) but also the components relevant to knowledge discovery processes, including concept hierarchies, generalized knowledge, and knowledge discovery tools. The knowledge discovery tools are used to extract general knowledge dynamically, when necessary, from any set of interested data in the database. A unified framework is established for answering data and knowledge queries in a knowledge-rich database. A systematic study is performed on intelligent query answering of both data or knowledge queries in a database system associated with discovered knowledge and knowledge discovery tools.

This thesis is organized as follows. The next chapter will give an overview of both the methods developed for knowledge discovery in large databases and the methods for intelligent query answering in database systems. Chapter 3 introduces the principles and implementations of DBLEARN. Chapter 4 presents a data model for knowledge-rich databases. Chapter 5 examines four basic categories of query answering in knowledge-rich databases based on the combinations of data vs.knowledge queries and direct vs.intelligent query answering mechanisms. Chapter 6 presents the conclusion as well as discussion of future research issues. The tutorial and source codes of DBLEARN are given in appendix.

CHAPTER 2

OVERVIEW: KNOWLEDGE DISCOVERY AND INTELLIGENT QUERY ANSWERING IN LARGE DATABASES

We survey some recent progress in two research frontiers: (1) knowledge discovery in database systems which adopt the *learning from examples* philosophy , and (2) some intelligent database query answering techniques.

2.1 Knowledge Discovery in Large Databases

Knowledge discovery is the nontrivial extraction of implicit, previously unknown, and potentially useful information from data[18]. In machine learning, discovery is often equated with unsupervised learning, learning from data with little or no guidance from a teacher. A *discovery system* is then a program that automatically finds relationships in data that it previously did not know about.

Several such discovery systems have been successful in scientific domains, e.g., AM[40], GLAUBER[39], ABACUS[13], COPER[37], and FortyNiner[57]. These systems traditionally have been applied to scientific data known to contain strong regularities(e.g., Ohm's law, Kepler's law). On such problems, these systems readily find functions that represent or approximate the known laws. The systems are said to "discover" these laws because, though known to us, they were unknown to the systems, i.e., the laws were outside the systems' representation of the world.

Another class of systems applies discovery methods to real-world databases. Such systems have come to be known as KDD(knowledge discovery in databases) systems, e.g. CoverStory[53], EXPLORA[31] and DBLEARN[26]. The differences between scientific discovery systems and KDD systems primarily result from the different characteristics of the data they are typicaly applied to. KDD systems, operating on real databases, have to deal with difficult issues, such as finding tendencies[29] rather that laws, data that is constantly changing and often erroneous, critical data that is missing because it was not designed into the databases, and an overwhelming quantity of data. Scientific discovery systems have had the luxury of ignoring some or all of these issuse. But they too will have to confront these issues if they are to be effectively applied to the rapidly growing scientific databases storing vast amounts of information.

Regardless of the source of the data, the value of automated discovery in the future will be in finding truly novel and interesting patterns in large, unexplored databases, and also in providing plausible explanations for these discoveries. Many difficult problems remain to be solved before a truly useful, autonomous discovery system will become possible.

2.1.1 Definition of a KDD System

A KDD system is defined as follows[48]:

"A KDD system is an automated system for efficiently identifying and extracting interesting patterns from data stored in real-world databases."

The important aspects of this definition are: (1) the system has some autonomy, (2) it has efficient methods for extracting patterns, (3) it can identify when a pattern is interesting, and (4) it interfaces to a DBMS(database management system).

Inherent in the meaning of discovery is autonomy - if a system is told exactly where and how to look for patterns it is not performing discovery(though it may be doing some form of supervised learning). Total autonomy is not required, but the system must be automated to the extent that it makes some of its own decisions about where to search for patterns, and can decide when a pattern is likely to be of interest.

Clearly a discovery system must have methods for identifying and extracting patterns from the data. These in fact form the core of any discovery system. The term ***pattern*** refers to any relation among elements of a database, i.e. the records, attributes, and values. Databases are replete with patterns, but few of them are of much interest. A pattern is interesting to the degree that it is accurate, novel, and useful with respect to the end-user's knowledge and objectives[18]. Lastly, to be usable on large active databases, a KDD system requires direct access to a DBMS. This at the very least implies the ability to send queries and process the results.

2.1.2 A Model KDD System

Piatetsky-Shapiro and Matheus[48] proposed the following model of an idealized system shown in Figure 2.1.

The model comprised five main components:

- **Controller:** decides how to apply the focusing, pattern extraction, and pattern evaluation to the relevant parts of the DBMS under the constraints provided by the domain knowledge and user input
- **Database Interface:** accepts queries from the controller and returns the results for use by the extraction methods
- **Focus Component:** controls which portions of the database are to be analyzed
- **Extraction Methods:** are the algorithms used to extract potentially interesting patterns

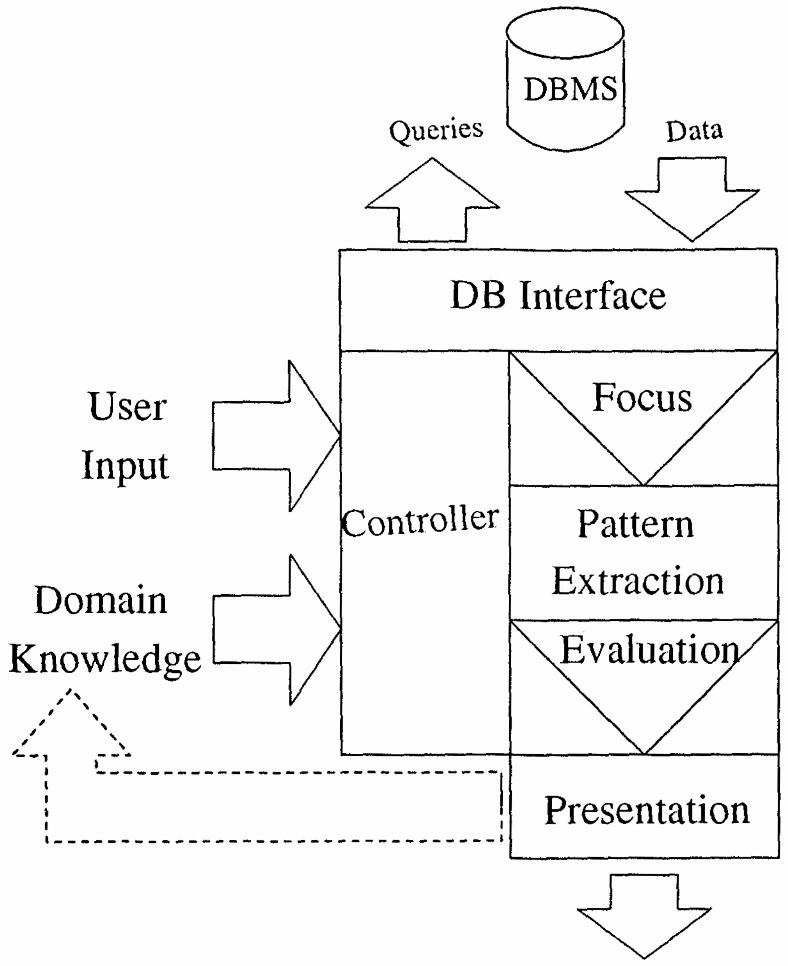

Figure 2.1: A Model KDD System

- **Evaluation Component:** screens the results of the extraction methods to ensure relevance to the current task as defined by the user input and domain knowledge

Information comes into the system from the user input, domain knowledge, and DBMS query results. The knowledge that is discovered is presented to the user and possibly added to the knowledge base for subsequent analysis.

In the above KDD model the system's autonomy comes from the controller. The basis for its decision making comes from the domain knowledge and user input. The controller interprets this input and uses it to direct the focus, extraction, and evaluation components. In practise, many KDD systems requires the end user to make the majority of these decisions[48][26].

Domain knowledge can assume many forms including(but not limited to):

- lists of relevant fields
- definitions of new fields
- lists of useful classes or categories
- generalization hierarchies
- functional or causal models

The primary purpose of domain knowledge is to bias the search for interesting patterns. This can be achieved by focusing attention on portions of the data, biasing the extraction algorithms, and assisting in pattern evaluation. The use of the domain knowledge in this way can result in greater efficiency and more useful results. It also can preclude the discovery of potentially useful patterns by leaving portions of the search space unexplored.

For domain knowledge to be useful it needs to be accessible to the discovery system, either directly from a knowledge base or through the user. In a completely automated system all domain knowledge would be encoded and be available online. Most existing discovery systems, however, require substantial guidance from the end user.

The focusing component of a discovery system determines what data should be retrieved from the DBMS. This requires specifying which tables need to be accessed, which attributes should be returned, and which or how many records should be queried. To do this the focus component needs detailed information: it needs to know about the database table structures; it must know which attributes are appropriate for the current task; if it is doing data sampling, it must have a way of randomly selecting the appropriate number of records; and, it must know the input required by the subsequent extraction algorithms.

A DBMS provides query routines for extracting records from tables. Interfacing to a DBMS requires that the system be able to formulate queries and process the results. Realistically, queries on large databases will have to be constructed and submitted as the need for specified data arises.

At the core of a discovery system are the algorithms that extract patterns from data. Virtually any machine-learning or statistical data-analysis algorithm can be incorporated into a KDD system.

Extracted patterns may not always be interesting, and even when they are there may be too many patterns to report all at once. Post evaluation of the extracted patterns may be required to select those of sufficient or greatest interest. This can be achieved by a combination of ways:

- use statistical techniques to verify the significance of the results within the database. Statistical significance alone, however, does not determine the appropriateness of a discovery
- test the results for consistency with available domain knowledge. (Note: testing the consistency of a statement with a body of facts is a hard problem.)
- defer evaluation to the end user

2.2 Intelligent Query Answering

A good question answering system often needs to provide a response that specified more information than strictly required by the question. It should not, however, provide too much information or provide information that is of no use to the person who made the query. Intelligent query answering consists of analyzing the intent of the query and providing generalized, neighborhood or associated information which is relevant to the query.

2.2.1 Why Provide Additional Information?

It is important in defining an efficient cooperative answering method, to understand the general reasons why an expert decides to provide such additional information. The basic idea is that when a person asks a question he is not interested in knowing the answer just to increase his knowledge, but he has the intention of performing some action, and that the answer contains information necessary or useful for realizing this action.

If we accept the basic idea that there is always an underlying intention behind each question, then the expert who wants to be cooperative must try to recognize this intention, in order to determine the most appropriate reaction implicitly expected by the interlocutor.

One method [1] of recognizing a user's intentions is to assume that an expert knows a set of predefined sequences of actions, called plans, the clients may want to realize. Then, when a client asks a question, the expert tries to attach this question to an action. If he succeeds, he then assumes that the user's intention is to perform this action, and the other actions which belong to the same plan. According to this point of view, the appropriate reaction of the expert is to provide the additional information which can be useful in performing this action or others in the same plan. Generally, the answer to a question can be useful not only in executing a plan, but also to help in building or modifying a plan.

However, the specification of the actual plan inference process is not detailed enough to allow it to perform in complex domains. One of the major problems in large domains is the effective management of the large number of potential expectations. Considerable work needs to be done to specify more control heuristics.

2.2.2 Techniques for Intelligent Query Answering in Rule Based Systems

Deductive databases are comprised of syntactic information and semantic information. The syntactic information consists of the *intentional database*(IDB) and the *extensional* *database*(EDB). The semantic information consists of a set of *integrity constraints*(IC). Some techniques have been developed to cope with intelligent query answering in deductive databases.

2.2.2.1 Rule Transformation

Imielinski [34] introduced a new concept of an answer for a query which includes both atomic facts and general rules. In a large knowledge base system, data is represented both in the form of general laws (given as Horn clauses) and assertions representing specific facts (e.g., tuples of relations). It is frequently beneficial to structure the answer for a query in a similar way, i.e., both in terms of tuples, as is traditionally the case, and in terms of general rules. He provided a method of transforming rules by relational algebra expressions built from projection, join, and selection and demonstrated how the answers consisting of both facts and general rules can be generated.

Conceptually, rules are often more informative and easier to comprehend than corresponding sets of derived tuples. For example, the fact that all students who specialize in a given area have to take all courses offered in this area can be represented better by a rule than by a corresponding derived set of tuples. Rules from the database can be transformed by the query if some of the rules can be evaluated after the evaluation of the query without affecting the final result. It is much less expensive to evaluate rules over the answer to the query than over the database state itself, since the result of the query is much smaller than the database. Besides, rule transformation extends the algebraic spirit of query processing from purely relational databases to databases with rules and is also another example of "lazy evaluation"[42] known in the area of programming languages.

Imielinski first described conditions under which single rules can be transformed by single relational operations. Then he generalized the discussion to relational expressions, and finally to sets of rules. Since the transformation of the sets of rules is particularly difficult, it is preferable to decompose the problem of transformation of sets of rules into the transformation of individual rules. In case the given set of rules is not transformable, an equivalent set of rules which can be transformed could be frequently constructed.

2.2.2.2 Query Relaxation

As noted by many researchers, including [2][12][43][54], one form of cooperative behavior involves providing associated information that is relevant to a query. Generalizing a query in order to capture neighboring information is a means to obtain possibly relevant information.

Gaasterland[19] defined a method to relax a query in order to find neighboring information and to control the relaxation process with user constraints. A query can be relaxed in at least three ways:

1. Rewriting a predicate with a more general predicate;
2. Rewriting a constant (term) with a more general constant (term); and
3. Breaking a join dependency across literals in the query.

The first two relaxations are achieved in a general manner using *taxonomy clauses* that define hierarchical type relationships between predicates and constants in the database language. For example, the following clauses define relationships between the predicates ***travel, flight,*** and *train*:

T1: $travel(From, To) \leftarrow serves\_area(A, From), serves\_area(B, To), flight(A, B).$

T2: $travel(From, To) \leftarrow serves\_area(C, From), serves\_area(D, To), train(C, D).$

With the relaxation technique, a user can ask a specific query and get related answers as well as direct answers. However, for large databases, many relaxations may be possible. In order to control the relaxation process, one approach is to allow the user to express their restrictions on the knowledge domain that they would like to have addressed for every query that is asked. A user's restriction on a database can be modeled as a set of constraints, called user constraints. User constraints express the states that a user wants to disallow and the states that a user wants to always persist. Each time a user asks a query, user constraints are applied to the query using semantic query optimization techniques. The resulting query produces answers that satisfy the user's restrictions. Also, a set of heuristics based on cooperative answering techniques are presented for controlling the relaxation process.

CHAPTER 3

DBLEARN: A KNOWLEDGE DISCOVERY SYSTEM

Knowledge discovery is the nontrivial extraction of implicit, previously unknown, and potentially useful information from data [18]. In the previous studies [4][24][26], an attribute-oriented induction method has been developed for knowledge discovery in databases. The method integrates learning-from-examples techniques with database operations and extracts generalized data from actual data in databases. A key to this approach is the attribute-oriented concept tree ascension for generalization which applies well-developed set-oriented database operations and substantially reduces the computational complexity of database learning processes.

In this thesis, the attribute-oriented approach is developed further for the discovery of multiple, statistical rules in large databases(based on number of records in a database).

A special intermediate generalized relation, prime relation, is extracted during attribute-oriented induction, which can be used not only for further generalization and extraction of inquired rules but also for direct extraction of general features and generation of multiple, statistical rules. Many interesting characteristic and discriminant properties of generalized data can be described using such statistical rules. Based upon these principles, a prototyped database learning system, DBLEARN, has been constructed and experiments have been performed on a relatively large Grant-Information Database with satisfactory performance.

3.1 Primitives for Knowledge Discovery in Databases

Three primitives should be provided for the specification of a learning task: task-relevant data, background knowledge, and expected representation of learning results. For illustrative purposes, we examine relational databases only, however, the results can be generalized to other kinds of databases as well [24].

3.1.1 Data relevant to the discovery process

A database usually stores a large amount of data, of which only a portion may be relevant to a specific learning task. Relevant data may extend over several relations. A query can be used to collect task-relevant data from the database.

Task-relevant data can be viewed as examples for learning processes. Undoubtedly, learning-from-examples [41][22] should be an important strategy for knowledge discovery in databases. Most learning-from-examples algorithms partition the set of examples into positive and negative sets and perform generalization using the positive data and specialization using the negative ones [41]. Unfortunately, a relational database does not explicitly store negative data, and thus no explicitly specified negative examples can be used for specialization. Therefore, a database induction process relies mainly on generalization, which should be performed cautiously to avoid over-generalization.

3.1.2 Background knowledge

Concept hierarchies represent necessary background knowledge which directs the generalization process. Different levels of concepts are often organized into a taxonomy of concepts. The concept taxonomy can be partially ordered according to a general-to-specific ordering. The most general concept is the null description (described by a reserved word "ANY"), and the most specific concepts correspond to the specific values of attributes in the database [30]. Using a concept hierarchy, the rules learned can be represented in terms of generalized concepts and stated in a simple and explicit form, which is desirable to most users.

Example 3.1 *The concept hierarchy table of a typical university database is shown in Figure 3.1 , where $A \subset B$ indicates that B is a generalization of A. Notice that*

$$Birthplace(City \subset Province \subset Country)$$

indicates that the concept hierarchy for the attribute *Birthplace* is given by the data stored in the relation *Student* following the partial order: $City, Province$ and $Country$. A concept tree, such as *status* shown in Figure 3.2, represents a taxonomy of concepts of the values in

$\{freshman, ..., senior\} \subset undergraduate$
$\{M.A., M.S., Ph.D.\} \subset graduate$
$\{undergraduate, graduate\} \subset ANY(status)$
$\{0.0 \sim 1.99\} \subset poor$
$\{2.0 \sim 2.99\} \subset average$
$\{3.0 \sim 3.49\} \subset good$
$\{3.5 \sim 4.0\} \subset excellent$
$\{poor, average\} \subset weak$
$\{good, excellent\} \subset strong$
$\{weak, strong\} \subset ANY(GPA)$
......
$Birthplace(City \subset Province \subset Country)$
$\{Canada, U.S.A.\} \subset NorthAmerica$
$\{China, ..., Japan\} \subset Asia$
......
$\{Asia, ..., Europe\} \subset OtherRegions$
$\{NorthAmerica, OtherRegions\} \subset ANY(Country)$
$Birthdate(Day \subset Month \subset Year)$

Figure 3.1: Concept hierarchies in the database.

an attribute domain.

Concept hierarchies can be provided by knowledge engineers or domain experts. This is reasonable for even large databases since a concept tree registers only the distinct discrete attribute values or ranges of numerical values for an attribute which are, in general, not very large and can be input by domain experts. Many concept hierarchies, such as Birthplace in Example 3.1, are actually stored in the database implicitly. Also, concept hierarchies can be discovered automatically or refined dynamically based on the statistics of data distribution and the relationships between attributes [15].

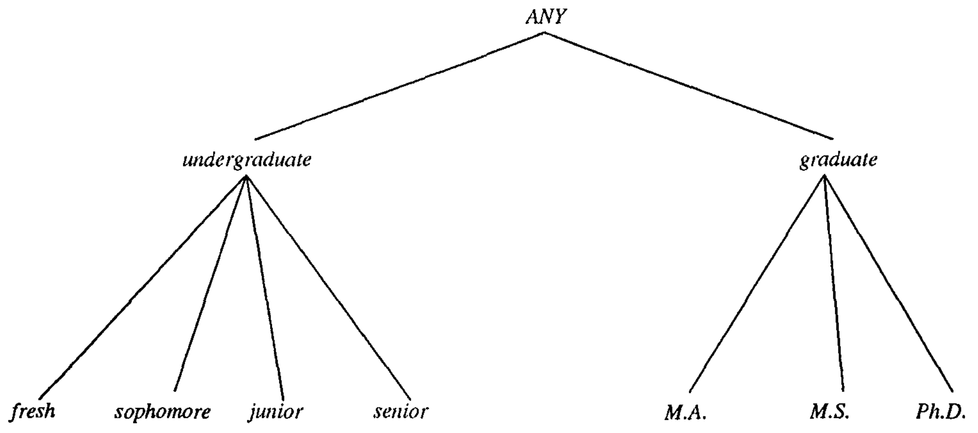

Figure 3.2: A concept tree for status.

