ATTRIBUTE-ORIENTED INDUCTION IN OBJECT-ORIENTED DATABASES

by

Jinshi Xia

B.Eng., Tsinghua University, China, 1986

M.Eng., Tsinghua University, China, 1988

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

in the School
of
Computing Science

© Jinshi Xia 1993

SIMON FRASER UNIVERSITY

September 1993

*All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.*

APPROVAL

Name: Jinshi Xia

Degree: Master of Applied Science

Title of thesis: **Attribute-Oriented Induction in Object-Oriented Databases**

Examining Committee: Dr. Wo-Shun Luk, Chairman

Dr. Nick Cercone
Senior Supervisor

Dr. Jiawei Han
Supervisor

Dr. Rizwan Jafferali Kheraj, MPR Teltech Ltd
External Examiner

Date Approved: August 30, 1993

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Attribute-Oriented Induction in Object-Oriented Databases.

Author: ______

(signature)

Jinshi Xia

(name)

Sept. 7, 1993

(date)

ABSTRACT

Knowledge discovery in databases is the nontrivial extraction of implicit, previously unknown, and potentially useful information from data such that the extracted knowledge may facilitate deductive reasoning and query processing in database systems. This branch of study has been ranked among the most promising topics for database research for the 1990s.

Due to the dominating influence of relational databases in many application fields, knowledge discovery from databases has been largely focused on relational databases. The gradual adoption of *object-oriented database systems* has expressed a need for the study of knowledge discovery from object-oriented databases as well. *Object-oriented databases*(OODBs) are concerned with complex data structures and diverge greatly from relational database systems. In order to effectively conduct knowledge discovery in an object-oriented database, existing relational algorithms need be modified accordingly to take full advantage of the object-oriented data model.

The attribute-oriented induction method has been successful for knowledge discovery in relational databases and we choose this method to study the new demands OODBs impose on a learning algorithm. In this thesis, we study the characteristics of the object-oriented data model and their effects on the attribute-oriented induction algorithm. We extend the attribute-oriented induction method to object-oriented

paradigms, focusing on handling complex attributes, and present a algorithm for learning characteristic rules in an object-oriented database. We follow the *least commitment principle* and break down complex objects into primitive ones and then apply attribute-oriented generalization techniques. Learning in databases with a cycled class composition hierarchy is specifically addressed.

ACKNOWLEDGMENTS

I would like to thank Professor Nick Cercone and Professor Jiawei Han for supervising this thesis. Their support, encouragement, and guidance helped me make this thesis a reality. I would also like to thank Dr. Rizwan Kheraj for his helpful comments.

CONTENTS

ABSTRACT iii

ACKNOWLEDGMENTS v

1 Introduction 1

2 Knowledge Discovery in Databases 4

2.1 An Overview 4

2.1.1 Formalization 5

2.1.2 Algorithms 5

2.1.3 Types of Learned Rules 7

2.1.4 Evaluation 7

2.2 The Attribute-Oriented Induction Method 9

3 The Object-Oriented Database Model 19

3.1 Objects and Object Identifiers 20

3.2 Complex Attributes and Methods 21

3.3 Class 23

3.4 An Example Database 25

4 Attribute-Oriented Induction in OODBs 30

4.1 General Discussion 30

4.2 Complex Attributes 33

4.2.1 Collection Attributes 34

4.2.2 Reference Attributes 38

4.3 The **OOLCHR** Algorithm 41

5 General Improvement of the Induction Method 44

5.1 Maintaining Generalization Correctness 46

5.2 Induction Speedup 50

5.3 The **OOLCHR*** Algorithm 54

6 Concluding Remarks 56

6.1 Conclusions 56

6.2 Future Research 58

REFERENCES 61

CHAPTER 1

Introduction

The computerization of scientific, business, and government activities produces an ever-increasing stream of data because every bit of information is typically recorded in a computer. This flood of raw data outgrows human abilities to analyze and calls for automated data analysis and knowledge extraction from databases[10]. The study of this automated process, *knowledge discovery*, also known as *data mining*, has been ranked among the most promising topics for database research for the 1990s[20]. By learning from databases, interesting relationships among data can be discovered automatically, and the extracted knowledge may facilitate deductive reasoning and query processing in database systems.

Due to the dominating influence of relational databases in many application fields, the focus of learning from databases has been largely targeted toward relational

databases and there have been proposed a number of successful methods. The attribute oriented-induction method for knowledge discovery in databases[2] is an effective and efficient tool in relational databases as well as in extended relational and deductive databases. The past decade has seen the widespread acceptance of relational database management systems(DBMSs) for business applications. However, existing commercial DBMSs, both small-scale and large-scale, have proven inadequate for applications such as computer-aided design, software engineering, and office automation. A new DBMS–*object-oriented database system*– has been proposed, studied, and constructed[7]. The new objected-oriented data model opens up a new topic in knowledge discovery in databases.

Object-oriented databases(OODBs) are concerned with complex data structures, such as those required to represent the parts of a document, a program, or a design. They diverge greatly from relational database systems in that the value of an attribute is allowed to be non-atomic, and that relationship between objects is via references rather than foreign keys. In order to effectively conduct knowledge discovery in an object-oriented database, existing relational algorithms need be modified accordingly to take full advantage of the object-oriented data model.

In this paper, we study the characteristics of the object-oriented data model and their effects on attribute-oriented induction algorithms. We extend the attribute-oriented induction method to object-oriented paradigms, focusing on handling complex attributes, and present the **OOLCHR** algorithm for learning characteristic rules in an object-oriented database. We conclude that the attribute-oriented method can also be used in object-oriented databases. We also examine the induction process in detail and present an asynchronous multiple-level ascension algorithm, **Async*** to

guarantee the correctness of the induction and to improve the induction efficiency.

The organization of this paper is as follows. Chapter 2 introduces the primitives of knowledge discovery in databases and the relational attribute-oriented induction method. Chapter 3 gives a general description of the object-oriented database model. Chapter 4 summarizes our study of attribute-oriented induction in object-oriented databases and presents the **OOLCHR** algorithm. Chapter 5 discusses methods to guarantee the correctness of the induction process and to improve the induction efficiency, and the result is summarized in the **Async*** Algorithm. Chapter 6 concludes our study.

CHAPTER 2

Knowledge Discovery in Databases

We introduce some theoretical issues related to knowledge discovery in databases in general, and the relational attribute-oriented induction method in particular.

2.1 An Overview

It has been estimated that the amount of information in the world doubles every 20 months and the size and number of databases probably increases even faster. The computerization of scientific, business, and government activities produces an ever-increasing stream of data because every bit of information is typically recorded in a computer. This flood of raw data outgrows human abilities to analyze and calls for automated data analysis and knowledge extraction from databases[10]. The study of this automated process, *knowledge discovery*, also known as *data mining*, has been ranked among the most promising topics for database research for the 1990s[20].

2.1.1 Formalization

Knowledge discovery is the nontrivial extraction of implicit, previously unknown, and potentially useful information from data. It can be characterized as follows[10]. Given a set of facts(data) F, a language L, and some measure of certainty C, we define a *pattern* as a statement S in L that describes relationships among a subset Fs of F with a certainty C, such that S is simpler(in some sense) than the enumeration of all facts in Fs. 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.* The output of a program that monitors the set of facts in a database and produces patterns in this sense is *discovered knowledge.*

The analysis and extraction of knowledge from large databases is a very important application field of machine learning. Since a database generally contains a large number of records and each record can be regarded as an example, the machine learning paradigm, *learning from examples*, has been the one used by many learning algorithms. From the viewpoint of machine learning, *learning from examples* belongs to *synthetic symbolic empirical learning*[18] and can be defined as the process of reasoning from the specific to the general. In the interest of our study in this thesis, *learning from examples* is interchangeable with *induction* or *generalization.*

2.1.2 Algorithms

Discovery algorithms are procedures designed to extract knowledge from data; they usually contain two steps. The first step is to recognize interesting patterns in data. Traditional numerical analysis and conceptual clustering techniques can be used[1, 15].

The second step is to describe the concept in a concise and meaningful manner, that is, to generate rules. Symbolic learning methods(synthesis) and explanation-based learning methods(analysis) can be used independently or combined. In machine learning, these two processes are sometimes referred to as *unsupervised* and *supervised* learning respectively. Since very large databases abound with raw data, the inductive learning from examples(synthesis) has been widely used in knowledge discovery from large databases.

The inputs to a learning algorithm may include a database, domain knowledge, the learning task, and some constraints for controlling the learning output. The database provides learning *examples*, that is, each instance is generally considered an example in the *learning from examples* paradigm. The learning task specifies the goal of one learning session, and the learning constraints lay down some limitations to the format and form of the final learned rule(the output). *Domain knowledge*(also called background knowledge) may include a wide spectrum of human knowledge. The degree to which domain knowledge is used affects the applicability of a knowledge discovery algorithm. The use of domain knowledge to the extreme will produce a specialized learning algorithm, i.e., domain-specific algorithm, that will outperform any general method in its domain but will not be useful outside it. A good general-purpose learning system should provide a general facility for incorporating domain-specific knowledge into the induction process as an exchangeable package(ideally the domain-specific knowledge should be isolated from the general purpose inductive process)[18]. The presence/absence of this facility also reflects the flexibility of a learning system.

2.1.3 Types of Learned Rules

The goal of knowledge discovery in databases is to generate knowledge represented in a high level language form. Two types of rules, *characteristic rules* and *classification rules* can be learned from databases based on the following two learning processes.

- Summarization(finding characteristic rules): Summarize class records by describing their common or characteristic features.
- Discrimination(finding classification rules): Describe qualities sufficient to discriminate records of one class from another.

Thus, the symptoms of a specific disease can be summarized as a characteristic rule, while a classification rule should be generated to distinguish one disease from others.

2.1.4 Evaluation

Discovery algorithms for large databases can be evaluated by the following criteria[8, 9].

- *Adequacy of the representation language*: Knowledge representations such as frames and semantic nets are semantically rich but may lead to exponential increase in complexity to the induction process while the less structured representations such as attribute-value lists are more uniform. A choice is made according to what is symantically needed and the uniformity that is available to the representation.

- *Rules of generalization implemented*
- *Computational efficiency*
- *Flexibility and extensibility*: ease to be extended to discover descriptions with forms other than conjunctive generalizations and to include mechanisms which facilitate the detection of errors in the input data, and to provide a general facility for incorporating domain-specific knowledge into the induction process as an exchangeable package, and to perform constructive induction.

The current research in generic algorithms focuses on the development of efficient incremental algorithms that are guided by expert knowledge[10]. Some recently proposed learning algorithms were surveyed in [2].

These algorithms, when applied to large databases, are either low in computational efficiency or generate an overly large set of rules. The poor computational efficiency stems from the fact that these algorithms adopt the tuple-oriented approach which examines the training examples tuple by tuple and thus have a large search space. To reduce the size of the set of rules generated, some knowledge-guided control mechanism need be introduced into the learning process so that only interesting rules are generated in a well-formed format. The attribute-oriented approach to knowledge discovery in databases put forward in [2] represents a step forward in improving the learning efficiency and the quality of the learned rule. Experiments have shown that it is a very promising learning method for very large relational databases. The next section will introduce it in detail.

2.2 The Attribute-Oriented Induction Method

The attribute-oriented induction method presented in [2, 11] is primarily targeted towards relational databases. Following the *learning from examples* paradigm, it applies an attribute-oriented concept tree ascending technique which integrates database operations with the learning process. This method substantially reduces the complexity of the database learning processes and can learn both conjunctive rules and restricted forms of disjunctive rules.

Two algorithms, **LCHR** and **LCLR**, have been developed to learn characteristic rules and classification rules respectively. Since both algorithms utilize essentially the same generalization techniques, we concentrate on illustrating the procedure of **LCHR** and all discussions hereafter will be with regard to learning characteristic rules.

First of all, data relevant to the query submitted by the user is collected into one relational table using database operations such as selection, projection, and join. For each of the attribute in the table, there can exist a concept hierarchy. The concept hierarchy is where we incorporate higher level concepts in the learning process. Different levels of concepts can be organized into a taxonomy of concepts. The concepts in a taxonomy can be partially ordered according to general-to-specific ordering. The most general point is the null description(described by a reserved word "ANY"), and the most specific points correspond to the specific values of attributes in the database. Usually, the concept hierarchies are provided prior to the learning process by domain experts. An example concept hierarchy is shown in Figure 2.1. Alternately, this concept tree can be represented in a textual form as shown in Figure 2.3, where the

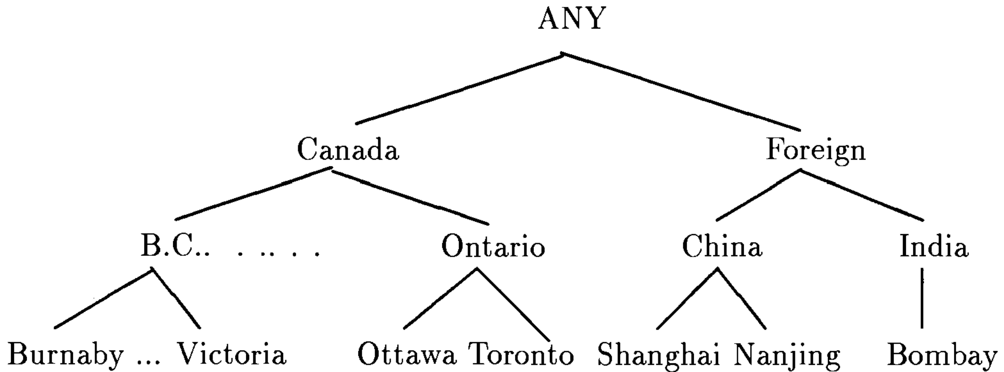

Figure 2.1: The concept hierarchy for `Birthplace`(graphical form)

concept appearing after the ":" is a generalization of the concepts inside the curly brackets. We only use the graphical form when we examine the tree ascension technique in Chapter 5. Since most concept hierarchies are structured like a tree, we use concept hierarchies and concept trees interchangeably hereafter.