Different concept hierarchies can be constructed on the same attribute based on different viewpoints or preferences. For example, the birthplace could be organized according to administative regions, geographic regions, sizes of cities, etc. Usually, a commonly referenced concept hierarchy is associated with an attribute as the default one. Other hierarchies can be chosen explicitly by preferred users in the learning process.

Notice that different kinds of set-subset relationships are represented as concept hierarchies in our study. For example, China is a *part-of* Asia, while senior is a *specialization-of* undergraduate. They are treated similarly in our concept hierarchies as set-subset relationships since they play similar roles in attribute-oriented induction. However, it will be useful to discriminate different roles in a detailed semantic analysis of learning intentions.

3.1.3 Representation of learning results

Many kinds of rules, such as characteristic rules, discriminant rules, statistical rules, etc. can be discovered by induction processes. A characteristic rule is an assertion which characterizes a concept satisfied by all or most of the examples in the class undergoing learning (called the target class). For example, the symptoms of a specific disease can be summarized by a characteristic rule. A discriminant rule is an assertion which discriminates a concept of the class being learned (the target class) from other classes (called contrasting classes). For example, to distinguish one disease from others, a discriminant rule should summarize the symptoms that discriminate this disease from others.

From a logical point of view, each tuple in a relation is a logic formula in conjunctive normal form, and a data relation is characterized by a large set of disjunctions of such conjunctive forms. Thus, both the data for learning and the rules discovered can be represented in either relational form or first-order predicate calculus.

A relation which represents intermediate (or final) learning results is called an intermediate (or a final) generalized relation. In a generalized relation, some or all of its attribute values are generalized data, that is, nonleaf nodes in the concept hierarchies. An attribute in a (generalized) relation is at a desirable level if it contains only a small number of distinct values in the relation. A user or an expert may like to specify a small integer for an attribute as a desirable attribute threshold. In this case, an attribute is at the desirable level if it contains no more distinct values than its attribute threshold. Moreover, the attribute is at the minimum desirable level if it would contain more distinct values than the threshold when generalized or specialized to a level lower than the current one. A special intermediate generalized relation R' of an original relation R is the prime relation of R if every attribute in R' is at the minimum desirable level.

Some learning-from-examples algorithms require the final learned rule to be in conjunctive normal form [41]. This requirement is unreasonable for large databases since the generalized data often contain different cases. However, a rule containing a large number of disjuncts indicates that it is in a complex form and further generalization should be performed. Therefore, the final generalized relation should be represented by either one tuple (a conjunctive rule) or a small number (usually 2 to 8) of tuples corresponding to a disjunctive rule with a small number of disjuncts. A system may allow a user to specify the preferred generalization threshold (or generalized relation threshold), a maximum number of disjuncts of the resulting formula. For example, if the threshold value is set to three, the final generalized rule will consist of at most three disjuncts.

Exceptional data often occur in a large relation. It is important to consider exceptional cases when learning in databases. Statistical information helps learning algorithms handle exceptions and/or noisy data [42][7]. A special attribute, vote, can be added to each generalized relation to register the number of tuples in the original relation which are generalized to the current tuple in the generalized relation. The attribute vote carries database statistics and supports the pruning of scattered data and the generalization of the concepts which take a majority of votes. The final generalized rule will be the rule which represents the characteristics of a majority number of facts in the database (called an approximate rule) or indicates statistical measurement of each conjunct or disjunct in the rule (called a statistical rule).

3.2 Basic Principles of Attribute-Oriented Induction

A set of basic principles for attribute-oriented induction in relational databases are summarized as follows [24].

1. Generalization only on the relevant set of data: *Generalization should be performed only on the set of data in the database which is relevant to the learning request.*
2. Generalization on the smallest decomposable components:*Generalization should be performed on the smallest decomposable components (or attributes) of a data relation.*
3. Attribute removal: *If there is a large set of distinct values for an attribute but (1) there is no higher level concept provided for the attribute, or (2) its higher-level concepts are expressed in another attribute of the same tuple, the attribute should be removed in the generalization process.*
4. Concept tree ascension: *If there exists a higher level concept in the concept tree for an attribute value of a tuple, the substitution of the value by its higher level concept generalizes the tuple.*
5. Vote propagation: *The value of the vote of a tuple should be carried to its generalized tuple and the votes should be accumulated when merging identical tuples in generalization.*
6. Attribute threshold control: *If the number of distinct values of an attribute in the target class is larger than its attribute threshold, further generalization on this attribute should be performed.*

Remarks: *The above strategies are correct and necessary for the extraction of generalized rules from databases.*

Reasoning: Principle 1 is based on the concept of query processing in databases. Principle 2 is based on the least commitment principle (commitment to minimally generalized concepts) which avoids over-generalization. Principle 3 corresponds to the generalization rule, dropping conditions, in learning-from-examples [41]. Principle 4 corresponds to the generalization rule, climbing generalization trees, in learning-from-examples [41]. Principle 5 is based on the merging of identical tuples. Principle 6 is based on the desirability of representation of each attribute at its desirable level.

The attribute-oriented induction process is illustrated in Example 3.2.

Example 3.2 *Let the university database be modeled by a deductive ER model [28] in which the extensional database (EDB) is mapped to the following schema.*

$Course(Cnum, Title, Semester, Department, Instructor, TA, Enrollment, Time).$

...

$Student(Name, Status, Sex, Major, Birthdate(Day, Month, Year),$
$Birthplace(City, Province, Country), GPA).$

Suppose a truth-valued virtual attribute $IsTA$ is defined in $Student$, and the value is true only if the student is a TA in some course, i.e., the computation of $IsTA$ involves the join of two relations, *Student* and *Course*. Suppose that the learning task is to learn characteristic rules for *cs (computing science)* students relevant to the attributes *Name, Sex, Status, Age, Birthplace, GPA, and IsTA* using the default concept hierarchies presented in Figure 1.2 and the default threshold values. The learning task is represented in DBLEARN as follows.

learn characteristic rule for $Major$ = "cs"
from $Student$
in relevance to $Name, Sex, Age, Birthplace, GPA, IsTA$

For this learning request, preprocessing is performed by selecting *cs* students and projecting on relevant attributes *Name, Sex*, etc. A special attribute *vote* is attached to each tuple of the result relation with its initial value set to 1. Such a preprocessed data relation is called an initial relation.

Since there is no higher level concept specified on the first attribute *Name*, the attribute should be removed in generalization, which implies that a generalized rule cannot be characterized by the attribute *Name*. The *Birthdate* information can be transformed into *Age* since the learning task is interested not in *Birthdate* but in *Age*. Moreover, *city* and *province* attributes should also be removed since they contain a large number of distinct values but their generalized information is contained in the attribute *Birthplace*(*country*) in the same tuple. After removing these attributes, the data relation contains 6 remaining attributes: *Status, Sex, Age, Birthplace(country), IsTA and GPA* (plus one special attribute *vote*).

Status	Sex	Age	Birthplace	GPA	IsTA	vote
grad	M	25_30	Canada	good	Y	8
grad	F	25_30	Canada	excellent	Y	2
...	...	...	...	...	...	...
underg	M	16_25	Asia	good	N	6

Table 3.1: A prime relation from the initial set of data.

By removing the removable attribute and generalizing each generalizable attribute to its minimum desirable level, the initial data relation is generalized to the prime relation. In our example, the prime relation, as shown in Table 3.1, contains a small number of distinct values in each attribute as follows: *Status*: {*grad, undergrad*}, *Sex*: {*M, F*}, *Age*: {*16_25, 26_30, >30*}, *Birthplace*: {*Canada, USA, Asia, Europe*}, *GPA*: {*poor, average, good, excellent*}, and *IsTA*: {*Yes, No*}.

The basic attribute-oriented induction process is summarized in the following algorithm.

Algorithm 3.1 *Attribute-oriented induction in the derivation of the prime relation from a large relational data set.*

Input: (i) A learning task-relevant data set R, of arity n with a set of attributes A_i $(1 \leq i \leq n)$; (ii) a set of concept hierarchies, H_i on attribute A_i; and (iii) a set of attribute thresholds, T_i for attribute A_i.

Output. The prime relation *R'*.

Method.

```
 $R_t = R$; /* $R_t$ is a temporary relation. */
 for each attribute $A_i$ in $R_t$ do {
  if $A_i$ is removable then remove $A_i$;
```

```
    if $A_i$ is not at the desirable level
    then generalize $A_i$ to the desirable level.
    }
    /* Identical tuples in $R_t$ are merged with the number of identical tuples registered in vote. */
  $R' := R_t$.
```

Notice that generalization for each attribute A_i is implemented by (1) collecting the distinct A_i values in the relation, (2) computing the minimum desirable level L, and (3) generalizing the attribute to this level L by replacing each value in A_i's with its corresponding superordinate concept in H_i at level L.

3.3 Extraction of Generalized Rule

Since only attribute thresholds are utilized in the attribute-oriented induction in Algorithm 3.1, the derived prime relation may often contain more tuples than the generalization threshold. Two methods have been developed for the extraction of generalized rules from the prime relation: (1) further generalization to a final generalized relation confined by the generalization threshold and extraction of the inquired rule(s), and (2) direct extraction of generalized features and presentation of feature-based multiple rules.

3.3.1 Further generalization and rule extraction

Method 1 is realized based on the following two additional principles.

1. Generalization threshold control: *If the number of tuples of a generalized relation in the target class is larger than the generalization threshold, further generalization on the relation should be performed.*

2. Rule formation: *A tuple in a final generalized relation is transformed to conjunctive normal form, and multiple tuples are transformed to disjunctive normal form.*

Notice that during further generalization by generalization threshold control, there are usually alternative choices at selection of a candidate attribute for further generalization. Criteria, such as the preference of a larger reduction ratio on the number of tuples or the number of distinct attribute values, the simplicity of the final learned rules, etc. can be used for selection. Interesting rules can often be discovered by following different paths leading to several generalized relations for examination, comparison and selection. Following different paths corresponds to the way in which different people may learn differently from the same set of examples. The generalized relations can be examined by users or experts interactively to filter out trivial rules and preserve interesting ones [57].

Let the default generalization threshold be 8. If the prime relation in Table 1 consists of 40 tuples, it is obviously necessary to perform further generalization. Suppose the preference is to retain 3 attributes: *Status, Birthplace* and *GPA*. Then, other attributes are generalized to *ANY* and removed from the generalized relation. *Birthplace and GPA* are further generalized, which results in the final generalized relation with seven tuples, as shown in Table 3.2.

By rule transformation, the final generalized relation is equivalent to rule (r_1), that is, a computing science student is in one of the following seven cases: (1) *North-America-born graduate students with strong GPA (13.2%)*, ..., and (7) *Other-regions-born undergraduate students with weak GPA (4%)*. Notice that since a charateristic rule characterizes all of the

Status	Birthplace	GPA	vote
grad	NorthAmerica	strong	33
grad	NorthAmerica	weak	2
grad	OtherRegions	strong	15
underg	NorthAmerica	strong	105
underg	NorthAmerica	weak	65
underg	OtherRegions	strong	20
underg	OtherRegions	weak	10

Table 3.2: Final generalized relation.

data in the target class, its then-part represents the necessary condition of the class.

$$
\begin{aligned}
(r_1)\quad \forall(x)\quad & cs\_student(x) \rightarrow \\
& (Birthplace(x) \subset NorthAmerica \wedge Status(x) \subset graduate \\
& \wedge\ GPA(x) \subset strong)[13.2\%] \\
& \vee \cdots\cdots \vee \\
& (Birthplace(x) \subset OtherRegions \wedge Status(x) \subset undergraduate \\
& \wedge\ GPA(x) \subset weak)[4\%].
\end{aligned}
$$

Rule r_1 is a statistical rule. It can also be expressed as an approximate rule by dropping the conditions or conclusions with negligible probabilities.

Status	Sex		Age			Birthplace				GPA				IsTA		vote
	M	F	16_25	26_30	>30	Canada	USA	Asia	Europe	poor	avg	good	exclnt	Y	N	
grad	40	10	10	20	20	30	5	10	5	1	1	30	18	30	20	50
underg	120	80	140	60	0	130	40	30	0	15	60	100	25	0	200	200
total	160	90	150	80	20	160	45	40	5	16	61	130	43	30	220	250

Table 3.3: A Status feature table mapped from the prime relation.

3.3.2 Direct extraction of generalized features and statistical rules

Since every feature (attribute value) has been generalized to a desirable level in the prime relation, interesting relationships and statistics of features can be extracted directly from the derived prime relations. The generation of general relationships and rules can be facilitated by extraction of generalized feature tables from the prime relation.

Example 3.3 *Let Table 1 be the prime relation generalized from the learning task. Generalized feature tables can be extracted from the prime relation. For example, to compare students with different status (graduate vs. undergraduate), the prime relation can be mapped into a Status feature table (Table 3.3).*

The *Status* feature table consists of 3 rows: each of the two distinct *Status* values in the prime relation {"*grad*", "*undergrad*"} corresponds to one row, and the last row (*total*) is the summation of information in the previous rows. It consists of 5 major columns, each corresponding to one attribute in the prime relation, plus one special column for *vote.* Each major column in the table is further divided into k subcolumns, each corresponding to one distinct value in the attribute. For example, GPA is divided into 4 subcolumns: *poor, avg, good, exclnt,* each corresponding to one distinct value in GPA.

The table contents are derived from the prime relation as follows. Each slot in the table (except for the last row) corresponds to the number of occurrences of the corresponding values in the prime relation. For example, the slot for "*grad*" and "*good (GPA)*" corresponds to the number of grad's with good GPAs, that is, the summation of all the votes of those rows with $Status$ = "*grad*" and GPA = "*good*" in the prime relation. The special column vote registers the number of occurrences of the corresponding class in the relation. For example, 50 in "*grad*" indicates that there are in total 50 graduates in the prime relation. The special row *total* summarizes the total number of occurrences with each feature in all the classes. For example, total = 160 in the column "$Sex = M$" indicates that there are totally 160 male students computed in the prime relation. □

In general, we have the following algorithm for the extraction of a feature table from a prime relation.

Algorithm 3.2 *Extraction of the feature table T_A for an attribute A from the prime relation R'.*

Input: A prime relation R' consists of (i) an attribute A with distinct values $\{a_1, ..., a_m\}$, (ii) k other attributes B_1, ..., B_k (suppose different attributes have unique distinct values), and (iii) a special attribute, *vote*.

Output. The feature table T_A for the attribute A.

Method.

1. The feature table T_A consists of $m + 1$ rows and $l + 1$ columns, where l is the total number of distinct values in all the k attributes. Each slot of the table is initialized

to 0.

2. Each slot in T_A (except the last row) is filled by the following procedure,

```
for each row r in R' do {
 for each attribute B_i in R' do
  $T_A[r.A, r.B_i] := T_A[r.A, r.B_i] + r.vote;$
 $T_A[r.A, vote] := T_A[r.A, vote] + r.vote;$ }
```

3. The last row p in T_A is filled by the following procedure:

```
for each column s in T_A do
 for each row t (except the last row p) in T_A do
 $T_A[p, s] := T_A[p, s] + T_A[t, s];$ □
```

Remark. *Algorithm 3.2 correctly registers the number of occurrences for each general feature in the prime relation R'.*

Reasoning. Following the algorithm, each tuple in the prime relation is examined exactly once with every feature registered in the corresponding slot in the feature table. Their column-wise summation is registered in the last row.

The extracted feature table can be used to derive the relationships between the classification attribute and other attributes at a high level. For example, a rule, *all the TAs are graduate students*, can be extracted from Table 3 based on the fact that the class *grad* takes all the *IsTA* count. The table is especially useful for extraction of multiple, statistical rules. For example, from the first row *grad* and the first major column *Sex*, we have:

$$grad(x) \rightarrow male(x)[80\%] \vee female(x)[20\%],$$

which indicates that *80% of graduate (cs) students are male and 20% are female.*

In general, the following algorithm is summarized for the extraction of generalized rules from a feature table.

Algorithm 3.3 *Extraction of generalized rules from the feature table T_A.*

Input: A feature table T_A for the attribute A, where A has a set of distinct generalized values $\{a_1, ..., a_m\}$. Another attribute B in the table has a set of distinct generalized values $\{b_1, ..., b_n\}$. The slot of the table corresponding to the row with the value a_i and the column with the value b_j is referenced by $T_A[a_i, b_j]$.

Output. A set of generalized rules relevant to A and B extracted from the feature table.

Method.

1. For each row a_i, the following rule is generated in relevance to attribute B, which presents the distribution of different generalized values of B in class a_i.

   $a_i(x) \rightarrow b_1[p_{i1}] \vee ... \vee b_n[p_{in}]$.

   where p_{ij} is the probability that the value b_j of B is in class a_i, which is computed by,

   $p_{ij} = T_A[a_i, b_j]/T_A[a_i, vote]$.

2. For each column b_j, the following rule is generated in relevance to all the classes, which presents the distribution of the generalized value b_j of B among all the classes.

   $b_j(x) \rightarrow a_1[q_{1j}] \vee ... \vee a_m[q_{mj}]$.

   where q_{ij} is the probability that the value b_j of B is distributed in class a_i among all the classes, which is computed by,

   $q_{ij} = T_A[a_i, b_j]/T_A[total, b_j]$. □

Remark. *Algorithm 3.3 correctly generates relationships between two attributes A and B at a high level using the feature table.*

Reasoning. Algorithm 3.2 shows that the feature table registers all of the number of occurrences of each feature and two specific properties, *total* and *vote*, in each class. The above algorithm extracts generalized rules from each column and each row in the feature table by computing the proportion of the number of occurrences of each generalized feature vs. its corresponding total number of occurrences in each row or each column.

3.4 Experiments on the NSERC Grant Information Database

Based upon the attribute-oriented induction technique, a prototyped experimental database learning system, **DBLEARN**, has been constructed. The system is implemented in C with the assistance of UNIX software packages LEX and YACC (for compiling the **DBLEARN** language interface) and operates in conjunction with the SyBase DBMS software. A database learning language for **DBLEARN** is specified in an extended BNF grammar. The architecture of the system is shown in Figure 3.3. In the learning process, the DBLEARN system first accepts the user's request through the user-interface. Based on the specified learning task, the DBLEARN system obtains the relevant data from a database and the relevant concept hierarchies from a file. The learning program performs attribute-oriented induction to extract generalized rules. After learning is performed, the learning result is reported to the user through the user-interface.

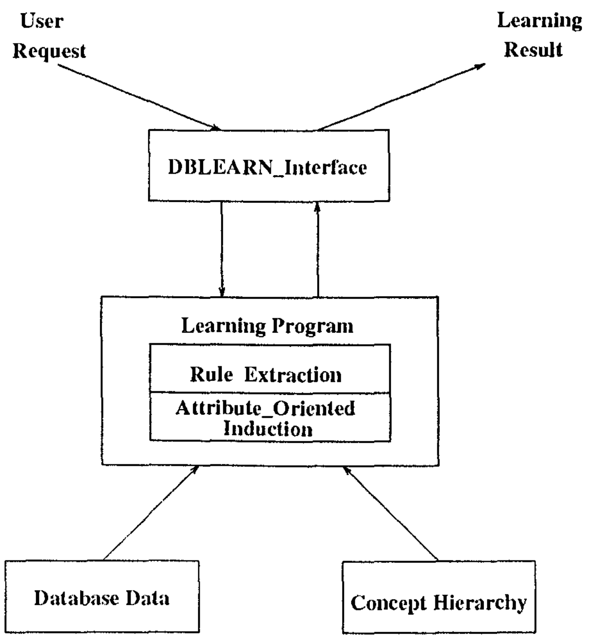

Figure 3.3: The architecture of DBLEARN.