The goal of the learning algorithm is to represent the set of learning examples(tuples) in more general terms and thereby reduce the number of tuples to within a certain predefined value, the *threshold*, by using the following strategies.

Strategy 2.1 *If there is a large set of distinct values for an attribute but there is no concept hierarchy provided for the attribute, the attribute should be removed during generalization.*

Strategy 2.2 *Generalization should be performed on the non-decomposable components of a data relation.*

Strategy 2.3 *If there are many distinct values for an attribute and there exists a*

concept tree for the attribute, each value in the attribute of the relation should be substituted for by a higher level concept in the learning process.

Strategy 2.4 *If the number of distinct values of an attribute is larger than the specified threshold value, further generalization on this attribute should be performed.*

Strategy 2.5 *If the number of tuples of a generalized relation is larger than the specified threshold value, further generalization on the relation should be performed.*

In a relational database, the First Normal Form(1NF) is generally observed and each attribute is simply the non-decomposable component of a data relation. Therefore, Strategy 2.2 states the obvious. However, the principle is very important for databases that do not observe 1NF. We will discuss this further when we come to attribute-oriented induction in object-oriented databases in subsequent chapters. Strategy 2.4 limits the maximum number of distinct values for an attribute in the final learned result and Strategy 2.5 limits the maximum number of tuples in the final table. Strategy 2.1 corresponds to the *dropping condition* rule, and Strategy 2.3 corresponds to the *climbing tree* rule in machine learning. The concept tree ascending technique is the major generalization technique used in both attribute-oriented generalization and tuple-oriented generalization. Since generalization in an attribute-oriented approach is performed on individual attributes first and the learned rule is obtained by piecing together generalized attributes, it greatly reduces the computational complexity of the induction method[2].

We now show this algorithm through an example. Suppose we have a university database containing information about students and we want to learn characteristic

```
{computing, math, biology, chemistry, statistics, physics}:science
{music, history, liberal arts, literature}:art
{science, art}:ANY
```

Figure 2.2: Concept hierarchy for `Major`

```
{Burnaby, Richmond, Vancouver, Victoria}:British Columbia
{Calgary, Edmonton}:Alberta
{Ottawa, Toronto}:Ontario
{Bombay}:India
{Shanghai, Nanjing}:China
{China, India}:Foreign
{British Columbia, Alberta, Ontario}:Canada
{Foreign,Canada}:ANY
```

Figure 2.3: The concept hierarchy for `BirthPlace`

rules of graduate students. The concept hierarchies for the relevant attributes are given in Figure 2.2, Figure 2.3, and Figure 2.4. By following the steps just introduced, we first collect data related to "graduate" into a relational table, as shown in Table 2.1.

By Strategy 2.1, attribute `Name` is removed. We then examine the remaining three attributes. Each attribute contains many distinct values and is associated with some higher level concept in the concept tree of Figure 2.2, Figure 2.3, or Figure 2.4. We should replace the value of an attribute by its higher level concept in the concept tree

```
{2.0--2.9}:average
{3.0--3.4}:good
{3.5--4.0}:excellent
{average, good, excellent}:ANY
```

Figure 2.4: The concept hierarchy for `GPA`

to generalize its representation, for instance, from "physics" to "science" and from "Vancouver" to "British Columbia", using Strategy 2.3 and Strategy 2.4. The result is shown in Table 2.2. Suppose we have a threshold value of three in this example, and by Strategy 2.5, further generalization is needed. The final relational table is shown in Table 2.3.

The output of the learning result is in order. Depending on how the learned knowledge is to be used, various transformations of the final table can be conducted. Put in English, the rule says, "a graduate is either a canadian with an excellent GPA or a foreign student majoring in sciences with a good GPA". Or we can represent the learned rule using predicate logic, as shown below.

```
for any x, graduate(x)=>
           (Birth_Place(x)=Canada & GPA(x)=excellent) |
           (Major(x)=science & Birth_Place(x)=Foreign & GPA(x)=good).
```

We may also leave the result in its tabular form for further study. In this thesis, we represent the final learning result in a tabular form together with an English interpretation.

The attribute-oriented induction method can also be applied to learning classification rules. The difference is that in the extraction of classification rules, the facts which support the target class(e.g., "graduates") serve as positive examples, while the facts which support the other class(es)(e.g., "undergraduates") serve as negative examples. Since the learning task is to discover the concepts that have discriminant properties, the portion of facts in the target class that overlaps with other classes

Name	Major	BirthPlace	GPA
Anderson	history	Vancouver	3.5
Fraser	physics	Ottawa	3.9
Gupta	math	Bombay	3.3
Liu	biology	Shanghai	3.4
Monk	computing	Victoria	3.8
Wang	statistics	Nanjing	3.2

Table 2.1: The set of data relevant to "graduates"

Major	BirthPlace	GPA
art	B.C.	excellent
science	Ontario	excellent
science	B.C.	excellent
science	India	good
science	China	good

Table 2.2: A generalized relation

Major	BirthPlace	GPA
art	Canada	excellent
science	Canada	excellent
science	Foreign	good

Table 2.3: Further generalization of the relation

should be detected and removed from the description of classification rules. For instance, suppose we want to learn a classification rule which distinguishes graduate students from undergraduate students. We can start the learning process as if we were learning two characteristic rules for graduate students and undergraduate students. In the process of generalization, we remove overlapping tuples that are shared by both graduate class and undergraduate class. The final learned rule for graduate students will be discriminant due to the fact that relevant tuples belong to the graduate class *only*.

The attribute-oriented induction method can be summerized in terms of the evaluation criteria we introduced earlier as follows.

- *Representation* The learned rule can be represented in predicate logic which is adequate for most applications
- *Rules implemented* Dropping conditions and tree climbing
- *Computational efficiency* Many operations during the learning process can be carried out using database operations[1]. The reduction of the search space due to attribute orientation instead of tuple orientation greatly improves the efficiency
- *Extensibility* Allows disjunctive rules and constructive induction and the domain-specific knowledge(the concept hierarchy) can be implemented as an exchangeable package

[1]Taking advantage of well-implemented database operations can be complicated and database-dependent. This issue is discussed at length in Chapter 6.

```
ALGORITHM LCHR;
INPUT: (1) a relational database,
       (2) a set of concept trees,
       (3) the learning task, and
       (4) the threshold value(T).
OUTPUT: A characteristic rule learned from the database.
BEGIN
  Step 1. Select the task-relevant data into table P.
  Step 2. Call Procedure INDUCTION(P,T).
  Step 3. Transform the final relation into a predicate formula.
END
```

Figure 2.5: The **LCHR** algorithm

We now give the formalization of the **LCHR** algorithm in Figure 2.5 and Figure 2.6.

The **LCHR** and **LCLR** algorithms are designed for learning characteristic rules and classification rules from relational databases. Therefore, both algorithms can only handle the well-formatted data stored in relational databases. With the wider application of database concepts, new database systems have emerged to suit the need of non-business data management, such as computer-aided design and office automation. These new generation database systems differ from relational database systems in quite a number of facets, and the need for knowledge discovery is also urgent to deal with the explosion of data. To conduct learning in the new database systems, the learning algorithm should take into account all the new features. On the other hand, existing learning algorithms have left much room for improvement. These two considerations are the motivation for our study of attribute-oriented induction in object-oriented databases which will be presented in the following chapters. For clarity, we will refer to the **LCHR** as the *relational* attribute-oriented induction

```
PROCEDURE INDUCTION(P, T);
/* P consists of attributes A1, A2, ..., An; d1, d2, ..., dn denote
 * the number of distinct values of A1, A2, ..., An in the current
 * (working) table; and N the total number of tuples in the table.
 */
BEGIN
  FOR i := 1 TO n DO
  BEGIN
    WHILE di > T DO
      IF there does not exist a concept tree for Ai,
      THEN remove Ai
      ELSE replace each value of Ai by its parent in the concept tree
           and eliminate redundant tuples;
  END;
  WHILE N > T DO
    BEGIN
      Select the attributes containing substantially more distinct
      values or with a better reduction ratio, and replace each
      value of them by its corresponding parent in the concept
      tree, and eliminate redundant tuples;
    END
END.
```

Figure 2.6: The **INDUCTION** procedure

algorithm.

CHAPTER 3

The Object-Oriented Database Model

In the past decade, there have been major changes in products for business applications, including the widespread acceptance of relational database management systems(RDBMSs). However, it is sometimes inefficient and clumsy to use existing commercial DBMSs for applications such as computer-aided design, software engineering, and office automation. These applications all involve complex structures and relationships among structures that cannot be directly represented using the relational database model. A new DBMS–*object-oriented database system*– has been proposed, studied, and constructed[7].

Object-oriented databases(OODBs) are concerned with complex data structures, such as those required to represent the parts of a document, a program, or a design. They treat any real-world entity as an object and represent the entity in a

straight-forward manner. In this sense, the advent of OODBs brings about important methodological advantages, notwithstanding a lack of theoretical importance. To date, there is no consensus about *precisely* what object-oriented means in general, and what object-oriented database is in particular, but an object-oriented data model should contain the following core concepts: object and object identifier, attributes and methods, class, and class hierarchy and inheritance[16]. Object and object identity define the *structural* object-orientation of a DBMS, that is, with objects as data structures. Methods are used to encapsulate the semantics of an object and characterize the *behavioral* object-orientation of a DBMS. Without loss of generality, in the next several sections, we will present a brief discussion on these core concepts.

3.1 Objects and Object Identifiers

Essentially all OODBs incorporate the two most basic features of objects:

Object grouping: Objects can serve to group data that pertain to one real-word entity. For example, we can treat a document as an object which groups chapters, indexes, appendices, etc. into one entity, namely, a document. Chapters serve as attributes of the document object. In like manner, a chapter can be defined as another type of object which groups sections into one entity. The uniform treatment of any real-word entity as an object simplifies the user's view of the real world. This implies that the state of an object consists of values for the attributes of the object, and the values are themselves objects, possibly with their own states. This leads to the concept of complex attributes whose values are non-primitive objects.

Object identity: Objects can have a unique identity independent of the values that they contain. A system that is *identity-based* allows an object to be referenced via a unique internally generated number, an *object identifier*, independent of the value of its primary key, if any. The adoption of object identifier facilitates the representation of the state of an object, namely, the state of an object is naturally represented as a set of identifiers of the objects which are the values of the attributes of the object. For performance reasons, if the domain of an attribute is a primitive class, the values of the attribute are directly represented; that is, instances of a primitive class have no identifiers associated with them.

In a nutshell, an object is an aggregation of attribute-value pairs and the value can be an object of another type. Each object has a unique internal value to be identified.

3.2 Complex Attributes and Methods

In relational databases, an attribute corresponds to a column of a relation. In object-oriented databases, the domain of an attribute may be any class: user-defined or primitive. This represents a significant difference from the normalized relational model in which the domain of an attribute is restricted to a primitive class[16]. In an object-oriented database, object attributes may have *complex* values, such as sets or references to other objects. There are three kinds of *complex* attributes: references, collections, and procedures.

Reference attributes, or associations, are used to represent relationships between objects. They take on values that are objects–that is, references to entities. Reference attributes are analogous to pointers in a programming language, or to foreign keys in

a relational system[7]. In our document example, the chapter attribute of a document object can be a reference attribute, i.e., it takes a pointer to a chapter object as its value. The ability to take an object as the value for an attribute greatly simplifies the modeling of a database and makes the modeling more straightforward and natural. From the viewpoint of knowledge discovery, it is more natural and convenient if we can pose a query regarding an entity directly instead of having to know the entity's components.

The second kind of complex attribute, *collections*, is used for lists, sets, or arrays of values. The collections may include simple attribute values and also references. Operations are provided for creating, inserting, or deleting an element from a collection. Relational first normal form does not permit collection-valued attributes; in contrast, many object-oriented database systems allow collections. Thus, we diverge in an important way from the relational model on this point. Collection attributes reflect a real world need, for instance, to describe a person's hobbies(set), ordered preferences(list), etc.

Derived attributes are those whose values can be defined procedurally rather than stored explicitly, by specifying a procedure to be executed when the value is retrieved or assigned. For example, we may store such personal information as birth date and age in a personnel database. The birth date will not change but the age does. It would be desirable to define a procedure for the age attribute so that it always represents the difference between the current date and the birth date. Derived data correspond roughly to *views* in the relational database literature, but procedural languages may define more complex derivations than views, and are generally used to define individual attributes rather than relations. Since knowledge discovery in databases is a read-only

operation to the database and does not change the state of the database in any way, derived attribute values, once retrieved from the database, can be treated just like regular attribute values. Derived attributes do not pose any special problem to the knowledge discovery process, so we will not discuss derived attributes further.