3.4.1 The Database

Experimentation using DBLEARN has been conducted on a real database, the Grants Information database, which contains the information about the research grants awarded by NSERC (*the Natural Sciences and Engineering Research Council of Canada*) in the year of 1990-1991. The central table *Award* in the database is made up of tuples each of which describes an award by NSERC to a researcher. The fields constituting each tuple specify the different properties of the award, including the name of the recipient, the amount of the award and so on. In the schema diagram shown in Figure 3.4, nodes representing the properties of awards are represented by nodes linked to the *Award* node.

There are a number of subsidiary tables which record details about some of the properties of awards, (e.g., the province of the organization in which the recipient is to carry

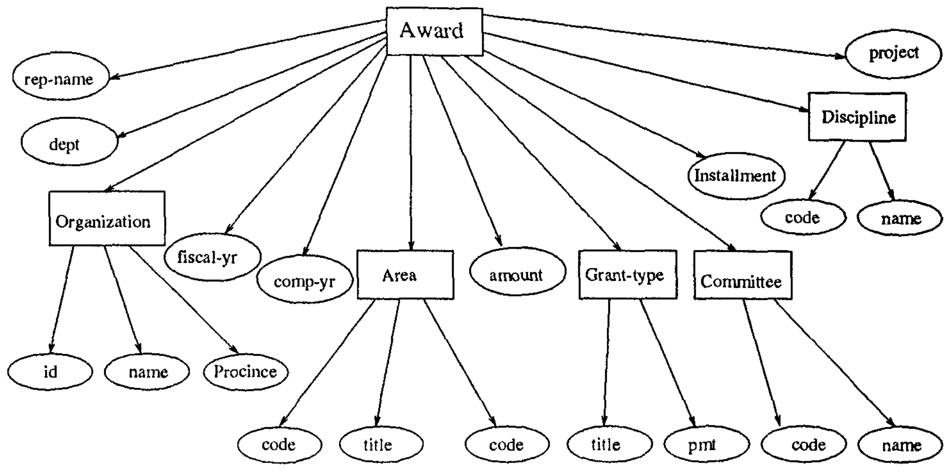

Figure 3.4: The NSERC database schema.

out the research). Most subsidiary tables are used simply to associate a code denoting a particular entity to phrases describing the entity. In the schema diagram, table are specified by rectangular nodes.

3.4.2 Background Knowledge

Recall that the background knowledge in **DBLEARN** is represented by a set of concept hierarchies. In each hierarchy, the most general concept is the null description (described by a reserved word "ANY"), and the most specific concepts correspond to the specific values of attributes in the database. Figure 3.5 shows the concept hierarchy for provinces in Canada, where $A \subset B$ indicates that B is a generalization of A. Notice that the superordinate concepts for 3 provinces *B.C.*, *Ontario*, and *Quebec* remain ungeneralized since these 3 provinces take most of research grants and it is our intention to distinguish these 3 from other provinces.

$\{BritishColumbia\} \subset B.C.$
$\{Alberta, Saskatchewan, Manitoba\} \subset Prairies$
$\{Ontario\} \subset Ontario$
$\{Quebec\} \subset Quebec$
$\{NewBrunswick, NovaScotia, Newfoundland, Prince\_Edward\_Island\}$
$\subset Maritime$
$\{B.C., Prairies, Ontario, Quebec, Maritime, Others\} \subset ANY(province)$

Figure 3.5: A concept hierarchy for attribute *province*.

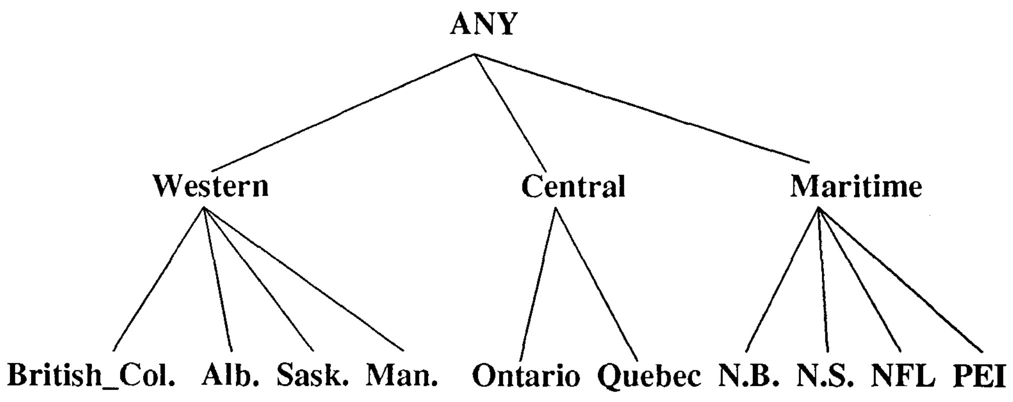

Figure 3.6: A concept hierarchy before adjustment.

Other concept hierarchies, such as {1 ...19,999} ⊂ 1_20K, {20,000 ... 39,999} ⊂ 20_40K, ..., {26000 ... 26499} ⊂ AI (where 26000, ... and 26499 represent NSERC discipline codes), are also stored in the concept hierarchy table.

The concept hierarchies were first constructed by domain experts. However, a concept hierarchy can be adjusted automatically in **DBLEARN** based on clustering behavior and database statistics. A concept hierarchy for provinces in Canada provided by a domain expert or a user could look like Figure 3.6.

The automatic adjustment is performed by first obtaining the distribution of attribute

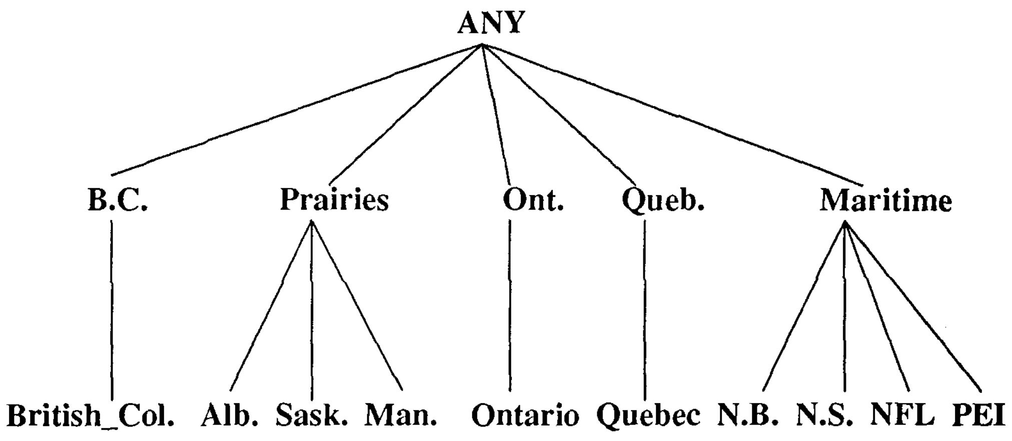

Figure 3.7: A concept hierarchy after adjustment.

values in the database and then spliting or merging node/nodes in order to make the number of tuples covered by each node in the same level of the hierarchy even. After the adjustment, the hierarchy in Figure 3.6 looks like the one shown in Figure 3.7.

3.4.3 Experimental Examples

Many learning requests have be posed to this database during our experimentation. Interesting knowledge rules/relationships about NSERC research grant awards in relevance to geographic location, research areas, etc. have been discovered by our experimentation. One such experimental example is illustrated as follows.

Example 3.4 *Let the query be to discover a characteristic rule for NSERC support of operating grants for AI (Artificial Intelligence) researchers in relevance to the geographical locations, the number of grants and the amount distribution of the grants in 1990 to 1991. The learning task is presented in* DBLEARN *as follows.*

learn characteristic rule for *disc_code = "AI"*

from *award*

where*grant_code = "Operating_Grants"*

in relevance to *amount, province, prop(vote), prop(amount)*

Notice that *prop(attribute)* is a built-in function which returns the percentage of the summation of the *attribute* value in the generalized tuple divided by the summation of the same *attribute* value in the whole generalized relation.

When the query is posed to the system, relevant data are collected by data retrieval from the Grant Information Database. Then attribute-oriented induction is performed on the collected data. The learning result of the query is presented in Table 3.4. The row "*Amount = 20_40K, Geo_Area = B.C., prop(#_of_grants) = 12.7%, and prop(amount) = 16.3%*" indicates that for the Operating Grants in AI in the amount between $20,000 and $39,999, B.C. researchers take 12.7% of the total number of grants and 16.3% of the total amount of grants. The last row contains the summary information of the entire generalized relation. Some negiligible proportion (about 0.2%) of the AI operating grants scattered across Canada are ingored in the table. Thus, the total number of grants in the table takes 99.8% of the total available AI operating grants.

Notice that the relationships between amount_category, geo-area, number_of_grants, amount_of_grants, etc. can be also presented in the pairwise form, when necessary, using the extracted prime relation. The system interacts with users for explicit instructions on the necessity of such a presentation.

Discipline = "AI"		grant_code = "Operating Grant"	
Amount	Geo_Area	prop(#_of_grants)	prop(amount)
1_20K	B.C.	5.6%	4.1%
1_20K	Prairies	15.5%	10.3%
1_20K	Quebec	14.1%	9%
1_20K	Ontario	25.3%	17.8%
1_20K	Maritime	2.8%	1%
20_40K	B.C.	12.7%	16.3%
20_40K	Prairies	5.6%	6.2%
20_40K	Ontario	9.8%	13%
20_40K	Maritime	1.4%	1.7%
40_60K	B.C.	1.4%	4%
40_60K	Ontario	4.2%	11.3%
>60K	Quebec	1.4%	4.2%
$1,464,250	Canada	99.8%	98.9%

Table 3.4: Generalized relation for AI Operating Grants.

The performance of the **DBLEARN** system is satisfactory. The average response time of the above query (including the SyBase data retrieval time) is about 20 seconds on an IPX SPARC workstation.

We present two more experimental examples as follows, which give some interesting results and show how DBLEARN works in some complicated cases.

Example 3.5 *Let the query be to discover a characteristic rule for NSERC support of operating grants for computer science researchers in relevance to the geographical locations, the number of grants and the amount distribution of the grants in 1990 to 1991. The learning task is presented in* **DBLEARN** *as follows.*

learn characteristic rule for *"CS Operating Grants"*

from *award A, organization O, grant_type G*

where *O.org_code* = *A.org_code* **and** *G.grant_order* = *"Operating_Grants"*

  and *A.grant_code* = *G.grant_code* **and** *A.disc_code* = *"Computer"*

in relevance to *amount, province, prop(vote), prop(amount)*

using table threshold 18

Interacting with the user who issued the query, DBLEARN generates a feature table for attribute *amount* shown in Table 3.5. Multiple statistical rules can be extracted from this table. Two examples are shown as follows.

$$\begin{aligned}\forall(x)\ \ & BC\_CS\_Operating\_Grants(x) \leftarrow \\ & (amount = 0 - 20Ks[52.174\%]) \wedge (amount = 20Ks - 40Ks[37.683\%]) \wedge \\ & (amount = 40Ks - 60Ks[8.697\%]) \wedge (amount = 40Ks - 60Ks[1.446\%])\end{aligned}$$

$$\begin{aligned}\forall(x)\ \ & Prairies\_CS\_Operating\_Grants(x) \leftarrow \\ & (amount = 0 - 20Ks[59.97\%]) \wedge (amount = 20Ks - 40Ks[37.683\%]) \wedge \\ & (amount = 40Ks - 60Ks[1.446\%]) \wedge (amount = 40Ks - 60Ks[1.891\%])\end{aligned}$$

Example 3.6 *Let the query be to discover a discriminant rule for NSERC support for computer science reseachers in Ontario in contrast to that in Newfoundland , which is in relevance to the discipline, the grant type and the amount of grant. The learning task is presented in* **DBLEARN** *as follows.*

learn discriminant rule for *"Ont_Grants"*

where *O.province* = *"Ontario"*

amount	province					vote
	B.C.	Prairies	Ontario	Quebec	Maritime	
0-20Ks	36	40	119	67	33	295
20Ks-40Ks	26	26	62	25	5	144
40Ks-60Ks	6	1	25	5	0	37
60Ks-	1	2	6	1	0	10
Total	69	69	212	98	38	486

Table 3.5: An amount feature table.

in contrast to *"Newfoundland_Grants"*

where *O.province* = *"Newfoundland"*

from *award A, organization O, grant_type G*

where *A.grant_code* = *G.grant_code* **and** *A.org_code* = *O.org_code*

and *A.disc_code* = *"Computer"*

in relevance to *disc_code, grant_order, amount*

Since the task is to learn a discriminant rule, two data sets should be first retrieved by relational operations: (1) the target class: the grants awarded to Ontario computer science researchers, and (2) the contrasting class: the grants awarded to Newfoundland computer science researchers.

Generalization is performed synchronously in both classes. The prime relation is shown in Table 3.6. Overlapping tuples are marked by "*".

After excluding the properties that overlap in both classes in the prime relations, a final generalized relation is generated as shown in Table 3.7.

Learning Concept	disc_code	grant_order	amount	votes	mark
Ontario_CS_Grants	Computer	Operating_Grants	0-20Ks	119	*
	Computer	Operating_Grants	20Ks-40Ks	62	
	Computer	Other	0-20Ks	10	*
	Computer	Other	20Ks-40Ks	10	*
	Computer	Operating_Grants	40Ks-60Ks	25	
	Computer	Other	60Ks-	7	
	Computer	Other	40Ks-60Ks	5	
	Computer	Strategic_Grants	60Ks-	8	
	Computer	Operating_Grants	60Ks-	6	
	Computer	Strategic_Grants	40Ks-60Ks	1	
Newfoundland_CS_Grants	Computer	Operating_Grants	0-20Ks	9	*
	Computer	Other	0-20Ks	1	*
	Computer	Other	20Ks-40Ks	1	*

Table 3.6: A prime relation for both the target and the contrasting classes .

disc_code	grant_order	amount	votes	mark
Computer	Operating_Grants	20Ks-40Ks	62	
Computer	Operating_Grants	40Ks-60Ks	25	
Computer	Other	60Ks-	7	
Computer	Other	40Ks-60Ks	5	
Computer	Strategic_Grants	60Ks-	8	
Computer	Operating_Grants	60Ks-	6	
Computer	Strategic_Grants	40Ks-60Ks	1	

Table 3.7: A final generalized relation.

CHAPTER 4

A Data Model for Knowledge-Rich Databases

4.1 Definition

As an extension to the logic data model proposed in deductive database research [56], a *knowledge-rich data model* is constructed for databases with both deduction and knowledge discovery capabilities.

Definition. **A knowledge-rich database** (*KDB*) consists of six components: (1) *Schema*, a knowledge-rich database schema; (2) *EDB, an extensional database; (3) IDB, an intensional database; (4) H, a set of concept hierarchies; (5) GDB, a generalized database*; and *(6) KDT, a set of knowledge discovery tools* , defined as follows. *KDB* = {*Schema, EDB, IDB, H, GDB, KDT* }.

- **Schema**, a *knowledge-rich database schema* , describes the general structure and organization of KDB including (i) *physical* and *virtual* entities, attributes and relationships, and (ii) the organization of rules, integrity constraints and concept hierarchies, based on a deductive entity-relationship data model [28].
- **EDB**, an *extensional database* , consists of a set of predicates, each corresponding to an extensional data relation.
- **IDB**, an *intensional database* , consists of a set of *deduction rules* and *integrity constraints* (*ICs*).
- **H**, a set of *concept hierarchies* , specifies taxonomies of concepts on top of primitive data in extensional and intensional databases.
- **GDB**, a *generalized database* , consists of a set of *generalized rules* which summarize the regularities of the data at a high level.
- **KDT**, a set of *knowledge discovery tools* , performs knowledge discovery efficiently in databases, when necessary.

The first component, *Schema* , follows from a deductive entity-relationship data model [28] which extends an entity-relationship model [10][55] to incorporate rules, integrity constraints and complex data objects for deductive databases. The second and third components, *EDB* and *IDB* , are the same as in deductive databases [56] except that IDB rules can be defined by some nonprimitive data as well. Notice that a rule (or an integrity constraint) in the IDB can be first discovered by a knowledge discovery process and then be *recognized* and stored in the IDB as a regular rule or integrity constraint. However, once a *discovered* regularity is *recognized* and stored, it will play the same role as the *originally defined* one, which means that any data in the EDB violating this constraint have to be discarded first. Thus, we assume that all of the rules in IDB are defined ones.

The last three components, *H, GDB* and *KDT* , are the newly introduced knowledge discovery components which are used to incorporate discovered knowledge and knowledge discovery in databases.

H , a set of *concept hierarchies* , represents the relationships among concepts at different levels. The information about concept hierarchies can be *provided* by knowledge engineers or domain experts or be *discovered* automatically or semi-automatically using knowledge discovery tools based on the statistics of data distribution in databases and the relationships among different attributes [32]. Many concept hierarchies are implicitly stored in the database. For example, the hierarchical relationship among "*city* ", "*province* " and "*country* " attributes are usually stored in the database and can be made explicit at the schema level by indicating a part-of-hierarchy, "*city* $\subset$ *province* $\subset$ *country*". It is realistic to have some concept hierarchies provided by knowledge engineers or domain experts even in a large database system since a concept hierarchy registers *only* the *distinct* discrete attribute values or *ranges* of numerical values for an attribute, which is, in general, not very large. Further, by providing different concept hierarchies, users or experts may have preference to control the knowledge discovery or intelligent query answering processes.

GDB , the *generalized database* , is another important component in the knowledge-rich database. Since there are usually a very large set of generalized rules which can be extracted from any interesting subset of data in a database by performing generalization in different directions, it is unrealistic to store all of the possible generalized rules. However, it is often useful to store some generalized rules or intermediate generalized relations in the GDB based upon the importance of the knowledge and the frequency of inquiries.

The stored generalized rules are useful for querying database knowledge and semantic query optimization. Notice that a stored generalized rule should be incrementally updated after the updates of the relevant data set in order to preserve its correctness. This can be performed by an incremental learning algorithms provided in knowledge discovery tools [18][24].

The last component, *KDT* , consists of a set of *knowledge discovery tools* , which could be a set of knowledge discovery algorithms or a database-oriented knowledge discovery subsystem, such as INLEN [35], KDW++ [18], DBLEARN [24], etc. Since a knowledge-rich database stores only a small portion of all of the possible generalized knowledge, it is often necessary to evoke a knowledge discovery process dynamically and extract general regularity from a specific set of data relevant to the query. The KDT tools can be used for on-line knowledge discovery and intelligent query answering.

4.2 An Example

The *university* database presented in Example 4.1 is an illustrative example of such a knowledge-rich database.

Example 4.1 *Let a university database be modeled by a deductive entity-relationship model in which the extensional database (EDB) is mapped to a relational-like schema presented in Figure 4.1, where Cnum stands for* course number , *TA for* teaching assistant , *and GPA for* grade point average .

Course (Cnum, Title, Semester, Department, Instructor, TA, Enrollment, Time).
Professor (Pname, Department, Salary).
Student (Sname, Status, Sex, Major, Birth_date(Day, Month, Year),
Birth_place(City, Province, Country), GPA).
Grading (Student, Course, Grade).

Figure 4.1: Schema of the University database.

The concept hierarchies defined in the database are shown in Figure 4.2. The first three lines imply that the primitive data for *Status* is $\{freshman, ..., Ph.D.\}$, and their corresponding high-level data is *undergraduate* or *graduate* respectively. The entry "*Birth_place*$($*City* $\subset$ *Province* $\subset$ *Country*$)$" indicates that the concept hierarchy for the attribute *Birth_place* is given by the data stored in the relation *Student* according to the part_of hierarchy: *City, Province* and *Country*. For example, a tuple,

$$Student(Tom\_Jackson, ..., Birth\_place(Vancouver, BC, Canada), ...),$$

indicates that *Vancouver* is a part of British Columbia (*BC*), which is in turn a part of *Canada*, in the concept hierarchy for *Birth_place.*

Notice that there are many different kinds of hierarchical relationships among data in a database, such as part_of, is_a, subset_of, etc., which may play different roles in conceptual analysis. The different semantics among concept hierarchies are not essential in the knowledge discovery algorithm itself since different concepts are generalized to their corresponding higher level concepts by following their corresponding concept hierarchies in a similar manner in the generalization process. However, such semantic differences will be important in the analysis of query intent and provision of intelligent answers.