Methods are procedures used in OODBs to encapsulate or "hide" the attributes of an object, providing the only interface to access the attributes. This encapsulation allows an implementation to be changed without affecting programs using a data type. The attributes associated with an object are private, and only an object's methods may examine or update these data; the methods are public[7]. Since knowledge discovery algorithms work on a copy of the data stored in the database and do not change the stored data or other implementations in any way, the methods do not effect the learning process and we will not discuss them further.

3.3 Class

Class is used to group together objects that respond to the same message, use the same methods, and have variables of the same name and type. Each such object is called an *instance* of its class. All objects in a class share a common definition, though they differ in the value assigned to the variables[17].

Briefly, classes define the *type* of objects. Several meanings can be associated with class:

A class defines an object type or *intent*–the structure and behavior of objects of a particular type. The intent includes *structure*(that is, the attributes and relationships

in which objects having this type can participate) and *behavior*(that is, the methods associated with the type).

A class defines an *extent*–the set of objets that have a particular type. In some object-oriented database systems such as ObjectStore, such an extent has to be explicitly defined as an attribute in this class.The concept of classes is similar to the concept of abstract data types with some additional properties.

In an object-oriented database scheme, it is often the case that several classes are similar. It would be desirable to define a representation for the common variables of these classes in one place. To do so, we place classes in a specialization hierarchy, the *class hierarchy.* In the hierarchy, the child of a class is a subclass of this class. A subclass inherits all the attributes and methods of its superclass and can define its own attributes and methods. If a class inherits attributes and methods from only one class, this inheritance is called *single inheritance.* Otherwise, it is called *multiple inheritance.* In a system which supports single inheritance, the classes form a hierarchy, called a class hierarchy.

There exists another kind of hierarchy relating to classes, the *class composition hierarchy.* The fact that the domain of an attribute may be an arbitrary class gives rise to the nested structure of the definition of a class: a class consists of a set of attributes; the domains of some or all of the attributes may be classes with their own sets of attributes, and so on. This definition of a class results in a directed graph of classes rooted at that class, the class composition hierarchy. This hierarchy captures the specialization relationship between one class and its subclasses. We will give an example class hierarchy and an example class composition hierarchy shortly.

3.4 An Example Database

To illustrate the concepts we just introduced, we establish a database containing information of professors and graduate students in a graduate school. We first of all define a class called `Person`, and then define `Professor` and `Graduate` to be two *subclasses* of `Person`. There exists a *single inheritance* from `Person` to `Professor` and from `Person` to `Graduate`, that is, `Professor` and `Graduate` inherit attributes and methods from `Person`. Since class `Professor` contains a reference attribute, `Student`, there should be a link in the class composition hierarchy to reflect this. The class hierarchy and class composition hierarchy for the classes we defined are shown in Figure 3.1 and Figure 3.2 respectively. In the class hierarchy, the title on top of a box is the name of the class; the names inside a box are attributes defined for that class. A link between two boxes indicates that the lower class is derived from the upper class; the derived class inherits all the attributes and methods of the base class. In the class composition hierarchy, the names atop and inside a box bear the same meaning as in a class hierarchy. The link between an attribute in one box and the title of another box indicates that the value of this linked attribute is an object of the linked class.

Each of the three classes are defined as in Figure 3.3. We use a syntax similar to that of C++ and ObjectStore for the definition. In the definition, we assume `string` is a primitive type for character strings. The construct `Set<ElementType>SetName` defines a set named `SetName` whose elements are of `ElementType`; The construct `List<ElementType>ListName` defines a list named `ListName` whose elements are of `ElementType`. We omit all method definitions in our class definition.

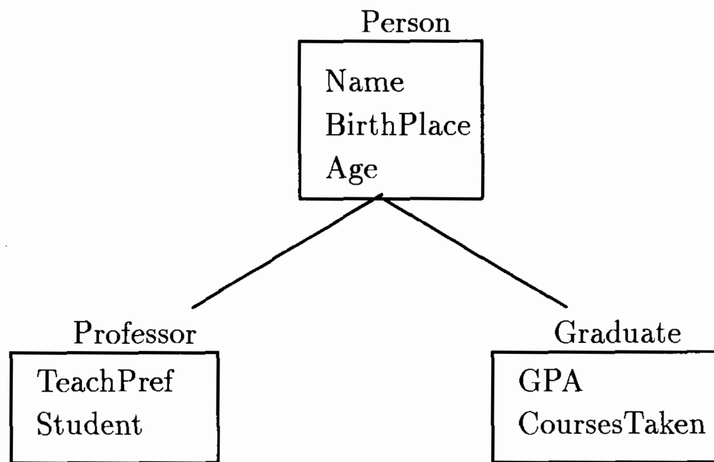

Figure 3.1: The class hierarchy

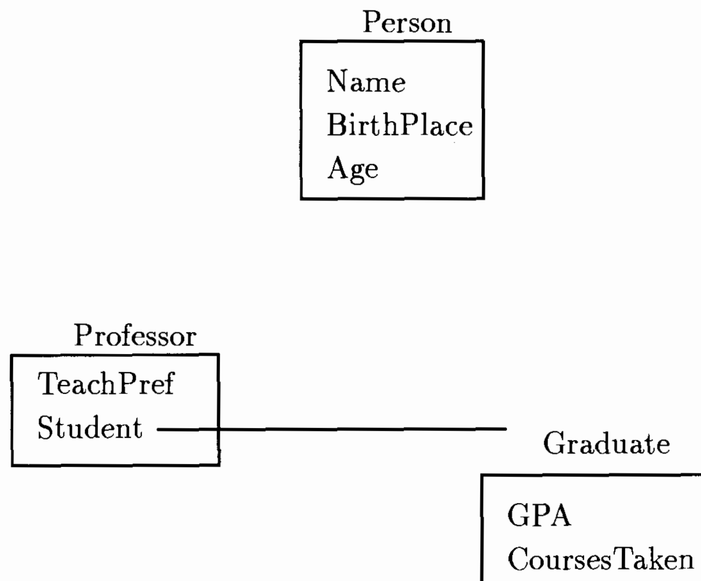

Figure 3.2: The class composition hierarchy

```
Class Person
{
protected:
  string Name;
  string BirthPlace;
  int  Age;
};

class Professor:public Person
{
  List<string> TeachPref;
  Set<Graduate*> Student;
};

class Graduate:public Person
{
  int  GPA;
  Set<string> CoursesTaken;
};
```

Figure 3.3: The class definition

Oid	Name	BirthPlace	Age	Salary	TeachPref	Graduate
p1	Ghandi	Bombay	45	39000	[600,800]	{g3,g4}
p2	Roberts	Ottawa	29	54000	[652,653]	{g1,g2}
p3	Gibson	Calgary	30	33000	[601,750]	{g5}
p4	Tanaka	Tokyo	49	49000	[800,801]	{g6}

Table 3.1: Instances of class `Professor`

The `TeachPref` attribute of class `Professor` indicates a professor's teaching preference, i.e., which courses he/she prefers teaching, and is a *list* of course numbers; we make `TeachPref` a list to differentiate among a professor's preferences. The `CoursesTaken` attribute of class `Graduate` is of *set* type since a student may have taken a number of courses, and there is no particular reason to order them. The `Student` attribute of `Professor` is a set of pointers to instances of class `Graduate`, reflecting the facts that a professor may supervise more than one graduate student and that a graduate student may be an instance of a class of arbitrary structure.

Now we can populate our example database with objects. Class `Person` is used as a *base* class only, therefore does not have any instance. Table 3.1 shows the instances of class `Professor`, and Table 3.2 shows the instances of class `Graduate` in our database. For brevity, we still list database objects in tables similar to how we list tuples in a relational table; the differences are, in addition to showing all the attribute-value pairs, we also show the object identities(*oid*s) and the values for reference attributes are oids. We use {} for enclosing elements of a set and [] for enclosing elements of a list.

The example database, though very simple, covers many of the OODB concepts we have discussed. The next chapter will examine the effects of OODB features on

Oid	Name	BirthPlace	Age	GPA	CoursesTaken
g1	Anderson	Vancouver	24	3.8	{800,600,803}
g2	Fraser	Ottawa	25	3.9	{600,601,603,801}
g3	Gupta	Bombay	29	3.3	{652,752}
g4	Liu	Shanghai	27	3.4	{650,751,752}
g5	Monk	Victoria	25	3.7	{651,652,653,752}
g6	Wang	Nanjing	28	3.2	{750,753,651,653}

Table 3.2: Instances of class `Graduate`

the attribute-oriented induction method and extend the relational attribute oriented induction method to object oriented data model in general.

CHAPTER 4

Attribute-Oriented Induction in OODBs

In this chapter, we extend the relational attribute-oriented induction method to object-oriented databases. We will study the impact of object-oriented features on the method and propose algorithms for the extension.

4.1 General Discussion

Before we begin discussing the impact of complex attributes and class inheritance to the induction, we take a look at the general procedure of induction in an OODB.

In a relational database, we start learning by collecting relevant data into a relational table using selection, projection, or join operations provided by the query

language; namely, we make a copy of the data portion in the database that are relevant to our learning task and the database remains intact. We should abide by the same principle in an OODB. However, in an OODB, instances are grouped into classes and related objects of different classes are connected through references, i.e., oids, which act as a counterpart of join in the relational sense. Attribute projection may not be supported in an OODB. Therefore, the "relevant data" returned from a query into an OODB is generally a set of oids pointing to instances of a specific class–the queried class. Relevant as well as irrelevant attributes of objects of this class can be accessed by dereferencing the oids in this set; attributes of related objects of other classes are accessed through reference attributes. For example, the query "select professors who are aged between 25 and 50" can be written in ObjectStore as

```
Set<Professor*> selectedProfs;
selectedProfs = Professor::extent[:(Age>=25) & (Age<=50):]
```

Here, `selectedProfs` is defined as `Set<Professor*>`, i.e., what is returned is a set of oids pointing to instances of class `Professor` whose ages are between 25 and 50. If we assume that the relevant instances are as in Table 3.1, then

```
selected_profs = {p1,p2,p3,p4}
```

After obtaining object instances relevant to our learning task, we may start the induction process. The basic strategies of attribute-oriented induction introduced in Chapter 2 are still applicable when the induction is conducted on an attribute whose values are of a primitive class. Since attribute projection may not be supported in an OODB, we encounter some technical differences here from in a relational database.

"Attribute removal" simply means that this attribute is no longer of interest to induction, and we can ignore its values when we interpret the induction result.

In the object-oriented database model, no two object instances are equal even if they have identical values for each of their attributes, because their oids will never be the same. The relational attribute-oriented induction method is value-based. In an object-oriented database, when we judge whether two object instances are "mergeable" in the relational sense, we only compare values in the relevant attributes, ignoring the oids. When checking whether two reference attributes have equal values, we see if the composing attributes of the pointed object instances are equal instead of comparing the oid values of the two reference attributes. Irrelevant attribute values are not compared. If two instances have identical values for each and every task-relevant attribute, one of them is considered redundant even though their oids are different. To "merge" two instances, we simply delete the oid of one of them from the class extent. Subsequent examples will demonstrate these. For brevity, we will not show values of irrelevant attributes; only values of relevant attributes and oids will be listed.

Example 4.1 *Characterize professors in relevance to* `Age` *and* `BirthPlace` *with threshold value of 2.*

Suppose the relevant instances of class `Professor` gathered is as in Table 3.1. In this learning task, only attributes `Age` and `BirthPlace` are relevant; other attributes are ignored. By assuming the concept hierarchy for `Age` in Figure 4.1 and the concept hierarchy for `BirthPlace` in Figure 4.2, we generalize on `Age` and `BirthPlace`. Similar to in relational databases, we replace each concept with its parent in the concept

```
{20--29}:twenties
{30--39}:thirties
{40--49}:forties
{50--59}:fifties
{60--69}:sixties
{twenties}:young_age
{thirties, forties}:mid_age
{fifties, sixties}:old_age
{young_age, mid_age, old_age}:ANY
```

Figure 4.1: Concept hierarchy for `Age`

```
{Burnaby, Richmond, vancouver, Victoria}:British Columbia
{Calgary, Edmonton}:Alberta
{Ottawa, Toronto}:Ontario
{Bombay}:India
{Shanghai, Nanjing}:China
{China, India}:Foreign
{British Columbia, Alberta, Ontario}:Canada
{Foreign, Canada}:ANY
```

Figure 4.2: The concept hierarchy for `BirthPlace`

hierarchy and delete redundant instances. After generalization, we get the result as shown in Table 4.1, which says that "we have young professors born in Canada and middle-aged professors born outside of Canada".

4.2 Complex Attributes

We will discuss the generalization of collection attributes and reference attributes in this section. In real applications, these two categories of complex attributes may

Oid	BirthPlace	Age
p2	Canada	young_age
p1	Foreign	mid_age

Table 4.1: Result for Example 4.1

be mixed. A collection attribute can have elements of pointers to another class; a reference attribute can point to a class consisting of collection attributes. We first discuss each category separately, and then present the integrated algorithm in the last section.