$\{freshman, sophomore, junior, senior\} \subset undergraduate$
$\{M.S., M.A., Ph.D.\} \subset graduate$
$\{undergraduate, graduate\} \subset ANY(status)$
$\{biology, chemistry, computing, ..., physics\} \subset science$
$\{literature, music, ..., painting\} \subset art$
$\{science, art\} \subset ANY(major)$
$\{0.0 \sim 1.99\} \subset poor$
$\{2.0 \sim 2.99\} \subset average$
$\{3.0 \sim 3.49\} \subset good$
$\{3.5 \sim 4.0\} \subset excellent$
$\{poor, average, good, excellent\} \subset ANY(GPA)$
$Birth\_place(City \subset Province \subset Country).$
$Birth\_date(Day \subset Month \subset Year).$

Figure 4.2: A concept hierarchy table of the database

IDB rules are defined on top of EDB predicates. For example, *award_candidate* and *pre_requisite* are two IDB predicates defined as follows.

(1a) *award_candidate(Name)* $\leftarrow$

status(X) $=$ *graduate, gpa(Name)* $\geq$ *3.75.*

(1b) *award_candidate(Name)* $\leftarrow$

status(X) $=$ *undergraduate, gpa(Name)* $\geq$ *3.5.*

(2a) *pre_requisite (Course, Pre_requisite_course)* $\leftarrow$

pre_requisite (Course, Pre_requisite_course).

(2b) *pre_requisite (Course, Pre_requisite_course)* $\leftarrow$

pre_requisite (Course, Required_course), pre_requisite (Required_course, Pre_requisite_course).

Let the following generalized rules be extracted by knowledge discovery tools from EDB and stored in GDB.

(1) *All of the teaching assistants are graduate students.*

$s \in Student$ **and** $c \in Course$ **and** $c.TA = s.Sname \rightarrow s.Status =$ *"graduate"*.

(2) *Every teaching assistant has a good or excellent grade point average.*

$s \in Student$ **and** $c \in Course$ **and** $c.TA = s.Sname \rightarrow$

$s.GPA = \{$ *"good"*, *"excellent"*$\}$.

Our study on intelligent query answering mechanisms in next chapter will reference this database substantially.

CHAPTER 5

Intellegent Query Answering in Knowledge-rich Databases

We now introduce a unified framework for answering data and knowledge queries in a knowledge-rich database. The study is performed on intelligent query answering with the focus on the application of discovered knowledge, concept hierarchies, and knowledge discovery tools to intelligent query answering in database systems.

5.1 Four Basic Categories of Query Answering Mechanisms in Knowledge-Rich Databases

In a knowledge-rich database system, there may exist two kinds of queries, *data queries* and *knowledge queries* , where a **data query** is to *find concrete data stored in databases, which* *corresponds to a basic retrieval statement in a database system* ; whereas a **knowledge query** is to *find rules and other kinds of knowledge in the database, which corresponds to querying database knowledge [45] including deduction rules, integrity constraints, generalized rules and other regularities* . For example, "*retrieving all of the students who took the course CMPT-459 in 1992* " is a data query; whereas "*describing the general characteristics of those students* " is a knowledge query. Furthermore, it is often desirable to provide intelligent and assisted answers to queries *besides* (or *instead of*) direct retrieval of data and knowledge. Thus, query answering mechanisms in a knowledge-rich database can be classified based on their responses to queries into two categories: *direct query answering* and *intelligent (or cooperative) query answering* . **Direct query answering** is *a direct, simple retrieval of data or knowledge from the knowledge-rich database* ; whereas **intelligent query answering** consists of *analyzing the intent of query and providing generalized, neighborhood or associated information relevant to the query* [11]. For example, simple retrieval of the names of the students who take the designated course is direct query answering to the above data query; whereas summarizing the characteristics of those students, such as "*90% of them majored in computing science and took CMPT-359 as prerequisites* ", provides an intelligent answer to the same data query. Therefore, there are four basic combinations of queries and query answering mechanisms:

- DD (Data query - Direct answering): direct answering of data queries;
- DI (Data query - Intelligent answering): intelligent answering of data queries;
- KD (Knowledge query - Direct answering): direct answering of knowledge queries; and
- KI (Knowledge query - Intelligent answering): intelligent answering of knowledge queries.

In this chapter, query answering mechanisms are examined in each of these four categories.

5.2 Direct answering of data queries

Direct answering of data queries corresponds to direct data retrieval in knowledge-rich databases. Traditional query processing in relational and deductive databases belongs to direct answering of data queries.

Data in a knowledge-rich database are classified into primitive data and high-level data. The former are actual data stored in data relations and, if appearing in some concept hierarchies, correspond to the primitive level (i.e., leaf) nodes of the hierarchies; whereas the latter are nonprimitive data subsuming primitive ones and residing at the nonprimitive level of concept hierarchies. Correspondingly, a primitive-level query is a query whose constants involve only primitive data; whereas a high-level query is a query whose constants involve high-level data. A *primitive-level data query* can be processed directly using relational and deductive query processing techniques.

A *high-level data query* can be processed in two steps. First, a query rewriting process can be performed to rewrite the query into one or a set of equivalent *primitive-level data queries* by substituting each high-level concept in the query with a set of or a range of its subordinate primitive-level concepts by consulting concept hierarchies in the KDB. Second, each rewritten query is then fed into a relational or deductive query processor for processing. Answers should be returned at the primitive level. Presentation of answers at a nonprimitive level, when desired, is considered as a task of *intelligent* query answering and will be discussed in the next subsection. One example of a high level query is illustrated below.

Example 5.1 *To* find the graduate students born in Canada, majoring in science, and with excellent GPAs , *the query can be formulated in a syntax similar to SQL as follows.*

retrieve *Name*
from *Student*
where *Status* = *"graduate"* **and***Major* = *"science"* **and***Birth_place* = *"Canada"* **and**
GPA = *"excellent"*

Notice that "*graduate* ", "*science* " and "*excellent* " are high-level concepts which are not stored in the relation *Student* . Using the information stored in concept hierarchy H, the query can be reformulated by substituting *graduate* with {*M.S., M.A., Ph.D.* }, and *GPA* = *"excellent"* with $GPA \geq 3.5$ and $GPA \leq 4.0$, etc. The rewritten query can be answered by direct data retrieval.

5.3 Intelligent answering of data queries

Intelligent answering of data queries refers to the mechanisms which answer data queries cooperatively and intelligently. Intelligent query answering is accomplished by analyzing the intent of a query and providing some generalized, neighborhood, or associated answers. There are many ways for a data query to be answered intelligently, including generalization and summarization of answers, explanation of answers or returning intensional answers, query rewriting using associated or neighborhood information, comparison of answers with those of similar queries, etc. Several mechanisms for intelligent answering of data queries using (generalized) database knowledge are examined.

5.3.1 Analysis of the intent of a query

To answer a query intelligently, the first important step is to analyze the intent of the query, determine whether it is necessary to provide assisted answers, and if it is, what kind of assistance should be provided. Such an analysis should be based on the available or discovered knowledge about database, queries, and users. Since a large volume of knowledge may exist or can be discovered in a database, one may often find that there exist too many "intelligent" ways to associate a query with the available or discoverable database knowledge. It is crucial to have knowledge about user's background and the role that he/she plays in order to understand user's intention, avoid superfluous answers, and provide users with quality assistance.

When posing a query, different users often have quite different intentions. For example, when asking the highest monthly balance of an account, a customer and a bank manager likely have different intentions. Therefore, an important task of query intent analysis is *user modeling* , which analyzes the user's background and intention and constructs different models for different classes of users.

Several interesting methods for query intent analysis have been developed in the studies on intelligent query answering [33][34][44][11][54]. Such analyses are based on the notions of generalization, association, aggregation, concept clustering, etc. Semantic data modeling, classification of *topics of interests* , and *plan analysis and formation* are powerful techniques for query intent analysis [34][44][11][54], which can be applied to the analysis of query intent in the KDB.

Since the knowledge-rich database is constructed based on an extended (deductive) entity-relationship model together with the deductive and knowledge discovery components, the new components provide powerful support for query intent analysis. Considering an airline reservation system as an example, the method for query intent analysis in the KDB is outlined as follows.

1. **Data classification and concept clustering** : Based upon the extended entity-relationship model of the system, entities, relationships, attributes and specific conditions can be classified and clustered. For example, departure time and arrival time can be associated with time_table, airports can be clustered according to some local distance, etc. The data classification and clustering task is facilitated by the availability of concept hierarchies and knowledge discovery tools.

2. **User modeling** : Based upon user's professional position (e.g., manager, clerk, business customer, tourist, etc.), confidence level (e.g., eligibility of accessing some sensitive data), accessing history (e.g., frequent flyer, business class traveler, being interested in some particular airlines, new customer, etc.) or other related information, a user can be associated with a particular user category built in the system. The linkage between a category of users and a class of preferred concepts or entity sets is constructed by experts in the development of intelligent query answering system. With the available knowledge-rich data model and knowledge discovery components,

users can be naturally categorized into some high-level user classes (e.g., luxury, economy, or regular classes for travelers) and be associated with a set of high-level concepts (e.g., the traveler's major interests expressed at a high concept level) to assist query intent analysis.

3. **Query classification** : A query can also be classified into different categories according to the query condition and the information to be inquired. For example, queries on travel plan can be categorized into long distance travel, short distance travel, etc. according to the conditions given in the query, or categorized into general browsing, detailed examination, ticket booking according to query actions. A query class can be linked with certain user categories, generalized concept classes and transformation rules to guide appropriate intelligent query answering for particular classes of queries.

4. **Transformation rule specification** : A set of transformation rules can be specified by experts based upon user category, query category, concept hierarchies and the relationships among high-level entities, attributes, and conditions. For example, if a user is in the category of tourist and new customer, the cost could be of a major concern at flight booking, and the information about low airfares could be of major interest. Such heuristics can be specified as transformation rules to guide intelligent query answering.

Query intent analysis can be performed by systematically applying techniques of user modeling, concept classification and clustering, query classification and transformation. Further, the constructed models and transformation rules should be testified by experiments and be tuned according to their effectiveness in intelligent query answering and the feedbacks from users[21][14]. The query rewriting and answer transformation processes discussed below are directed by the results of query intent analysis.

5.3.2 Query rewriting using associated or neighborhood information

Direct data retrieval may not always find enough answers for a user. Furthermore, a user may like to know more information than the direct answers to a query for decision making. Therefore, it is often useful to provide associated or neighborhood answers to a query. Associated query answering can be performed by (1) *presenting the information about some additional attributes which are not directly inquired but are relevant to the query* ; (2) *relaxation of certain query conditions* ; and (3) *adding an alternative query which is closely related to the original one* [11].

Let the answer set be viewed as a relation table. Three mechanisms can find their corresponding relational transformations: *width-extension, height-extension* , and *table-extension* .

- **Width-extension** : The first case (addition of relevant attributes) can be viewed as extension of the width of the answer table by adding some closely related attributes to the table. For example, an inquiry on the arrival time for air-flight booking can be answered by returning also the departure time and the possible transfer time, as well.
- **Height-extension** : The second case (relaxation of certain query conditions) can be viewed as an extension of the height of the table. For example, an inquiry on the available flights for air-flight booking can be answered by relaxation of the maximum dollar restriction, flight-time restriction, airline selection restriction, etc.
- **Table-extension** : The third case (answering an alternative query) can be viewed as an extension of the answer table or a switch to a similar table. For example, an

inquiry on the available flights for air-flight booking can be answered by returning the flights to a neighborhood arrival or departure airports, or even a suggestion of other means of transportation, such as by train, ferry, or bus, depending on the distance, time and cost of the transportation, etc.

Query rewriting rewrites a query according to the intent of the query. Clearly, query intent analysis plays an important role in the selection of appropriate extensions. For example, the selection of associated additional attributes (as width-extension) should be determined by analysis of the semantics of the query and the associated attributes at the higher concept level, and the selection of relaxed constraints (as height-extension) should be based on the analysis of query semantics and the role of query constraints.

Query rewriting can be implemented by mapping query constants to an appropriate level via generalization or specialization and mapping a query to a neighborhood one by providing with additional, associated or neighborhood information. The knowledge discovery components, which specify or discover generalization, aggregation, neighborhood, or association relationships among data in the database, provide important assistance in the analysis of query intent and in the rewriting of queries into their alternatives based on hierarchical or neighborhood relationships.

Example 5.2 *Consider a query to find the names of the teaching assistants of database courses in the University database. The query can be rewritten in the following ways.*

1. *table width extension* : e.g., providing more information about the teaching assistants, such as their GPA, courses_taken, teaching experience, etc.

2. *table height extension* : e.g., providing teaching assistant information for other related

courses, such as other computing science courses.

3. *table extension* : e.g., providing other relations, such as information about research assistants and project assistants in computing science if the user is a graduate student and the time is the beginning of a new semester (a job hunting season).

Obviously, the success of a query rewriting depends on the query intent analysis and the availability of associated, generalized and neighborhood information. Such information may exist in concept hierarchies or discovered knowledge rules or can be discovered by knowledge discovery tools.

5.3.3 Answer transformation and answer explanation

Together with the rewriting of queries, the set of answers may also be transformed, explained, compared or summarized in different ways for intelligent query answering.

5.3.3.1 Generalization and summarization of answers

A database user may be interested in general description or overall statistics of the answer set to a query but not interested in the detailed answer set itself. Thus, a data query can be answered by generalization and summarization of the answer set, that is, by presenting generalized data only, a combination of generalized and primitive data, or a summarization of concrete answers (possibly together with the presentation of concrete answers) using generalized data and database statistics. Such a process belongs to **answer transformation** .

Example 5.3 *A query which inquires about the information of a student Tom Jackson in the University database can be answered as* "Tom Jackson is an undergraduate student *(a concept at a level higher than* senior *student),* born in Canada *(not mentioning the specific city and province)* in 1971 *(not mentioning the specific date)". This is meaningful if the user (such as a university administrator) is concerned of the general information but not the detailed one. Also, a query which inquires* "who have good or excellent GPAs in computing science ?" *can be answered intelligently in several ways: (1)* "100% graduate students, 55% senior students, and 25% junior students " *(general, statistical information only), (2)* "all of the graduate students and the following undergraduate students ... " *(a combination of generalized and primitive data), (3) concrete answer (student names) plus a summarization of the answers at a high level, etc.* □

By presentation of general information or associating such information with concrete answers, answers to a query can be presented in a general and concise manner, thus making the implications of the answers better understood.

An important technique for answer transformation is the *mapping between different levels of data based on concept hierarchies* . Constants in a query or answers to a query can be mapped up or down along a concept hierarchy depending on the semantics and the intent of the query. A high-level query can be rewritten into a primitive-level one by mapping the high-level data in the query to a set of primitive data using concept hierarchies. Similarly, a low-level answer set can be transformed into a high-level one by mapping a set of primitive data in the answer set to a set of corresponding high-level ones according to user's need. The interactions between query conditions and rule bodies (conditions) also need the data/constant mapping among different levels. For example, to examine whether a query is relevant to a certain generalized relation, the query can be restated at the same concept level as that in the rule.

Another important technique for answer transformation uses *lazy evaluation* , that is, providing rule bodies (conditions) without presenting the full answers set. Detailed and concrete answers are provided only by further requests. Lazy evaluation as an intelligent query answering mechanism has been studied in deductive database research [34][45][54][49]. Instead of returning the concrete answer set, the query answering mechanism instantiates a deduction rule using query constants and returns the instantiated condition (body) of the rule or a mixture of instantiated rule condition (body) and concrete data as the answers to the query. Assume that the generalized rule is also in the form of "*head* $\leftarrow$ *body*" .Lazy evaluation can be performed by returning the body of a generalized rule if the query matches the head of the rule. Besides directly using the available rules, generalized rules can also be obtained by further generalization on the portion of an intermediate generalized relation which matches the query conditions.

Furthermore, generalization and summarization of answers can be implemented by taking advantage of the available generalized information and knowledge discovery tools. If there is a corresponding generalized rule, the processor returns the instantiated body of the rule when the query matches the head of the rule. If there is a corresponding intermediate generalized relation, further generalization and summarization can be performed on the portion of intermediate generalized data which matches the query condition. Further generalization may produce a generalized rule with a summary of statistical information in terms of generalized concepts. Otherwise, when there is no corresponding generalized information, generalization is performed by first retrieving the required answer set and then performing generalization on the retrieved answer set using the knowledge discovery tools.

5.3.3.2 Answer explanation

Another intelligent query answering method is **answer explanation** , which explains the answers to a query by presentation of the associated rules, demonstration of the reasoning process, or illustration of the general information[51][23][21]. The summarization of the statistics of an answer set discussed above can also be employed as a technique for answer explanation.

The following example demonstrates that it is often necessary to provide explanations to the answers when the query condition follows or contradicts a rule or an integrity constraint.

Example 5.4 *If a query condition follows or contradicts a rule or an integrity constraint, the query can be answered by presentation of the knowledge (such as the rule) rather than primitive data. Data retrieval is necessary only if the direct presentation of primitive data is explicitly required. For example, suppose there is a generalized rule,* "all of the teaching assistants are graduate students". *The query* "find all of the undergraduate students who are teaching assistants" *can be answered by returning an empty set without accessing the extensional database. However, it is user-friendly to also give an explanation by simply presenting the rule itself. Similarly, if* "all of the teaching assistants have good or excellent GPAs" *is a generalized rule, the query* "find all of the teaching assistants whose GPAs are greater than 2.5", *may return* "all of the teaching assistants", *together with the rule. Specific teaching assistant names are presented only when the user requests for more details.* □

The process described above can be implemented by testing of the query condition against the rule for containment or contradiction. If the query passes the test, lazy evaluation can be applied rather than returning detailed answers.

5.3.3.3 Answer comparison

Queries can also be answered intelligently by **answer comparison** , which compares or contrasts the general characteristics of its answers with some similar queries. Answer comparison may involve two steps: (i) rewriting a query into a neighborhood query, and (ii) generalization, summarization and comparison of two answer sets, one to the original query and one to the neighborhood query, at a general level. The first step, rewriting a query into a neighborhood query, can be performed by query intent analysis and substitution of some query constant(s) in the original query by some similar concept(s) using the knowledge about concept hierarchies. The second step involves learning characteristic and discriminant rules using knowledge discovery techniques [24] which has been explained in Chapter 3.

Example 5.5 *In answering the query,* "find all of the graduate students with excellent GPAs "*, it is interesting to find the undergraduate students with similar characteristics or the graduate students with weaker GPAs and compare the general characteristics and statistics between these answers. Such comparisons may lead to some interesting observations, such as* "more than 50% graduate students but only about 20% undergraduate students have excellent GPAs ". □

5.4 Direct answering of knowledge queries

A knowledge query is a statement which inquires about database knowledge, including concept hierarchies, deduction rules, integrity constraints and general characteristics of a particular set of data in a database. Direct answering of knowledge queries means that a query processor receives a knowledge query and answers it directly by returning the inquired knowledge. Since IDB knowledge and concept hierarchy information are stored in the database according to our assumption, a query on such knowledge can be answered by direct retrieval. However, the situation is different at querying generalized knowledge. A generalized database (GDB) usually stores only a small, but frequently used portion of generalized knowledge. Thus, an inquiry on general knowledge should be answered by direct retrieval only if the knowledge is available in GDB. Otherwise, the knowledge should be discovered dynamically by a knowledge discovery process, which has been discussed in detail in Chapter 3. In general, a knowledge query can be answered by consulting the concept hierarchy, retrieving stored rules (if available) or triggering a discovery process.

Different syntactic specifications can be adopted to distinguish knowledge queries from data queries. A data query is to *retrieve the data elements that satisfy a condition* Φ ; whereas a knowledge query is to *describe the data elements that satisfy* Φ . Following the notion proposed by Motro and Yuan [45], data queries and knowledge queries are distinguished in syntax by starting the former with **retrieve** but the latter with **describe** . Further, to distinguish different types of knowledge being inquired, concrete keywords such as *generalized rule, deduction rule, concept hierarchy, integrity constraint* , etc. can be used after the keyword **describe** . Moreover, to query a discriminant rule which distinguishes the general characteristics of one class (*target class*) from others (*contrasting classes*), the following syntax is adopted: **describe generalized rule for** *relation* **which distinguishes** *target_class* **from** *contrasting_class* **where** *condition* Φ.

Several knowledge queries are presented in the following examples.

Example 5.6 *To find the deduction rule* award_candidate *for Canadian graduate students, a query can be formulated as below.*

describe deduction rule *award_candidate(candidate)*

where *Status(candidate) = "graduate"* **and** *Birth_place(candidate) = "Canada"*

This query can be answered by direct retrieval of deduction rules. Notice that only the condition, *Status(candidate) = "graduate"* , matches the boody of a deduction rule for *award_candidate*, which indicates that there is no further distinction on birth place in the condition for an award candidate. Thus the rule (1a) is presented as the answer to the query.

Example 5.7 *To* describe the characteristics of the graduate students in computing science who were born in Canada with excellent GPA , *the query can be formulated as below.*

describe generalized rule

for *Student*

where *Status = "graduate"* **and** *Major = "cs"* **and** *Birth_place = "Canada"*

and*GPA = "excellent"*

Notice that the query represents a high-level knowledge query since "*graduate* ", "*Canada* " and "*excellent*" are not stored as primitive data in the *University* database. The query can be answered by directly retrieving the discovered rule, if available, or by performing induction on the relevant data set [24]. □

Example 5.8 *To* distinguish the characteristics of the graduate students from undergraduate students in computing science, born in Canada with excellent GPA , *the query can be formulated as below.*

describe generalized rule for *Student*
which distinguishes *Status* = *"graduate"*
from *Status* = *"undergraduate"*
where *Major* = *"cs"* **and** *Birth_place* = *"Canada"* **and** *GPA* = *"excellent"*

Notice that the query wishes to find a discriminant rule which contrasts the general properties of the two classes. The rule can be discovered dynamically by a knowledge discovery process from primitive data or from an intermediate generalized relation as illustrated in Chapter 3. □

5.5 Intelligent answering of knowledge queries

Intelligent answering of knowledge queries means that a knowledge query is answered in an intelligent way by analyzing the intent of the query and providing generalized, neighborhood or associated information. Similar to the intelligent answering of data queries, a knowledge query can be answered in many ways, such as generalization and summarization of answers, explanation of answers, query rewriting using associated or neighborhood information, comparison of answers with those of neighborhood queries, etc. The availability of database knowledge and knowledge discovery tools enhances the power and efficiency of intelligent query answering of knowledge queries. The ideas are illustrated in the following examples.

Example 5.9 *The knowledge query of Example 5.6, which is to find the deduction rule* award_candidate *, can be answered intelligently not only by returning the* award_candidate *rule eligible to Canadian graduate students but also by (i) providing an explanation that both Canadian and foreign graduate students share the same condition for the award, (ii) returning the* award_candidate *rule eligible for undergraduate students as well, or (iii) returning other associated information, such as award name, amount, application deadlines, regulations, summary of award history, or statistical information, etc.* □

Example 5.10 *The knowledge query of Example 5.7, which is to find the characteristics of designated graduate students, can be answered intelligently by returning the characteristic rule for Canadian graduate students with excellent GPA's, together with (i) the characteristics of Canadian graduate students with different majors or weaker GPAs for comparison, or (ii) an explanation of the reasons why such students got excellent GPA's.* □

Intelligent answering of knowledge queries can involve great complexity in query intent analysis and demand sophisticated implementation techniques. Therefore, the efficient realization of the underlying mechanisms is an interesting issue for future research.

5.6 Semantic query optimization using generalized knowledge

Semantic query optimization method applies database semantics, integrity constraints and knowledge rules to query optimization [6]. Techniques have been developed for semantic query optimization based on database semantics, deduction rules and integrity constraints [6][36][38]. With the availability of concept hierarchies and generalized knowledge, new techniques can be explored to enhance the power and applicability of semantic query optimization.

A generalized rule can be treated like a deduction rule or an integrity constraint in semantic query optimization. Therefore, the techniques developed for semantic query optimization in relational and deductive databases [6][36][38] can be directly applied to knowledge-rich databases with discovered knowledge. Furthermore, with the availability of concept hierarchies and generalized knowledge, query optimization using a generalized rule can be explored for the query conditions which are subsumed by the rule body (condition) or the rule head (conclusion) at a different level of concept hierarchies. This is based on the following theorem.

Theorem 5.1 (Condition specialization and conclusion generalization *) Let a knowledge rule be in the form of*

$$p_1 \wedge p_2 \wedge ... \wedge p_i \wedge ... \wedge p_n \rightarrow q.$$

The specialization of a condition p_i to p_i' and/or the generalization of the conclusion q to q' based on that the information in concept hierarchies will not change the validity of the rule. That is,

$$p_1 \wedge p_2 \wedge ... \wedge p'_i \wedge ... \wedge p_n \rightarrow q.$$
$$p_1 \wedge p_2 \wedge ... \wedge p_i \wedge ... \wedge p_n \rightarrow q'.$$

Proof . Since p'_i is a specialization of p_i, $p'_i \rightarrow p_i$. Similarly, $q \rightarrow q'$. Based on the transitivity property of logic rules, the above rules hold. □

As a simple example for the theorem, if *the GPA of every graduate student is greater than 3.2* , then *the GPA of every M.Sc. student must be either good or excellent* .

According to Theorem 5.1 and the principles of semantic query optimization in relational and deductive databases [6][36][38], the semantic query optimization techniques for application of generalized knowledge are presented as follows.

- **Query condition subsumed by the rule body (condition)** : *If a query condition is subsumed by the body of a rule, the conclusion (head) of the rule applies* .
  For example, if there is a generalized rule: *all of the teaching assistants for 400 level courses are Ph.D. students* , then querying the status of a student who is assisting 459 will return "*Status = Ph.D.* " without searching the EDB. Further, the rule can be returned as an explanation to the answer.
- **Query condition conflicting with the rule conclusion** : *If some query conjunct(s) in a conjunctive query is subsumed by the body of a rule but some other conjunct(s) conflicts with the head (conclusion) of the rule, the answer to the query is empty* .
  For example, if there is a generalized rule: *all of the teaching assistants are graduates* , then querying the courses which an undergraduate is assisting returns an empty set

without searching EDB. Also, the rule can be presented as an explanation.

- **Query conjunct elimination** : *If a query conjunct is implied by another condition in the query, the query conjunct can be removed from the query* .
  For example, if a query contains both *S.Status = "M.Sc."* and *S.GPA > 3.0* , the second conjunct can be removed from the query since it is implied from the first one in the generalized rule.

- **Query conjunct introduction** : *If a query conjunct is subsumed by the body of a rule, the head (conclusion) of the rule can be introduced as a new conjunct to the query if such an introduction may improve search efficiency (e.g. exploration of indexing or clustering properties of databases, etc.)* .
  For example, to *find all of the foreign students majoring in science with GPA between 3.2 to 3.4* , the query is subsumed by the body (condition) of a generalized rule in GDB: *all of the foreign students majoring in science with a good GPA are graduate students* . The new conjunct, *Status = "graduate"* , can be added to the query. The search can be improved if the tuples in *Student* are grouped or partitioned according to *Status* .

- **Sharpening query condition** . *If some query conjunct(s) is subsumed by the body (condition) of a rule, and the rule head (conclusion) introduces a more selective condition than some query conjunct, then the less selective query conjunct should be replaced by the more selective one* .
  For example, if some query condition is subsumed by a generalized rule which concludes that S.GPA = "*excellent* " ($\geq$ 3.5), then a less selective query conjunct, $S.GPA \geq 2.0$, can be replaced by a more selective one.

- **Query in relevance only to generalized knowledge** . *If the conditions of a query and its inquired information are relevant only to the generalized knowledge, it can be answered by consulting the generalized knowledge only* .

  For example, a query, "*how many graduate students in Computing Sciences were born in foreign countries?* ", can be answered directly by examining the corresponding prime relation if it is stored in the GDB.

As a summary of the discussion, an algorithm is presented here, which explores semantic query optimization using generalized rules.

Algorithm 5.1 *Semantic query optimization using generalized rules.*

Input : (i) A set of generalized rules R, (ii) a set of concept hierarchies, H, and (iii) an input data query q which consists of a set of conjuncts.

Output . A possibly optimized processing plan for query q.

Method .

1. Test whether there is a conjunct c_i implied by another conjunct c_j in the conjunctive query q. If there is, remove c_i.

2. Test whether some query conjunct(s) in a conjunctive query is subsumed by the body (condition) of a rule but some other conjunct(s) conflicts with the head (conclusion) of the rule. If it is so, the answer set is empty.

3. Test whether all of the (remaining) query conjuncts are subsumed by the body (condition) of a generalized rule. If so, return the conclusion (head) of the rule as the (intensional) answer to the query.

4. Test whether some of the query conjuncts are subsumed by the body of a generalized rule. If so, examine whether the conclusion (head) of the rule may (i) sharpen a

conjunct in the query or (ii) reduce the search effort. If so, replace the conjunct by the rule head (conclusion) in case (i), and add the rule head (conclusion) as a new conjunct to the query in case (ii). □

Remark. *Algorithm 5.1 correctly performs semantic query optimization using the rules stored in GDB.*

Reasoning. Step 1 corresponds to *query conjunct elimination*, Step 2 to *query condition conflicting with rule head (conclusion)*, Step 3 to *query conclusion subsumed by rule body (condition)* and *query in relevance only to generalized knowledge*, and Step 4 to *sharpening query condition* and *query conjunct introduction.*

CHAPTER 6

CONCLUSIONS AND DISCUSSION

6.1 Conclusions

In this thesis, a framework has been presented for intelligent query answering in a knowledge-rich database composed of deductive and knowledge discovery components. A knowledge-rich data model is constructed which consists of an extended entity-relationship schema, an extensional database, an intensional database, a set of concept hierarchies, a set of generalized rules, and a set of knowledge discovery tools.

Query answering mechanisms are classified into (1) direct answering of data queries, (2) intelligent answering of data queries, (3) direct answering of knowledge queries, and (4) intelligent answering of knowledge queries. Techniques have been developed for implementation of such mechanisms using discovered knowledge and/or knowledge discovery tools, which include deduction, generalization, data summarization, rule discovery, concept clustering, query rewriting, lazy evaluation, semantic query optimization.

The availability of generalized rules, concept hierarchies and knowledge discovery tools greatly enhances the power of intelligent query answering in the following aspects.

- It expands the spectrum of knowledge queries from inquiring deduction rules to inquiring general regularity of data, such as characteristic rules, discriminant rules, data evolution regularities, etc. [24].
- It facilitates the query intent analysis since the notions of generalization, aggregation, neighborhood, similarity, etc. can be studied systematically using the generalized knowledge and concept hierarchies.
- It facilitates intelligent query answering since answers can be presented in general terms, summarized by statistical information, and compared with similar groups of data at a high level.
- The intelligent query answering can be implemented efficiently using generalized rules and knowledge discovery tools using prime relations, feature tables, semantic query optimization and other implementation techniques.

6.2 Future Research

The enhanced power of intelligent query answering leads to two problems: *superfluous "intelligent" answers* and the *risk on database security.*

The first problem indicates that one may suffer from obtaining too many superfluous "interesting" answers to a query because there are many ways for a query to be answered intelligently. Techniques should be developed to control the answer generation process in intelligent query answering. In general, one may assume that an appropriate knowledge level is associated with each user. A user usually poses queries at his/her corresponding knowledge level and expects the answers to be presented at the same level. If the contents of the answer set are not at such level, generalization or specialization should be performed on the answer set as concept level adjustment. Further, with user modeling and query intent analysis, only those answers which match the query intent and the user model will be presented. More desirably, an intelligent query answering process can be triggered or directed by interaction with users. For example, after obtaining the preliminary set of answers to a query, some following-up questions can be raised by users, such as "*in more detail*?", "*in summary*?", "*why*?", "*other options*?", "*comparing with others*?", etc. These questions indicate what kind of intelligent answers are expected. Then the corresponding intelligent query-answering mechanisms can be evoked.

The second problem indicates that with the extended power of intelligent query answering, some sensitive or confidential information could be disclosed inappropriately to someone who should not know it [47]. One technique which may enhance the database security is to associate with a user model certain kinds of constraints on accessing rights. For example, if the user is a student (easily known from the login name), the constraints on intelligent answering of his/her query in a university database will be quite different from the same query posed by a professor. Sensitive information will not be disclosed to the users who do not have appropriate access rights. However, because of the power and complexity of deduction and knowledge discovery, it is difficult to tell to what extent that accessing certain piece of information may eventually lead to the disclosure of sensitive information by a sequence of deduction and induction. Therefore, more study should be performed on ensuring database security in intelligent query answering in databases augmented with deduction and knowledge discovery components.

APPENDIX A

A TUTORIAL ON THE DBLEARN SYSTEM

The DBLEARN system is designed to discover the database knowledge, including characteristic rules and discriminant rules, from a relational database supported by the SyBase system. Recall that A *characteristic rule* is an assertion which characterizes the concepts satisfied by all of the data stored in the database, and a *discriminant rule* is an assertion which discriminates the concepts of one class from other class(es). The DBLEARN system is implemented in C and runs under Unix on a Sun workstation.

A.1 The Architecture of the DBLEARN System

The architecture of the DBLEARN system was shown in the Figure 3.3, which consists of user-interface, learning program, database data and concept hierarchies. In the learning process, the DBLEARN system first accepts the user's request through the user-interface.

Based on the specified learning task, the DBLEARN system obtains the relevant data from a database and the relevant concept hierarchies from a file. The learning program performs attribute-oriented induction to extract generalized rules. After learning is performed, the learning result is reported to the user through the user-interface.

A.2 The Description of the Learning Programs

The DBLEARN system consists of six programs which accomplish different functions of a learning process.

A.2.1 learn.h and dblearn.h

The file "*learn.h*" is a library of the DBLEARN system which contains the declarations of data structures and constant variables. It is included in the program *parse.c*. The file "*dblearn.h*" is similar to the file *learn.h* except the variables defined in *dblearn.h* are external variables. "*dblearn.h*" is included in the programs *fetch.c* and *learn.c*.

A.2.2 lex.c

The program "*lex.c*" is lexical analyzer which uses the program LEX (a lexical analyzer generator) supported by the UNIX system. The program "*lex.c*" contains lexical specifications of the program. The generated program can recognize the learning request in an input stream and partition the input stream into lexical units which matches the expressions of the parsing tree. The command used to compile the *lex.c* program is "*lex lex.c*" which will generate a program named *lex.yy.c*.

A.2.3 parse.c

The program "*parse.c*" is a syntax analyzer which is implemented using the program YACC (a compiler-compiler) supported by the UNIX. A collection of grammar rules is specified in the program. Each rule describes an allowable structure and the corresponding action(s). The program accepts the "token" generated by the program *lex.c* and invokes a certain action when the token matches a specified structure.

The compilation of the *parse.c* program has two steps. The command "*yacc parse.c*" will first generate a file named *y.tab.c*, then the command "*cc y.tab.c*" will generate the executable code.

A.2.4 fetch.c

The program "*fetch.c*" is written using C supported by the SyBase system. It collects task-relevant data based on the user's learning request and passes the data to the learning program *learn.c*.

The compilation of the *fetch.c* program involves some library routines provided by the SyBase system. The command used to compile the *lex.c* program is

```
cc -I/usr/local/Sybase/include -c fetch.c /usr/local/Sybase/lib/libsybdb.a -lm.
```

A.2.5 learn.c

This program is a C program and performs the induction process. The learning program consists of two modules, LCHR and LDIR, which learn a characteristic rule and a discriminant rule, respectively. Either of these two modules will be invoked based on the user's learning request. The program applies an attribute-oriented induction method which performs generalization on the selected data attribute by attribute. The generalization strategies used in the learning process are the removal of nongeneralizable attributes and the ascension along the concept hierarchies. The learning process can be viewed as a sequence of table transformations, from a less generalized relation to a more generalized one. The generalization is controlled by a user-specified threshold value. The output of this program is a generalized relation that contains a small number ($\leq$ threshold value) of tuples. The learning results is also presented in a corresponding logic form.

This program can be compiled using the command "*cc -c learn.c*" which will generate an object code *learn.o*.

A.2.6 adjust.c

This program performs some miscellaneous functions of DBLEARN, such as the refinement of concept hierarchies and the display of a particular concept hierarchy. The display of a concept hierarchy is realized by calling some routines of HOOPS. This program can be compiled using the command "*cc -c adjust.c*" which will generate an object code *adjust.o*.

All the commands for compilation of the above four programs have been collected in a file named makefile shown as follows. The programs should be recompiled after any

```
HOOPS_LIBS = -lhoops -lX11 -lpixrect -lsuntool -lsunwindow -lm
SYBASE = /usr/local2/Sybase

SOURCES = adjust.c parse.c fetch.c learn.c
OBJECTS = y.tab.o adjust.o fetch.o learn.o
EXECUTABLES = dblearn


all: lex.yy.c y.tab.c dblearn

lex.yy.c : lex.c
           lex lex.c

y.tab.c: lex.yy.c parse.c learn.h
         yacc parse.c

y.tab.o: y.tab.c
         cc -g -c y.tab.c

fetch.o: fetch.c dblearn.h
         cc -I$SYBASE/include -g -c fetch.c $SYBASE/lib/libsybdb.a -lm

learn.o: learn.c dblearn.h
         cc -g -c learn.c

adjust.o: adjust.c dblearn.h
          cc -g -c adjust.c dblearn: $OBJECTS
          cc -I$SYBASE/include -g -o $EXECUTABLES $OBJECTS $SYBASE/lib/libsybdb.a
          $HOOPS_LIBS
```

Figure A.1: Makefile

modification, which can be done by simply typing the command "*make*".

A.3 The Specification of a Learning Request

A user-friendly interface is built in the DBLEARN system, by which users can specify the learning task, the threshold value, the relations and the attributes relevant to the learning task, the concept to be learned(target class) and the concept to be compared (contrasting class).

A.3.1 Getting Started

Type at your Unix prompt the command "**dblearn** *HierarchyName*" and on your screen, you will then see the prompt:

DBLEARN 1>

A *HierarchyName* is a directory under which the concept hierarchies you are interested in are stored. If you don't specify the *HierarchyName* here, you have to specify it after you get into DBLEARN by typing:

DBLEARN 1> **use** *HierarchyName*

A.3.2 Basic Structure

The basic structure of an DBLEARN expression consists of seven clauses: **learn**, **for**, **from**, **where**, **in relevance to**, **using**, and **in contrast to**.

- The **learn** clause specifies the learning task. Currently, only characteristic rule and discriminant rule can be specified.
- The **for** clause can specify the name of the target class. A string should follow the reserved word *for* and will be printed out in the final result as the name of the extracted rule.
- The **from** clause lists the relations from which the task-relevant data can be retrieved.

- The **where** clause consists of a predicate involving attributes of the relations that appears in the **in relevance to** clause. If the **where** clause is omitted, all of the tuples in the relations specified in the **from** clause are retrieved as the task-relevant data.

- The **in relevance to** clause is used to list the attributes desired in the generalized rule. If it is omitted, all of the attributes in all the relations appearing in the **from** clause will be involved in the learning process.

- The **using** clause could be used in the following three ways.