4.2.1 Collection Attributes

The common characteristic of *set*, *list*, and *array* is that they are *non-atomic*; they all have subcomponents. Each subcomponent, or element, of a collection attribute represents a concept itself, so a collection attribute can be regarded as a compound concept expressed in terms of a number of subconcepts. Strategy 2.2 in Chapter 2 states *Generalization should be performed on the smallest non-composable components of a data relation.* Here, though we are not dealing with a data relation, but a class having complex attributes, the *least commitment principle* is still valid. Therefore, we have

Strategy 4.1 *Generalization of a collection attribute should be performed on its composing elements and the generalized concept for the collection is expressed in terms of its generalized elements.*

```
{2.0--2.9}:average
{3.0--3.4}:good
{3.5--4.0}:excellent
{average, good, excellent}:ANY
```

Figure 4.3: The concept hierarchy for `GPA`

```
{600,601,602,603}:Theory
{650,651,652,653}:AI
{700,701,702,703}:Programming Languages
{750,751,752,753}:DB
{800,801,802,803}:Architecture
{Theory,AI,Programming Languages, DB, Architecture}:ANY
```

Figure 4.4: Concept hierarchy for `CourseTaken` and `TeachPref`

We first take a look at set attributes. Operators are provided for adding and deleting an element to and from a set. Two sets are equal when they contain identical elements. In our example database, the `CoursesTaken` of class `Graduate` is of set type; its value is a set of character strings. The following learning task involves a set attribute.

Example 4.2 *Characterize the graduates in relevance to* `CoursesTaken`*,* `BirthPlace`*, and* `GPA` *with threshold of 2.*

Assume the set of data relevant to this learning task is as shown in Table 3.2, and the concept hierarchies for `BirthPlace`, `GPA`, and `CourseTaken` are as shown in Figure 4.2, Figure 4.3, and Figure 4.4, respectively.

The domains of attributes `BirthPlace` and `GPA` are primitive classes, and their generalization can be done directly based on our discussion in the last section. `CourseTaken`

Oid	BirthPlace	GPA	CoursesTaken
g1	B.C.	excellent	{Architecture,Theory}
g2	Ottawa	excellent	{Theory,Architecture}
g3	India	good	{AI,DB}
g6	China	good	{AI,DB}

Table 4.2: After the first round of generalization

Oid	BirthPlace	GPA	CoursesTaken
g1	Canada	excellent	{Architecture,Theory}
g6	Foreign	good	{AI,DB}

Table 4.3: The learning result

is a set attribute and its generalization is unlike an attribute with a primitive class domain. By Strategy 4.1, we should generalize on each element of the set. We first replace each element in the set with its higher level concept by ascending the concept hierarchy by one level; we then eliminate redundant element(s) in the set caused by the generalization. This result is shown in Table 4.2. Since the number of instances is still greater than the threshold, further generalization is needed. We conduct one more round of generalization on `BirthPlace` in like manner and get the final result in Table 4.3.

The rule states that a graduate student is either a Canadian-born student with excellent GPA who has taken courses in Architecture and Theory or a foreign-born student with good GPA who has taken courses in AI and DB.

The next example illustrates induction on *list* attributes.

Oid	Age	TeachPref
p1	mid_age	[AI,DB]
p2	old_age	[Architecture,Programming Language]

Table 4.4: The final learned rule

Example 4.3 *Characterize professors in relevance to* `Age` *and* `TeachPref` *with threshold of 2.*

Suppose the concept hierarchy for `Age` is as as shown in Figure 4.1[2], and the concept hierarchy for `TeachPref` is as shown in Figure 4.4. By Strategy 4.1, we generalize on each element of it by ascending the concept tree. There are two different points in generalizing a list attribute: identical elements after generalization are not reduced to one, and concept substitution does not change the order of elements in the list. Two lists are equal only if their corresponding elements are equal pairwise. The learned rule is shown in Table 4.4. The learned rule indicates that a professor is either middle-aged and prefers teaching AI and DB courses or is old and prefers teaching Architecture and Programming Languages courses.

Some object-oriented database systems also allow *array* attributes. From the view point of attribute-oriented induction, an array can be treated as a multiple-dimensional list, and the induction method we discuss here regarding lists can be applied to arrays as well.

4.2.2 Reference Attributes

Reference attributes point to objects of another class that can be regarded as a concept grouping. The generalization of a reference attribute is therefore the generalization of subconcepts that make up the pointed class. Therefore, we can derive the generalization strategy as follows.

Strategy 4.2 *Generalization on a reference attribute is conducted on each composing attribute of the object class being pointed to.*

This is similar to how we handle collection attributes. However, since the objects being pointed to may be of a class that contains reference attributes as well, we may have to apply the above strategy several times. By the *least commitment principle*, generalization should start on the finest concepts in order to guarantee correctness. This means we may have to keep on dereferencing pointers until we reach a non-reference attribute. The class composition hierarchy contains the information that will guide this pointer dereference. When there exists a link from an attribute in one class to another class, and when this attribute is relevant to the learning task, this link(pointer) should be dereferenced, that is, generalization moves on to the referenced class. No more dereferencing is needed when we have come down to a leaf class in a class composition hierarchy.

Example 4.4 *Characterize professors in relevance to their* `BirthPlace`, `TeachPref`, *and* `Student` *with threshold of 2.*

Again, we assume the relevant instances of class `Professor` and class `Graduate` are as shown in Table 3.1 and Table 3.2. Here, `Student` is a reference attribute; it

Oid	BirthPlace	TeachPref	Student
p4	Canada	[Architecture,ProgrammingLang]	{g1,g2,g5}
p2	Foreign	[AI,DB]	{g3,g4,g6}

Oid	BirthPlace	GPA	CoursesTaken
{g1,g2,g5}	ANY	excellent	{Arch., ProgLang}
{g3,g4,g6}	Foreign	good	{AI,DB}

Table 4.5: the final result

contains pointer(s)(oids) to instances of class `Graduate`. Each instance of `Graduate` is further decomposable to several components, so generalization on attribute `student` equals to generalization on `BirthPlace`, `GPA`, and `CoursesTaken` of class `Graduate`. The final learned rule is shown in Table 4.5.

The learned rule says that "Canadian-born professors prefer to teach courses in Architecture or Programming Languages and their students have excellent GPA and have taken courses in Architecture and Programming Languages; Foreign-born professors prefer to teach courses in Artificial Intelligence or Databases and their students have good GPA and have taken courses in Artificial Intelligence and Databases".

The above discussion has assumed that the class composition hierarchy does not contain a cycle. In some applications, the class composition hierarchy may contain cycles. For example, in our example database, if we add an attribute `ThesisExaminer` of class `Professor` to class `Graduate` to refer to the professor who acts as the thesis examiner of a student, the resultant class composition hierarchy will contain a cycle, as shown in Figure 4.5.