    1. The **using attribute threshold** clause specifies the desired threshold value. The number of distinct values in each attribute should not be greater than the attribute threshold value. If it is omitted , a default value, **5**, will be chosen.
    2. The **using table threshold** clause specifies the desired threshold value. The number of tuples in the final generalized relation should not be greater than the table threshold value. If it is omitted, a default value, **10**, will be chosen.
    3. The **using hierarchy** clause specifies the files which contain the required concept hierarchy information. If it is omitted, the system will invoke a default file named *concept* for the required concept hierarchy information.

- The **in contrast to** clause is used to specify the name of the contrasting class.

A typical DBLEARN query for *learning characteristic rule* has the form:

learn characteristic rule
for *Target_Class_Name*
from $r_1, r_2, ..., r_n$
where P

in relevance to $A_1, A_2, ..., A_m$

using attribute threshold N_1

using table threshold N_2

using hierarchy $H_1, H_2, ..., H_k$

Each r_i represents a relation. P is a predicate. Each A_i represents an attribute. N_1 and N_2 are the attribute threshold value and table threshold value respectively. Each H_i represents a file which contains some of the required concept hierarchy information.

A typical DBLEARN query for *learning discriminant rule* has the form:

learn discriminant rule

for *Target_Class_Name*

where P_1

in contrast to *Contrasting_Class_Name*

where P_2

from $r_1, r_2, ..., r_n$

where P_3

in relevance to $A_1, A_2, ..., A_m$

using attribute threshold N_1

using table threshold N_2

using hierarchy $H_1, H_2, ..., H_k$

The first **where** clause is used to define what the target class is. The second **where** clause specifies the contrasting class. The third **where** clause specifies the common restrictions shared by the target class and the contrasting class. The third **where** clause could be omitted by adding P_3 with P_1 and P_2 respectively.

A.3.3 Examples

The following are some samples of learning request.

Example A.1 *The learning task "learning the characteristic rule for the operating grants awarded to computer science discipline from relation* award, organization, *and* grant_type *in relevance to attributes* amount *and* province, *with a table threshold value equal to 18, and using the concept hierarchy file* disc, amount, prov, *and* grant_type*" can be specified as follows.*

DBLEARN 1> **learn characteristic rule**

DBLEARN 2> **for** "CS_Op_Grants"

DBLEARN 3> **from** award A, organization O, grant_type G

DBLEARN 4> **where** O.org_code = A.org_code **and** G.grant_order = "Operating_Grants"
and A.grant_code = G.grant_code **and** A.disc_code = "Computer"

DBLEARN 5> **in relevance to** amount, province, prop(votes), prop(amount)

DBLEARN 6> **using table threshold** 18

DBLEARN 7> **using hierarchy** disc, amount, prov, grant_type

DBLEARN 8> **go**

Notice that *prop(attribute)* is a built-in function which returns the percentage of the summation of the *attribute* value in the generalized tuples divided by the summation of the same *attribute* value in the whole generalized relation. The type of the *attribute* must be "int" or "float". *Votes* is a special attribute which registers the number of tuples in the original relation which are generalized to one tuple in the final generalized relation. *Prop(votes)* returns the percentage of tuples covered by a generalized tuple in the final relation.

A default attribute threshold value, 5, is used in this query. Notice that in this example, "Computer" is a high level concept for attribute *disc_code* and DBLEARN can translate it into the corresponding primitive level concepts by consulting the corresponding concept hierarchy information stored in the file *disc*. Finally, you have to type "*go*" on a line by itself. It is the command terminator in DBLEARN, and lets DBLEARN know that you are done typing and ready for your command to be executed.

By performing attribute-oriented induction, DBLEARN first presents the prime relation and then gives users two alternatives, which are performing further generalization on the prime relation and extracting feature table for a particular attribute respectively. Based on users' selection, corresponding action will be taken to generate final results. One possible output of example A.1 is given as follows.

```
************************************************************************
*                         The Prime Relation                           *
************************************************************************
------------------------------------------------------------------------
|  amount        |    province          |  prop(votes)    |  prop(amount) |
------------------------------------------------------------------------
| 40Ks-60Ks      |    B.C.              |      1.23%      |        7.62%  |
| 20Ks-40Ks      |    B.C.              |      5.35%      |        6.20%  |
| 0-20Ks      | B.C.         |    7.41%  |     7.64% |
| 60Ks-       | B.C.         |    0.21%  |     4.15% |
| 20Ks-40Ks   | Prairies     |    5.35%  |     6.47% |
| 0-20Ks      | Prairies     |    8.23%  |     3.76% |
| 40Ks-60Ks   | Prairies     |    0.21%  |     1.88% |
| 60Ks-       | Prairies     |    0.41%  |    10.62% |
| 0-20Ks      | Ont.         |   24.49%  |     6.21% |
| 20Ks-40Ks   | Ont.         |   12.76%  |     4.32% |
| 40Ks-60Ks   | Ont.         |    5.14%  |     5.36% |
| 60Ks-       | Ont.         |    1.23%  |    11.03% |
| 0-20Ks      | Queb.        |   13.79%  |     2.83% |
| 20Ks-40Ks   | Queb.        |    5.14%  |     9.99% |
| 60Ks-       | Queb.        |    0.21%  |     4.20% |
| 40Ks-60Ks   | Queb.        |    1.03%  |     2.74% |
| 0-20Ks      | Maritime     |    6.79%  |     2.61% |
| 20Ks-40Ks   | Maritime     |    1.03%  |     2.36% |
---------------------------------------------------------------------------


[1]. Perform further generalization [2]. Extract feature table

Selection:   2


Available attributes:


[1]. amount

[2]. province

Selection:   1
```

```
**********************************************************************
*                        Amount Feature Table                        *
**********************************************************************
----------------------------------------------------------------------
| Amount     |                  Province                      |vote  |
|            | B.C.    Prairies   Ont.    Queb.    Maritime   |      |
----------------------------------------------------------------------
| 40Ks-60Ks  |  6         1        25       5          0      |  37  |
| 20Ks-40Ks  | 26        26        62      25          5      | 144  |
| 0-20Ks     | 36        40       119      67         33      | 295  |
| 60Ks-      |  1         2         6       1          0      |  10  |
----------------------------------------------------------------------
| Total      | 69        69       212      98         38      | 486  |
----------------------------------------------------------------------
```

In this example, the user prefers to generating a feature table for attribute *amount.*

Example A.2 *Similarly, the following learning request learns the discriminant rule that can distinguish the computer science grants awarded to Ontario from those awarded to Newfoundland.*

DBLEARN 1> **learn discriminant rule**

DBLEARN 2> **for** "Ontario_CS_Grants"

DBLEARN 3> **where** O.province = "Ontario"

DBLEARN 4> **in contrast to** "Newfoundland_CS_Grants"

DBLEARN 5> **where** O.province = “Newfoundland”

DBLEARN 6> **from** award A, organization O, grant_type G

DBLEARN 7> **where** A.grant_code = G.grant_code **and** A.org_code = O.org_code **and** A.disc_code = “Computer”

DBLEARN 8> **in relevance to** disc_code, amount, grant_order

DBLEARN 9> **go**

Notice that both attribute and table threshold value are default ones . All the concept hierarchy information required is stored in a default file *concept*. Generalization is performed synchronously in both the target class and the contrasting class. The prime relation for both classes is shown first. Overlapping tuples are marked by “*”. After removing overlapping tuples from the target class, the final generalized relation is generated. The following is the output of DBLEARN for example A.2.

```
**********************************************************************
*                         The Prime Relation                         *
**********************************************************************
----------------------------------------------------------------------
| Learning Concept |disc_code| grant_order      |  amount   |votes  mark |
----------------------------------------------------------------------
|                  |Computer |Operating_Grants|0-20Ks     |119      *   |
|                  |Computer |Operating_Grants|20Ks-40Ks  | 62          |
|                  |Computer |Other           |0-20Ks     | 10      *   |
|                  |Computer |Other           |20Ks-40Ks  | 10      *   |
|Ontario_CS_Grants |Computer |Operating_Grants|40Ks-60Ks  | 25          |
|                  |Computer |Other           |60Ks-      |  7          |
|                 |Computer |Other          |40Ks-60Ks |  5          |
|                 |Computer |Strategic_Grants|60Ks-    |  8          |
|                 |Computer |Operating_Grants|60Ks-    |  6          |
|                 |Computer |Strategic_Grants|40Ks-60Ks |  1          |
-----------------------------------------------------------------
|                 |Computer |Operating_Grants|0-20Ks   |  9        * |
|Newfoundland_CS  |Computer |Other          |0-20Ks    |  1        * |
|Grants           |Computer |Other          |20Ks-40Ks |  1        * |
-----------------------------------------------------------------
*****************************************************************
*                 The Final Generalized Relation                *
*****************************************************************
-----------------------------------------------------------------
| disc_code     |    grant_order     | amount    | votes    mark |
-----------------------------------------------------------------
| Computer      |  Operating_Grants  | 20Ks-40Ks |   62         |
| Computer      |  Operating_Grants  | 40Ks-60Ks |   25         |
| Computer      |  Other             | 60Ks-     |    7         |
| Computer      |  Other             | 40Ks-60Ks |    5         |
| Computer      |  Strategic_Grants  | 60Ks-     |    8         |
| Computer      |  Operating_Grants  | 60Ks-     |    6         |
| Computer      |  Strategic_Grants  | 40Ks-60Ks |    1         |
-----------------------------------------------------------------
```

A.4 Miscellaneous functions

Currently, DBLEARN provides a limited number of miscellaneous functions.

- **set demo** *1*

  Some intermediate results will be displayed. Users may get some detailed views how DBLEARN works.

- **set demo** *0*

  Only final rules will be given.

- **print schema from** *Relation_Name*

  Print out the schema of a relation.

- **display** *Attribute_Name* **in** *File_Name*

  Display the concept hierarchy information(tree structure) of an attribute stored in the file *File_Name.*

- **adjust hierarchy** *File_Name* : *Attribute_Name*
  based on relation *Relation_Name*

  The concept hierarchy of the attribute *Attribute_Name* stored in the file *File_Name* can be refined dynamically based on the statistics of data distribution in the relation *Relation_Name.*

- **help**

  On-line manual is given.

•• **quit**

Quit from DBLEARN.

APPENDIX B

Program Listing for Two Major Procedures

The **DBLEARN** source program is written in C, assisted by UNIX software packages LEX and YACC. The whole code is about 5,000 lines of C program. To save the space of printing, only two major source programs: (1) *parse.c*, the **DBLEARN** grammar specifications,(2) *learn.c*, the implementation of attribute-oriented induction algorithm are listed here.

Parse.c

```
/*************************************************************************/
/* The program ``parse.c'' is a syntax analyzer which is implemented     */
/* using the program YACC (a compiler-compiler) supported by the UNIX. A  */
/* collection of grammar rules is specified in the program. Each rule     */
/* describes an allowable structure and the corresponding action(s). The     */
/* program accepts the ``token'' generated by the program lex.c              */
/* and invokes a certain action when the token matches a specified structure.*/
/*****************************************************************************/

#include "learn.h"
%token  tID tINTVAL tSTRING tLEARN tCHAR tDISC tRULE tFOR tFROM
        tWITH tTHRESHOLD tGO tCOLON tDOT tEQ tCOMMA tLEFT tRIGHT
        tSM tLG tNOEQ tWHERE tFLOAT tIN tRELAT tRELEV tTO tUSING
        tHIERARCHY tCONTRAST tDIST tRETR tCREATE tSUPER tCLASS
        tVALUE tSTEP tUPDATE tSET tALIAS tDISCOVER tCLASSIFY
        tSCHEMA tINHERI tAND tOR tDISPLAY tPROP tSIGN tTABLE tATTR
        tDEMO tADJUST tBASED tON tPRINT tSCHEMA tSELECT tHELP tUSE
        tDELETE
DBLEARN    : {init(); remove_file(); } selection {remove_file(); return; }

selection  : tLEARN rule_type {int_q = 0; }
           | tDIST item
           | tCREATE hierar tGO
           | tUPDATE {printf("Sorry, this function is not available now!\n");
             return; }
           | tSET alias tGO
           | tDISCOVER {printf("Sorry, this function is not available now!\n");
             return; } classify {getinfo ("classify", "source");
             getinfo (relation_str, "source");
             getinfo (pos_where, "source"); getinfo (hie_str, "source");
             if (strlen(table_thres) == 0) strcpy(table_thres, "threshold 5");
```

```
  getinfo (table_thres, "source");
  if (strlen(attr_thres) == 0) strcpy(attr_thres, "attr_threshold 5");
  getinfo(attr_thres, "attr_threshold 5"); }
| tDISPLAY tID tBASED tON tID tGO  {read_bias_2($5);
  display_concept_2($2); }
| tSET tDEMO tINTVAL {demo = atoi($3);
  if (demo) {printf("\n\n*************************************\n");
  printf("*       DBLearn learning request     *\n");
  printf("*************************************\n\n"); }}
| tADJUST tHIERARCHY tID tCOLON tID tBASED tON tRELAT tID
  {strcpy(attr_str, "attribute "); strcat(attr_str, $5);
  getinfo (attr_str, "source");
  strcpy(relation_str, "relation "); strcat(relation_str, $9);
  getinfo (relation_str, "source"); strcpy(hie_str, "hierarchy ");
  strcat(hie_str, $3); getinfo (hie_str, "source");}
  tGO {adjust_hierarchy($5);}
| tPRINT tSCHEMA {init_str(str); strcpy(pos_where, "positive ");}
  relation where {strcat(pos_where, str);
  getinfo (relation_str, "source");
  getinfo (pos_where, "source"); } tGO {print_schema_table(); }
| tSELECT {int_q = 1; strcpy(attr_str, "attribute ");
  strcpy(relation_str, "relation ");} data_query tGO
  {getinfo ("data_query", "source"); getinfo (attr_str, "source");
  getinfo (relation_str, "source"); strcpy(data_query, "positive ");
  strcat(data_query, str); getinfo (data_query, "source");
  strcpy(data_query, "prop "); strcat(data_query, prop_str);
  getinfo(data_query, "source"); int_query(); }
```

```
           | tHELP {system("more /home/dk6/yhuang/Proj/prog/on-line"); }
           | tUSE tID tGO {strcpy(dir_str, $2); }
           | tCREATE tID tGO {create_db($2); }
           | tDELETE tID tGO {delete_db($2); }
           |

data_query : attr tFROM table_list where hier_thre

rule_type  : charact_rule tGO {getinfo ("charact_rule", "source");
             getinfo(attr_str, "source");
             getinfo(relation_str, "source"); getinfo(pos_where, "source");
             getinfo(hie_str, "source");
             if (strlen(table_thres) <= 0)
             strcpy(table_thres, "table_threshold 10");
             getinfo(table_thres, "source");
             if (strlen(attr_thres) <= 0)
             strcpy(attr_thres, "attr_threshold 5");
             getinfo (attr_thres, "source"); getinfo(title_str, "source");
             getinfo(prop_str, "source"); learn_charact_rule(); }
           | discrim_rule tGO {getinfo ("discrim_rule", "source");
             getinfo (attr_str, "source"); getinfo(relation_str, "source");
             getinfo(pos_where, "source"); getinfo(neg_where, "source");
             getinfo(hie_str, "source");
             if (strlen(table_thres) == 0)
             strcpy(table_thres, "table_threshold 10");
             getinfo(table_thres, "source");
             if (strlen(attr_thres) <= 0)
```

```
          strcpy(attr_thres, "attr_threshold 5");
          getinfo(attr_thres, "source"); getinfo(title_str, "source");
          getinfo(prop_str, "source"); learn_class_rule(); }
        | inherit_rule tGO {getinfo ("inherit_rule", "source");
          getinfo (attr_str, "source"); getinfo(relation_str, "source");
          getinfo(pos_where, "source"); getinfo(hie_str, "source");
          if (strlen(table_thres) == 0)
          strcpy(table_thres, "table_threshold 10");
          getinfo(table_thres, "source");
          if (strlen(attr_thres) == 0) strcpy(attr_thres, "attr_threshold 5");
          getinfo(attr_thres, "source"); getinfo(title_str, "source");
          getinfo(prop_str, "source"); }

charact_rule:tCHAR tRULE tFOR {init_str(str); strcpy(title_str, "title ");
          strcpy(pos_where, "positive "); } name
          {if (strlen(str) > 0) {strcat(pos_where, " ( ");
          strcat(pos_where, str); strcat(pos_where, " ) ");
          strcat(title_str, str); }}
          relation where {if (strlen(pos_where) > 9)
          strcat(pos_where, " AND ");
          strcat(pos_where, str); } attr_list hier_thre

discrim_rule:tDISC tRULE tFOR {init_str(str); strcpy(title_str, "title ");
          strcpy(pos_where, "positive "); } name {if (strlen(str) > 0)
          {strcat(pos_where, " ( "); strcat(pos_where, str);
          strcat(pos_where, " ) "); strcat(title_str, str); }}
          where {if (strlen(pos_where) > 9) strcat(pos_where, " AND ");
```

```
                strcat(pos_where, str); }
                tIN tCONTRAST tTO {init_str(str); strcpy(neg_where, "negative ");
                strcat(title_str, " vs "); } name {if (strlen(str) > 0)
                {strcat(neg_where, " ( ");
                strcat(neg_where, str); strcat(neg_where, " ) ");
                strcat(title_str, str); }}
                where {if ((strlen(neg_where) <= 9) && (strlen(str) <= 0))
                {printf("No contrasting class specified! \n"); return; }
                if ((strlen(neg_where) > 9) && (strlen(str) > 0))
                {strcat(neg_where, " AND "); strcat(neg_where, str); }
                else if (strlen(str) > 0) strcat(neg_where, str); }
                relation where {if (strlen(str) > 0)
                { if (strlen(pos_where) > 9)  strcat(pos_where, " AND ");
                strcat(pos_where, str); }
                if (strlen(str) > 0) { if (strlen(neg_where) > 9)
                strcat(neg_where, " AND ");
                strcat(neg_where, str); }} attr_list hier_thre

inherit_rule:tHIERARCHY tINHERI tRULE tFOR tID {init_str(str);
                strcat(str, "attribute ");
                strcat(str, $5); getinfo (str, "source");} relation where
                attr_list hier_thre

item        : {init_str(str); strcpy(pos_where, "positive "); } cond
                {strcat(pos_where, " ( "); strcat(pos_where, str);
                strcat(pos_where, " ) ");} relation where {strcat(pos_where, str);
                tFROM {init_str(str); strcpy (neg_where, "negative "); } cond
                {strcat(neg_where, str); }
                relation where {strcat(neg_where, str); } attr_list


hierar      : tHIERARCHY tFOR tID tSUPER tCLASS tID
                {strcpy(title_str, "concept");
                init_str(attr_str); if (check_name($3, "hie_table") == 0)
                strcpy(attr_str, $3);
                else {printf("\n concept hierarchy for %s alread exists...\n", $3);
                return;} }
            | tHIERARCHY tFOR tID tSUPER tCLASS tID
                {strcpy(title_str, "concept");
                init_str(buf1); strcat(buf1, $3); init_str(attr_str);
                if (check_name($3, "hie_table") == 0)  strcpy(attr_str, $3);
                else {printf("\n concept hierarchy for %s alread exists...\n", $3);
                return;} }
                class {if (strlen(attr_str) > 0) getinfo(attr_str, "hie_table"); }
            | tHIERARCHY tID tFOR tID tSUPER tCLASS tID  {strcpy(title_str, $2);
                init_str(attr_str);
                if (check_name($4, "hie_table") == 0)  strcpy(attr_str, $3);
                else {printf("\n concept hierarchy for %s alread exists...\n", $4);
                return;} }
            | tHIERARCHY tID tFOR tID tSUPER tCLASS tID  {strcpy(title_str, $2);
                init_str(buf1); strcat(buf1, $4); init_str(attr_str);
                if (check_name($4, "hie_table") == 0)  strcpy(attr_str, $4);
                else {printf("\n concept hierarchy for %s already  exists...\n", $4);
                return; } }
                class {if (strlen(attr_str) > 0) getinfo(attr_str, "hie_table"); }
class       : class tLEFT tCLASS tID {init_str(buf2); strcat(buf2, $4); }
              tVALUE value tRIGHT
            | tLEFT tCLASS tID {init_str(buf2); strcat(buf2, $3); } tVALUE
              value tRIGHT

value       : value tCOMMA tSTRING {init_str(str); strcat(str, $3);
              remove_quote(str);
              strcat(str, " "); strcat(str, buf2); strcat(str, " ");
              strcat(str, buf1); getinfo (str, title_str); }
            | value tCOMMA digit {strcat(str, " "); strcat(str, buf2);
              strcat(str, " "); strcat(str, buf1); getinfo (str, title_str); }
            | tSTRING {init_str(str); strcat(str, $1); remove_quote(str);
              strcat(str, " "); strcat(str, buf2); strcat(str, " ");
              strcat(str, buf1);
              getinfo (str, title_str); }
            | digit {strcat(str, " "); strcat(str, buf2); strcat(str, " ");
              strcat(str, buf1); getinfo (str, title_str); }

digit       : tINTVAL {strcpy(str, $1); }
            | tFLOAT  {strcpy(str, $1); }


alias       : tALIAS tID tFOR tID {if (check_name($4, "hie_table") == 0)
              { printf("\n concept hierarchy for %s does not exist... \n", $4);} }

classify    : tCLASSIFY tSCHEMA tFOR {init_str(str);
```

```
            strcat(pos_where, "positive "); }
            cond {strcat(pos_where, str); } relation hier_thre


name       : cond
           | tSTRING {strcat(title_str, $1); }


relation   : tFROM {strcpy(relation_str, "relation "); } table_list


table_list : table_list tCOMMA tID {strcat(relation_str, ",");
             strcat(relation_str, $3); }
           | table_list tCOMMA tID tID {strcat(relation_str, ",");
             strcat(relation_str, $3);
             strcat(relation_str, " "); strcat(relation_str, $4); }
           | tID tID {strcat(relation_str, $1); strcat(relation_str, " ");
             strcat(relation_str, $2); }
           | tID {strcat(relation_str, $1); }


attr_list  : tIN tRELEV tTO {strcpy (attr_str, "attribute ");
             strcpy (prop_str, "prop "); } attr
           |


attr       : attr tCOMMA {strcat(attr_str, ","); } attr_type
           | attr_type


attr_type  : {init_str(str); } attr_name {strcat(attr_str, str);
             strcat(attr_str, " "); }
           | tPROP tLEFT tID tRIGHT {strcat(prop_str, $3);
```

```
              strcat(prop_str, " ");
              if (attr_str[strlen(attr_str)-1] == ',')
              attr_str[strlen(attr_str)-1] = ' '; }


hier_thre   : tUSING tHIERARCHY {strcpy(hie_str, "hierarchy " ); }
              hierarchy hier_thre
            | tUSING tTABLE tTHRESHOLD tINTVAL
              {strcpy(table_thres, "table_threshold ");
              strcat(table_thres, $4); } hier_thre
            | tUSING tATTR tTHRESHOLD tINTVAL
              {strcpy(attr_thres, "attr_threshold ");
              strcat(attr_thres, $4); } hier_thre
            |

hierarchy   : hierarchy tCOMMA tID {strcat(hie_str, ", "); strcat(hie_str, $3);}
            | tID {strcat(hie_str, $1); }

concept     : tID tIN tID tGO {init_str(str); strcat(str, $1);
              display_concept(str, $3); }

where       : tWHERE {init_str(str); strcat(str, " ( "); } cond
              {strcat(str, " ) "); }
            | {init_str(str); }

cond        : cond1
            | tLEFT {strcat(str, " ( "); } cond1 tRIGHT {strcat(str, " ) "); }
```

```
cond1        : cond tAND {strcat(str, " AND "); } cond
             | cond tOR {strcat(str, " OR "); } cond
             | condition

condition    : attr_name tEQ {strcat(str, " = "); } attr_name
             | attr_name tSM {strcat(str, " < "); } attr_name
             | attr_name tLG {strcat(str, " > "); } attr_name
             | attr_name tEQ tINTVAL {strcat(str, " = "); strcat(str, $3); }
             | attr_name tSM tINTVAL {strcat(str, " < "); strcat(str, $3); }
             | attr_name tLG tINTVAL {strcat(str, " > "); strcat(str, $3); }
             | attr_name tEQ tFLOAT {strcat(str, " = "); strcat(str, $3); }
             | attr_name tSM tFLOAT {strcat(str, " < "); strcat(str, $3); }
             | attr_name tLG tFLOAT {strcat(str, " > "); strcat(str, $3); }
             | attr_name tEQ tSTRING {strcat(str, " = "); strcat(str, $3); }

attr_name    : tID tDOT tID {strcat(str, $1); strcat(str, "."); strcat(str, $3);}
             | tID {strcat(str, $1); }
```

Learn.c

```
/********************************************************************************/
/* This program is a C program and performs the induction process. The         */
/* learning program consists of two major procedures, learn_charact_rule and */
/* learn_class_rule, which learn a characteristic rule and a discriminant     */
/* rule, respectively. Either of these two modules will be invoked based on */
/* the user's learning request.                                               */
/********************************************************************************/

#include "lex.yy.c"
#include "dblearn.h"
```

```
/******************************************************************************/
/*  Procedure: main                                                           */
/*  Parameter: Directory in which a perticular concept hierarchy is stored    */
/*  Function : Main routine                                                   */
/******************************************************************************/

main(argc, argv)
int argc;
char **argv;
{
    int i;

    if (argc > 1) strcpy(dir_str, argv[1]);
    lineno = 0;

    remove_file();
    newline();
    while (1)
    {
      x = yyparse();
      lineno = 0;
    }

}/*main*/

/******************************************************************************/
/*  Procedure: learn_charact_rule                                             */
/*  Parameter: None                                                           */
/*  Function : Learning process for characteristic rule                       */
/******************************************************************************/

learn_charact_rule ()
{
    /* only positive tuples are selected for learning characteristic rule */
    /* the exptype is set to 1 */
    int exptype = 1, ind;
    int i = 0;
    int debug = 0;
    char attr_name[MAXSTR];
    char line[MAXLINE], word[MAXSTR];
    if (check_dir() == 0) return(0);
    fp = fopen ("source", "r");
    if (fp == NULL)
    {
        printf ("unable to read file %s\n", "source");
        return (-1);
    }
    while (fgets (line, MAXLINE, fp) != NULL)
    {
        get_one_value(line, 0, word);
        if (strcmp(word, "table_threshold") == 0) {
            get_one_value(line, 15, word);
            max_tuple = atoi(word);
        }
        if (strcmp(word, "attr_threshold") == 0) {
            get_one_value(line, 14, word);
            max_value = atoi(word);
        }
    }
    fclose(fp);

    fetch();
    bias_table.index = bias_table.index - schema_table.attr_num;

    max_num = pos_tuple.index;
    if (debug)
        print_pos (1);

    char_tuple_reduction();
}

/*****************************************************************************/
/*  Procedure: learn_class_rule                                              */
/*  Parameter: None                                                          */
/*  Function : Learning process for discriminant rule                        */
/*****************************************************************************/

learn_class_rule()
{
    /* both positive tuples and negative tuples are selected for learning */
    /* discriminant rule, the exp_type is set to -1 */
    int exptype = -1;
    int i = 0;
    int debug = 0;
    char attr_name[MAXSTR];
    char pos_target[MAXSTR];
    char neg_target[MAXSTR];
    char line[MAXLINE], word[MAXSTR];

    if (check_dir() == 0) return(0);
    fp = fopen ("source", "r");
    if (fp == NULL)
    {
        printf ("unable to read file %s\n", "source");
        return (-1);
    }
    while (fgets (line, MAXLINE, fp) != NULL)
    {
        get_one_value(line, 0, word);
        if (strcmp(word, "table_threshold") == 0) {
            get_one_value(line, 15, word);
            max_tuple = atoi(word);
        }
        if (strcmp(word, "attr_threshold") == 0) {
            get_one_value(line, 14, word);
            max_value = atoi(word);
        }
    }
    fclose(fp);

    fetch();

    max_num = pos_tuple.index;
    if (debug)
        print_pos (2);
    class_tuple_reduction();
}

/*****************************************************************************/
/*  Procedure: char_tuple_reduction                                          */
/*  Parameter: None                                                          */
/*  Function : Induction process for learning characteristic rules           */
/*****************************************************************************/
```

```
char_tuple_reduction()
{
    int i = 0, j = 0, select;
    int debug = 0;
    int attr_num = schema_table.attr_num;
    int rule_type = 1;
    char attr_name_array[MAXATTR][MAXSTR];
    struct relation temp_rel;

    if (debug) printf ("threshold %d\n", max_tuple);

    for (i = 0; i < attr_num; i++)
        strcpy (attr_name_array[i], schema_table.attr[i].attr_name);

    /* for each attribute, if the number of distinct values is greater
       than the threshold, all the values in this attribute should
       be generalized */

    for (i = 0; i < attr_num; i++)
        if (distinct_val(attr_name_array[i],max_value) == 1) {
            generalize(attr_name_array[i], rule_type);
        }

    /* if the number of tuples in the generalized relation is greater
       than the threshold, further generalization on some selected
       attribute(s) should be performed */

    remove_same_1(1);

    if (pos_tuple.index > max_tuple)
    {
        /* when the size of table has been reduced to max_tuple * 2,*/
        /* check the noise data and remove it */
        if (pos_tuple.index < (max_tuple * 2))
        {
        remove_noise_data (rule_type);
        }
    }
    if (int_q == 0) {
      if (pos_tuple.index > max_tuple) {
       printf("\n[1].Further generalization [2]. Extract feature table\n");
       printf("\nSelection: ");
```

```
        scanf("%d", &yes_no);
        if (demo) printf("  %d\n", yes_no);
        if (yes_no == 2) {
            do{
               printf("\n\nAvailable attributes:\n\n");
               for (i = 0; i < schema_table.attr_num; i++)
             printf("[%i].%s\n",i+1, schema_table.attr[i].attr_name);
               printf("\nSelection: ");
               scanf("%d", &select);
               if (demo) printf("  %d\n", select);
               else printf("\n");
            } while ((select < 1) || (select > schema_table.attr_num));
            extract_feature_table(attr_name_array[select-1]);
        }
           else {
            if (pos_tuple.index > max_tuple) further_general(rule_type);
            printf ("\n");
            printf ("\n");
            printf("\n\n****************************************\n");
            printf("*     The final generalized relation   *\n");
            printf("****************************************\n\n");
            print_pos (1);
            printf("\n**********************************************\n");
            printf(" The characteristic rule for %s is: \n", target_str);
            printf("**********************************************\n\n");
            simplify(1);
            printf ("\n");
            printf ("\n");
        }
    }
    else {
         printf ("\n");
         printf ("\n");
         printf("\n\n****************************************\n");
         printf("*     The final generalized relation   *\n");
         printf("****************************************\n\n");
         print_pos (1);
         printf("\n\n [1]. Print out generalized rule [2].Extract feature
               table\n");
         printf("\nSelection: ");
         scanf("%d", &yes_no);
         if (demo) printf("  %d\n", yes_no);
```

```
      if (yes_no == 2) {
       do{
          printf("\n\nAvailable attributes:\n\n");
          for (i = 0; i < schema_table.attr_num; i++)
         printf("[%i]. %s\n", i+1, schema_table.attr[i].attr_name);
          printf("\nSelection: ");
          scanf("%d", &select);
          if (demo) printf("  %d\n", select);
          else printf("\n");
       } while ((select < 1) || (select > schema_table.attr_num));
       extract_feature_table(attr_name_array[select-1]);
       }
    else {
       printf("\n*********************************************\n");
       printf("  The characteristic rule for %s is: \n", target_str);
       printf("*********************************************\n\n");
       simplify(1);
       printf ("\n");
       printf ("\n");
       }
 }
}
else { /* answer explanation for intelligent query answering */
      do{
          temp_rel.index = pos_tuple.index;
          for (i = 0; i < pos_tuple.index; i++) {
        temp_rel.table[i].votes = pos_tuple.table[i].votes;
        strcpy(temp_rel.table[i].data, pos_tuple.table[i].data);
        for (j = 0; j < schema_table.attr_num; j++)
            temp_rel.table[i].prop[j] = pos_tuple.table[i].prop[j];
          }
          if (pos_tuple.index>max_tuple) further_general(rule_type);
       printf("\n*********************************************\n");
       printf("  The generalized rule for the answer is: \n", target_str);
       printf("*********************************************\n\n");
          simplify(1);
          pos_tuple.index = temp_rel.index;
          for (i = 0; i < temp_rel.index; i++) {
        pos_tuple.table[i].votes = temp_rel.table[i].votes;
        strcpy(pos_tuple.table[i].data, temp_rel.table[i].data);
        for (j = 0; j < schema_table.attr_num; j++)
            pos_tuple.table[i].prop[j] = temp_rel.table[i].prop[j];
```

```
              }
              printf("\n\nAvailable attributes:\n\n");
              for (i = 0; i < schema_table.attr_num; i++)
             printf("[%i]. %s\n", i+1, schema_table.attr[i].attr_name);
              printf("\nSelection: ");
              scanf("%d", &select);
              if (demo) printf("  %d\n", select);
              else printf("\n");
           } while ((select < 1) || (select > schema_table.attr_num));
           extract_feature_table(attr_name_array[select-1]);
            print_feature_table();
    }

}

/****************************************************************************/
/*  Procedure: extract_feature_table                                        */
/*  Parameter: IN: attibute on which the feature table is based             */
/*  Function : Extract feature table for a perticular attribute             */
/****************************************************************************/

extract_feature_table(attr_name)
    char attr_name[MAXSTR];
{
    /* extract feature table from prime relation */

    int attr_index = 0, index = 0, idx = 0;
    int i = 0, j = 0, k = 0, l = 0, m = 0, n = 0;
    int length = 0;
    int found_set = 0;
    int count = 0, ind = 0;
    char value[MAXSTR];

    strcpy(feature.prime_attr.attr_name, attr_name);
    feature.prime_attr.fea_num = 0;

    while (attr_index < schema_table.attr_num)
    {
       if (strcmp (attr_name, schema_table.attr[attr_index].attr_name) == 0)
           break;
       attr_index++;
    }
```

```
length = schema_table.attr_length;

for (i = 0; i < pos_tuple.index; i++)
 {
   index = attr_index * length;
   /* copy the value in the attr_index attr */
   get_one_value (pos_tuple.table[i].data, index, value);

   for (j = 0; j < feature.prime_attr.fea_num; j ++)
    if (strcmp (value, feature.prime_attr.fea_name[j]) == 0)
        break;
   if (j >= feature.prime_attr.fea_num) {
    count = feature.prime_attr.fea_num;
    strcpy(feature.prime_attr.fea_name[count], value);
    feature.prime_attr.fea_num ++;
   }
 }
count = feature.prime_attr.fea_num;
strcpy(feature.prime_attr.fea_name[count], "Total");
feature.prime_attr.fea_num ++;

index = 0;
attr_index = 0;
while (attr_index < schema_table.attr_num)
{
  if (strcmp (attr_name, schema_table.attr[attr_index].attr_name) != 0)
    {
     length = schema_table.attr_length;
     strcpy(feature.fea[ind].attr_name, schema_table.attr[attr_index].attr_name);
     for (i = 0; i < pos_tuple.index; i++)
     {
      index = attr_index * length;
      /* copy the value in the attr_index attr */
      get_one_value (pos_tuple.table[i].data, index, value);

      for (j = 0; j < feature.fea[ind].fea_num; j ++)
     if (strcmp (value, feature.fea[ind].fea_name[j]) == 0)
         break;
      if (j >= feature.fea[ind].fea_num) {
      count = feature.fea[ind].fea_num;
      strcpy(feature.fea[ind].fea_name[count], value);
      feature.fea[ind].fea_num ++;
      }
     }
     ind ++;
    }
  attr_index ++;
}

feature.attr_num = schema_table.attr_num;
strcpy(feature.fea[ind].attr_name, "vote");
feature.fea[ind].fea_num = 1;

attr_index = 0;
while (attr_index < schema_table.attr_num)
{
   if (strcmp (attr_name, schema_table.attr[attr_index].attr_name) == 0)
       break;
   attr_index++;
}
length = schema_table.attr_length;

for (i = 0; i < pos_tuple.index; i++)
{
   index = attr_index * length;
   /* copy the value in the attr_index attr */
   get_one_value (pos_tuple.table[i].data, index, value);
   for (j = 0; j < feature.prime_attr.fea_num - 1; j++)
    if (strcmp(value, feature.prime_attr.fea_name[j]) == 0) {
        l = j;
        break;
    }
   for (j = 0; j < schema_table.attr_num; j++)
    if (j != attr_index ) {
           if (j > attr_index) m = j-1;
           else m = j;
           index = j * length;
           get_one_value (pos_tuple.table[i].data, index, value);
           for (k = 0; k < feature.fea[m].fea_num; k++)
          if (strcmp(value, feature.fea[m].fea_name[k]) == 0) {
            n = k;
            break;
          }
           feat[l][m][n] = feat[l][m][n] + pos_tuple.table[i].votes;
```

```
                feat[feature.prime_attr.fea_num - 1][m][n] =
                feat[feature.prime_attr.fea_num - 1][m][n] +
                                   pos_tuple.table[i].votes;
        }


        feat[1][schema_table.attr_num - 1][0] =
                            feat[1][schema_table.attr_num - 1][0]
                            + pos_tuple.table[i].votes;
        feat[feature.prime_attr.fea_num-1][schema_table.attr_num - 1][0] =
         feat[feature.prime_attr.fea_num - 1][schema_table.attr_num - 1][0] +
         pos_tuple.table[i].votes;
       }

    if (int_q == 0) print_feature_table();
}

/******************************************************************************/
/*  Procedure: class_tuple_reduction                                          */
/*  Parameter: None                                                           */
/*  Function : Induction process for learning discriminant   rules            */
/******************************************************************************/

class_tuple_reduction()
{
    int i = 0, j = 0;
    int rule_type = 2;
    int attr_num = schema_table.attr_num;
    int unmark_num = 0;
    char attr_name_array[MAXATTR][MAXSTR];
    for (i = 0; i < attr_num; i++)
    {
        strcpy (attr_name_array[i], schema_table.attr[i].attr_name);
    }

    /* 1 indecates the positive tuple table, -1 the negtive tuple table */
    /* for each attribute, if the number of distinct values is greater
       than the threshold, all the values in this attribute should
       be generalized */

    for (i = 0; i < attr_num; i++)
        if (distinct_val(attr_name_array[i],max_value) == 1) {
            generalize(attr_name_array[i], rule_type);
            /*2 indicates discriminant rule */
        }

    /* determine the number of the unmarked tuples in the generalized relation */

    remove_same_1(1);
    remove_same_1(-1);
    intersect();

    for (i = 0; i < pos_tuple.index; i++)
        if (pos_tuple.table[i].data[schema_table.attr_length *
           schema_table.attr_num] != '*')
            unmark_num ++;

    /* if the number of unmarked tuples is greater than the threshold value,
       further generalization on some selected attribute(s) should be performed */

    if (unmark_num > max_tuple)
    {
        /* when the size of unmarked pos_tuple has been reduced to */
        /* max_tuple * 2, check the noise data and remove it */
        if (unmark_num < (max_tuple * 2))
        {
        remove_noise_data (rule_type);
        }
        if (unmark_num > max_tuple)
        further_general(rule_type);
    }

    printf("\n\n*************************************************\n");
    printf ("*    The final generalized relation    *\n");
    printf ("***********************************************\n");
    remove_mark_tuple ();
    print_pos (2);
}

/*****************************************************************************/
/*  Procedure: distinct_val                                                  */
/*  Parameter: IN: attribute name                                            */
/*          threshold value for this attribute                               */
/*  Function : Determine if the number of distince value of a perticular     */
/*         attribute is greater than a predefined threshold value              */
/******************************************************************************/

distinct_val(attr_name, limits)
char *attr_name;
int limits;
{
    /* return the number of distinct values in attribute "attr_name" */
    int index = 0;
    int j = 0;
    int attr_index = 0;
    int length = 0;
    int distinct_num = 0;
    int same;
    char value[MAXSTR];
    char dist_val_table[MAX_THRES][MAXSTR];
    struct tuple *pos_ptr = &pos_tuple.table[index];
    /* find the attr whose name is "attr_name" */
    while (attr_index < schema_table.attr_num)
    {
       if (strcmp (attr_name, schema_table.attr[attr_index].attr_name) == 0)
           break;
       attr_index++;
    }
    length = schema_table.attr_length;
    for (index = 0; index < pos_tuple.index; index++, pos_ptr++)
    {
         if (distinct_num > limits) return (1);
         if (pos_ptr->data[schema_table.attr_num * length] != '*')
         {
        get_one_value (pos_ptr->data, attr_index * length, value);
        same = 0;
        for (j = 0; j < distinct_num; j++)
            if (strcmp (dist_val_table[j], value) == 0)
            {
                same = 1;
                break;
            }
        if (same == 0)
        {
            strcpy (dist_val_table[distinct_num], value);
            distinct_num++;
        }
          }
    }
    return (-1);
}

/*****************************************************************************/
/*  Procedure: distinct_val_1                                               */
/*  Parameter: IN: attribute name                                           */
/*  Function : Return the number of distince value of a perticular attribute */
/*****************************************************************************/

int distinct_val_1(attr_name)
char *attr_name;
{
    int index = 0;
    int j = 0;
    int attr_index = 0;
    int length = 0;
    int distinct_num = 0;
    int same;
    char value[MAXSTR];
    char dist_val_table[MAXTUPLE][MAXSTR];
    struct tuple *pos_ptr = &pos_tuple.table[index];
    /* find the attr whose name is "attr_name" */
    while (attr_index < schema_table.attr_num)
    {
       if (strcmp (attr_name, schema_table.attr[attr_index].attr_name) == 0)
           break;
       attr_index++;
    }
    length = schema_table.attr_length;
    for (index = 0; index < pos_tuple.index; index++, pos_ptr++)
    {
         if (pos_ptr->data[schema_table.attr_num * length] != '*')
         {
        get_one_value (pos_ptr->data, attr_index * length, value);
        same = 0;
        for (j = 0; j < distinct_num; j++)
            if (strcmp (dist_val_table[j], value) == 0)
            {
                same = 1;
                break;
            }
        if (same == 0)
        {
            strcpy (dist_val_table[distinct_num], value);
            distinct_num++;
        }
          }
    }
    return (distinct_num);
}

/**************************************************************************/
/*  Procedure: generalize                                                 */
/*  Parameter: IN: attribute name                                         */
/*         rule type -- 1. characteristic rule 2. discriminant rule       */
/*  Function : Perform generalization on a perticular attribute           */
/**************************************************************************/

generalize(attr_name, rule_type)
char *attr_name;
int rule_type;
{
    int found = 0;
    int exptype = 1;
    int bias_idx = 0;
    int attr_idx = 0;
    struct bias *bias_ptr = &bias_table.table[bias_idx];
    struct attr_info *attr_ptr = &schema_table.attr[attr_idx];

    /* find the index of attr_name in the schema */

    while (attr_idx < schema_table.attr_num)
    {
       if (strcmp (attr_name, attr_ptr->attr_name) == 0)
           break;
       attr_idx++;
       attr_ptr++;
    }

    /* check whether there is any bias for this attribute */
    for (bias_idx = 0; bias_idx < bias_table.index; bias_idx++, bias_ptr++)
    {
        if (strcmp (attr_name, bias_ptr->attr_name) == 0)
        {
            found = 1;
            break;
        }
    }
    /* no bias for this attribute */
    if (found == 0)
    {
       project (attr_name);
    }
    /* there is bias for this attribute */
    else
    {
       /* for learning characteristic rule */
       if (rule_type == 1)
       {
        do {
           substitute (attr_name, exptype);
        } while (distinct_val(attr_name,max_value) == 1);
       }
       /* for learning discriminant rule */
       else
       {
        do {
           exptype = 1;
           substitute (attr_name, exptype);
           /* remove_same_1(exptype); */
           exptype = -1;
           substitute (attr_name, exptype);
           /* remove_same_1(exptype); */
           /* intersect (); */

           /*for NSerc project */
           /*if (watch == 1)
             print_both ();*/
        } while (distinct_val(attr_name,max_value) == 1);
       }
    }
}
```

```
/*****************************************************************************/
/*  Procedure: substitute                                                    */
/*  Parameter: IN: attribute name                                            */
/*         example type -- 1. positive examples 2. negative examples         */
/*  Function : substitute the lower level concept in a perticular attribute  */
/*         by the higher level concepts                                      */
/*****************************************************************************/

substitute (attr_name, exp_type)
char *attr_name;
int exp_type;
{

    /* substitute the lower level concept in attribute "attr_name"
       by the higher level concepts */

    int attr_index = 0, index = 0, idx = 0, id = 0;
    int i = 0, j = 0, k = 0, m = 0;
    int length = 0;
    int found_set=0;
    int bias[MAXBIAS];
    int b_index = 0, b_idx = 0;
    char value[MAXSTR], value1[MAXSTR];
    while (attr_index < schema_table.attr_num)
    {
       if (strcmp (attr_name, schema_table.attr[attr_index].attr_name) == 0)
           break;
       attr_index++;
    }
    length = schema_table.attr_length;

    for (i = 0; i < bias_table.index; i++)
        if (strcmp (bias_table.table[i].attr_name, attr_name) == 0) {
        bias[b_index] = i;
        b_index ++;
        }

    /* for positive examples */
    if (exp_type == 1)
    {
     for (i = 0; i < pos_tuple.index; i++)
```

```
{
  index = attr_index * length;
  /* copy the value in the attr_index attr */
  get_one_value (pos_tuple.table[i].data, index, value);

  for (k = 0; k < b_index; k++)
   {
     b_idx = bias[k];
     if ((strcmp (value, bias_table.table[b_idx].low.concept) == 0) ||
         (check_concept(value, b_idx) == 1))
     {
     /* substitute the lower concept by the higher concept */
    idx = attr_index * length;
    m = 0;
    if (bias_table.table[b_idx].num == -1)
       strcpy(value1, bias_table.table[b_idx].high.concept);
    else {
       id = bias_table.table[b_idx].num;
       strcpy(value1, bias_table.table[id].low.concept);
    }
    while (m < length && value1[m] != ' ' && value1[m] != '\n' &&
       value1[m] != '\0')
    {
      pos_tuple.table[i].data[idx] = value1[m];
      m++;
      idx++;
    }
    while (m < length)
    {
      pos_tuple.table[i].data[idx] = ' ';
      m++;
      idx++;
    }
    break;
     }
 }
}

/* for negative examples */
else
{
 for (i = 0; i < neg_tuple.index; i++)
```

```
    {
      index = attr_index * length;
      /* copy the value in the attr_index attr */
      get_one_value (neg_tuple.table[i].data, index, value);

     for (k = 0; k < b_index; k++)
       {
         b_idx = bias[k];
         if ((strcmp (value, bias_table.table[b_idx].low.concept) == 0) ||
         (check_concept(value, b_idx) == 1))
         {
         /* substitute the lower concept by the higher concept */
        idx = attr_index * length;
        m = 0;
        if (bias_table.table[b_idx].num == -1)
           strcpy(value1, bias_table.table[b_idx].high.concept);
        else {
           id = bias_table.table[b_idx].num;
           strcpy(value1, bias_table.table[id].low.concept);
        }
        while (m < length && value1[m] != ' ' && value1[m] != '\n' &&
           value1[m] != '\0')
        {
          neg_tuple.table[i].data[idx] = value1[m];
          m++;
          idx++;
        }
        while (m < length)
        {
          neg_tuple.table[i].data[idx] = ' ';
          m++;
          idx++;
        }
        break;
         }
      }
   }
}

/*****************************************************************************/
/*  Procedure: further_general                                               */
/*  Parameter: IN: rule type -- 1. characteristic rule  2. discriminant rule */
/*  Function : Perform further generalization on prime relation                */
/****************************************************************************/

further_general (rule_type)
int rule_type;
{
    /* perform further generalization */

    int i = 0, l_index=0, l_dist_num, c_disc_num, tuple_num;
    int unmark_num = 0;
    int attr_num = schema_table.attr_num;
    char max_attr[MAXSTR];

    do {
    i = 0;
    l_index = 0;
    unmark_num = 0;
    strcpy (max_attr, schema_table.attr[i].attr_name);
    /* for learning characteristic rules */
    if (rule_type == 1)
    {
       /* select the attribute which has most distinct values */

       i = 1;
       l_dist_num = distinct_val_1(schema_table.attr[l_index].attr_name);
       while (i < schema_table.attr_num)
       {
            c_disc_num = distinct_val_1(schema_table.attr[i].attr_name);
            if (l_dist_num < c_disc_num) {
                l_index = i;
                l_dist_num = c_disc_num;
            }
            i++;
       }
       strcpy(max_attr, schema_table.attr[l_index].attr_name);

       generalize (max_attr, rule_type);
       remove_same_1(1);
       if (pos_tuple.index > max_tuple)
       {
           /* when the size of unmarked pos_tuple has been reduced to */
           /* max_tuple * 2, check the noise data and remove it */
```

```
            if (pos_tuple.index < (max_tuple * 2))
            {
            remove_noise_data (rule_type);
            }
        }
        tuple_num = pos_tuple.index;
    }

    /* for learning discriminant rule */
    else
    {
        while (i < schema_table.attr_num - 1)
        {
         if (distinct_val_1(schema_table.attr[i].attr_name) <
             distinct_val_1(schema_table.attr[i+1].attr_name))
            strcpy (max_attr, schema_table.attr[i+1].attr_name);
         i++;
        }
        generalize (max_attr, rule_type);
        remove_same_1(1);
        remove_same_1(-1);
        intersect();
        for (i = 0; i < pos_tuple.index; i++)
            if (pos_tuple.table[i].data[schema_table.attr_length *
                        schema_table.attr_num] != '*')
            unmark_num ++;
        if (unmark_num > max_tuple)
        {
            /* when the size of unmarked pos_tuple has been reduced to */
            /* max_tuple * 2, check the noise data and remove it */
            if (unmark_num < (max_tuple * 2))
            remove_noise_data (rule_type);
        }
        tuple_num = unmark_num;
    }
    } while (tuple_num > max_tuple);
}
```

REFERENCES

[1] J. A. Allen and C. R. Perrault. Analysing intention in utterance. *Artificial Intelligence*, 15:143–178, 1980.

[2] J.A. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–160, 1984.

[3] J.R. Anderson and M. Matessa. A rationale analysis of categorization. In *Proc. of 7th International Machine Learning Conference*, pages 76–84, 1990.

[4] Y. Cai, N. Cercone, and J. Han. Attribute-oriented induction in relational databases. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 213–228. AAAI/MIT Press, 1991.

[5] N. Cercone, J. Han, P. McFetridge, F. Popowich, D. Fass, C. Groeneboer, G. Hall, and Y. Huang. System X and DBLEARN: How to get more from your relational database, easily. *accepted by Integrated Computer-Aided Engineering (a special issue on Intelligent Information Systems)*, 1993.

[6] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Trans. Database Syst.*, 15:162–207, 1990.

[7] K. C. C. Chan and A. K. C. Wong. A statistical technique for extracting classificatory knowledge from databases. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 107–124. AAAI/MIT Press, 1991.

[8] P. Cheeseman, P. Kelly, J. Self, and J. Sutz. Autoclass: a bayesian classification system. In *Proc. of 5th International Conference on Machine Learning*, pages 54–65, San Mateo, CA, 1988.

[9] B.A. Cheikes. Methodological issues in the design of intelligent and cooperative information systems. In *Proc. International Conference on Intelligenct and Cooperative Information Systems*, pages 3–12, Rotterdam, Netherlands, May 1993.

[10] P. Chen. The entity-relationship model : Toward a unified view of data. *ACM Trans. Database Syst.*, 1:9–36, 1976.

[11] F. Cuppens and R. Demolombe. Cooperative answering: A methodology to provide intelligent access to databases. In *Proc. 2nd Int. Conf. Expert Database Systems*, pages 621–643, Fairfax, VA, April 1988.

[12] F. Cuppens and R. Demolombe. How to recognize interesting topics to provide cooperative answering. *Information Systems*, 14:163–173, 1989.

[13] B. C. Falkenhainer and R. S. Michalski. Integrating quantitative and qualitative discovery: the ABACUS system. *Machine Learning*, 1:367–401, 1986.

[14] T. Finin, A.K. Joshi, and B.L. Webber. Natural language interactions with artificial experts. *Proceedings of the IEEE*, 74:921–938, 1986.

[15] D. Fisher. Improving inference through conceptual clustering. In *Proc. 1987 AAAI Conf.*, pages 461–465, Seattle, Washington, July 1987.

[16] D. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2:139–172, 1987.

[17] D. Fisher, M. Pazzani, and P. Langley. *Concept Formation: Knowledge and Experience in Unsupervised Learning.* Morgan Kaufmann, 1991.

[18] W. J. Frawley, G. Piatetsky-Shapiro, and C. J. Matheus. Knowledge discovery in databases: An overview. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 1–27. AAAI/MIT Press, 1991.

[19] T. Gaasterland. Restricting query relaxation through user constraints. In *Proc. International Conference on Intelligenct and Cooperative Information Systems*, pages 359–366, Rotterdam, Netherlands, May 1993.

[20] S. Gantimahapatruni and G.Karabatis. Enforcing data dependencies in cooperative information systems. In *Proc. International Conference on Intelligenct and Cooperative Information Systems*, pages 332–341, Rotterdam, Netherlands, May 1993.

[21] L. Gasser. Social conceptions of knowledge and action: DAI foundations and open systems semantics. *Artificial Intelligence*, 47:107–138, 1991.

[22] M. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, 1987.

[23] B.J. Grosz and C.L. Sidner. Attention, intentions, and the structure of discourse. *Computational Linguistics*, 12:175–204, 1986.

[24] J. Han, Y. Cai, and N. Cercone. Knowledge discovery in databases: An attribute-oriented approach. In *Proc. 18th Int'l Conf. Very Large Data Bases*, pages 547–559, Vancouver, Canada, August 1992.

[25] J. Han, Y. Cai, and N. Cercone. Data-driven discovery of quantitative rules in relational databases. *IEEE Trans. Knowledge and Data Engineering*, 5:29–40, 1993.

[26] J. Han, Y. Cai, N. Cercone, and Y. Huang. DBLEARN: A knowledge discovery system for databases. In *Proc. 1st Int'l Conf. on Information and Knowledge Management*, pages 473–481, Baltimore, Maryland, Nov. 1992.

[27] J. Han, Y. Huang, and N. Cercone. Intelligent query answering by knowledge discovery techniques. In *IEEE Trans. Knowledge and Data Engineering*, (to appear), 1993.

[28] J. Han and Z. N. Li. Deductive-ER: Deductive entity-relationship model and its data language. *Information and Software Technology*, 34:192–204, 1992.

[29] J. Han, Y.Cai, N. Cercone, and Y.Huang. Discovery of data evolution regularities in large databases. In *Journal of Computer and Software Engineering(a special issue on methodologies and tools for intelligent information systems), (to appear)*, 1993.

[30] D. Haussler. Bias, version spaces and valiant's learning framework. In *Proc. 4th Int. Workshop on Machine Learning*, pages 324–336, Irvine, CA, 1987.

[31] P. Hoschka and W. Kloesgen. A support system for interpreting statistical data. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 325–346. AAAI/MIT Press, 1991.

[32] X. Hu. Conceptual clustering and concept hierarchies in knowledge discovery. In *M.Sc. Thesis*, Simon Fraser University, Canada, Dec. 1992.

[33] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Comput. Surv.*, 19:201–260, 1987.

[34] T. Imielinski. Intelligent query answering in rule based systems. *J. Logic Programming*, 4:229–257, 1987.

[35] K. A. Kaufman, R. S. Michalski, and L. Kerschberg. Mining for knowledge in databases: Goals and general description of the INLEN system. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 449–462. AAAI/MIT Press, 1991.

[36] J. King. QUIST: A system for semantic query optimization in relational databases. In *Proc. 7th Int. Conf. Very Large Data Bases*, pages 510–517, Cannes, France, 1981.

[37] M. M. Kokar. Coper: a methodology for learning invariant functional descriptions. In Michalski et. al., editor, *Machine Learning: a Guide to Current Research*, pages 151–154. Kluwer Academic Publishers, 1986.

[38] L. V. S. Lakshmanan and R. Missaoui. On semantic query optimization in deductive databases. In *Proc. 8th Int. Conf. Data Engineering*, pages 368–375, Phoenix, AZ, Feb. 1992.

[39] P. Langley, H.A. Simon, G.L. Bradshaw, and J.M. Zytkow. Scientific discovery: Computational explorations of the creative process. *MIT Press*, Cambridge, MA, 1987.

[40] D. B. Lenat. On automated scientific theory formation: a case study using the AM program. In J. E. Hayes, D. Michie, and L. I. Mikulich, editors, *Machine Intelligence 9*, pages 251–286. Halsted Press, 1977.

[41] R. S. Michalski. A theory and methodology of inductive learning. In Michalski et. al., editor, *Machine Learning: An Artificial Intelligence Approach, Vol. 1*, pages 83–134. Morgan Kaufmann, 1983.

[42] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell. *Machine Learning, An Artificial Intelligence Approach, Vol. 2.* Morgan Kaufmann, 1986.

[43] A. Motro. Seave: A mechanism for verifying user presuppositions in query systems. *ACM Transactions on Office Information Systems*, 4, pages 21–38, 1986.

[44] A. Motro. Using integrity constraints to provide intensional responses to relational queries. In *Proc. 15th Int. Conf. Very Large Data Bases*, pages 237–246, Amsterdam, Netherlands, Aug. 1989.

[45] A. Motro and Q. Yuan. Querying database knowledge. In *Proc. 1990 ACM-SIGMOD Int'l Conf. Management of Data*, pages 173–183, Atlantic City, NJ, June 1990.

[46] B. Nordhausen and P. Langley. An integrated approach to empirical discovery. In J. Shrager and P. Langley, editors, *In Computational Models of Scientific Discovery and Theory Formation*, pages 97–128. Morgan Kaufmann, 1990.

[47] D. E. O'Leary. Knowledge discovery as a threat to database security. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 507–516. AAAI/MIT Press, 1991.

[48] G. Piatetsky-Shapiro and C.J. Matheus. Knowledge discovery workbench: An exploratory environment for discovery in business database. *Workshop Notes from the Ninth National Conference on Artificial Intelligence: Knowledge Discovery in Databases*, pages 11–24, 1991.

[49] A. Pirotte and D. Roelants. Constraints for improving the generation of intensional answers in a deductive database. In *Proc. 5th Int. Conf. Data Engineering*, pages 652–659, Los Angeles, CA, Feb. 1989.

[50] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[51] C.L. Rete. A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence*, 19:17–37, 1982.

[52] D.E. Rummelhart and J.L. McClelland. Parallel distributed processing. *MIT Press*, 1, 1986.

[53] J. Schmitz, G. Armstrong, and J.D.C. Little. Coverstory - automated news finding in marketing. *DSS Transactions*, pages 46–54, 1990.

[54] C.D. Shum and R. Muntz. Implicit representation for extensional answers. In *Proc. 2nd Int. Conf. Expert Database Systems*, pages 497–522, Vienna, VA, April 1988.

[55] T. J. Teorey, D. Yang, and J. P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Comput. Surv.*, 18:197–222, 1986.

[56] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Vols. 1 & 2.* Computer Science Press, 1989.

[57] J. Zytkow and J. Baker. Interactive mining of regularities in databases. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 31–54. AAAI/MIT Press, 1991.