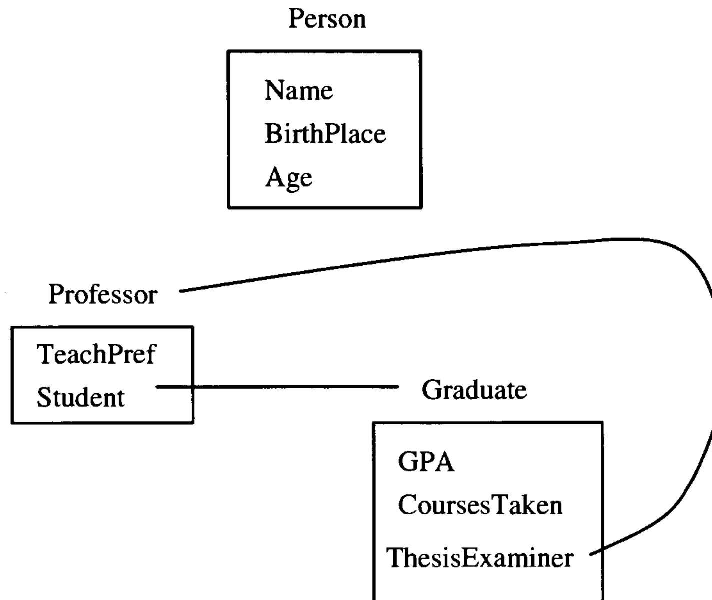

Figure 4.5: Class composition hierarchy with a cycle

The cycle exists due to the fact that a reference attribute of a class indirectly references this same class. If we conducted induction on a reference attribute in a cycle by simply dereferencing the pointers, we would get into an infinite loop. Therefore, a cycle has to be broken at a certain point such that the dereferencing will terminate. There can be several ways to limit the depth of deferencing a cycle. To get interesting knowledge rules without too much complexity, we would desire to trace the full length of a cycle but not to dereference the same class twice. We propose the following simple and effective method to handle class-composition hierarchy with a cycle. We start out a learning process with a certain class in the hierarchy and dereference pointers as we normally do until we encounter a reference attribute which points to the class we started from. We stop dereferencing any further, instead, we conclude the dereferencing by describing the last reference attribute using only

the non-referencial attributes of the class pointed to. Let us examine the query in Example 4.4 again, this time assuming attribute `ThesisExaminer` has been added to class `Graduate`.

The dereferencing of attribute `Student` of class `Professor` leads the induction to all attributes of class `Graduate`. Since attribute `ThesisExaminer` of `Graduate` is again a reference attribute that points back to class `Professor`, forming a loop, we break the loop by describing `ThesisExaminer` using the non-referential attributes of class `Professor` only, that is, every attribute except `Student`.

The above method to deal with cycled class-composition hierarchy is simple and also effective. Tracing the full length of the cycle path captures all the details that may be needed to describe the queried class of instances. In case a deeper description is desired, a separate query can be issued.

4.3 The OOLCHR Algorithm

We have demonstrated our solutions to perform generalization on complex attributes, and a formal presentation of the algorithm is in order. The **OOLCHR** algorithm is an extension of the relational **LCHR** algorithm and can conduct generalization on complex attributes. The algorithm and accompanying procedures are listed in Figure 4.6 and Figure 4.7. In the algorithm, '*A' refers to the object class pointed to by 'A'.

The algorithm is based on the strategies we discussed in the previous section. After task-relevant instances have been selected, the procedure **OOinduction** is called.

```
ALGORITHM OOLCHR;
INPUT: (1) an object-oriented database,
       (2) a set of concept trees,
       (3) the learning task, and
       (4) the threshold value(T).
OUTPUT: A characteristic rule learned from the database.
BEGIN
  Step 1. Select the task-relevant instances into set S.
  Step 2. Call Procedure OOinduction(S,T).
  Step 3. Transform the final relation into a predicate formula.
END
```

Figure 4.6: The **OOLCHR** Algorithm

This procedure processes each relevant attribute such that the attribute threshold is satisfied. It then goes on to ensure that the final learned result contains no more instances than the threshold dictates. For each relevant attribute can be simple as of a primitive class, or complex as of a set, a list, an array, or a reference, **OOinduction** has to treat each attribute according to its type; this is done by the subprocedure **Resolve(A)**. The element in a collection attribute can again be a complex class, and the attributes of the class pointed to by a reference attribute may also be of a complex class. This gives **Resolve(A)** a recursive nature. Here, by primitive, we mean the system defined classes such as integer, float, char, etc, or a user- defined class that does not belong to a complex class. Since complex attributes are defined using primitive classes and can always break down to primitive classes, and we have introduced a mechanism to break cycles in class composition, **Resolve(A)** is guaranteed to terminate. Moreover, since all the extensions we made strictly follow the *least commitment principle*, the correctness of our algorithm is also guaranteed.

```
PROCEDURE Resolve(A);
/* The argument A is an attribute name on which generalization
 * is done in this procedure */
BEGIN
  CASE A OF
  Reference: IF (*A has not been dereferenced)
             THEN FOR (each attribute of the class) DO Resolve(attribute)
             ELSE FOR (each non-reference attribute) DO Resolve(attribute);
        Set: FOR (each element of A) DO Resolve(element);
             Delete repetitive elements from A;
       List: FOR (each element of A) DO Resolve(element);
  Primitive: FOR (each instance in set S) DO
               Replace the value of attribute A by its
                 parent concept in the concept tree;
             Delete repetitive instances in set S;
END

PROCEDURE OOinduction(S, T);
/* S is the set of instances of some class relevant to the learning task.
 */
BEGIN
  FOR (each task-relevant attribute A of the class) DO
    WHILE (the number of distinct values of A > T) DO
      IF there does not exist a concept tree for A
      THEN mark A as ``irrelevant'' and exit
      ELSE call Resolve(A);
  /* now the threshold constraint is satisfied by each attribute */

  WHILE |S| > T DO
    BEGIN
      Select the attribute containing substantially more distinct
      values or with a better reduction ratio, and replace each
      value of them by its corresponding parent in the concept
      tree, and eliminate redundant instances in S;
    END
  /* now the threshold constraint is satisfied by the instance set */
END
```

Figure 4.7: The **OOinduction** procedure

CHAPTER 5

General Improvement of the Induction Method

In the attribute-oriented induction method, the concept hierarchy plays an important role; it is the place where knowledge is embedded. Whether the concept hierarchy is generated automatically or is provided by a human expert, how to make the best use of it has a marked effect on the learning efficiency and the learning result.

So far in our discussion, we have assumed that all concept trees are balanced trees and all of the primitive concepts reside at the same level at start. By primitive, we mean the concepts(attribute values) that appear in the database. The assumptions were necessary for *synchronous* generalization on each attribute. Namely, we have assumed that every possible domain value of the attribute has a corresponding leaf node in the concept tree and that all the leaf nodes reside at the same depth from the root. With each round of generalization, the substitution of a higher level concept for

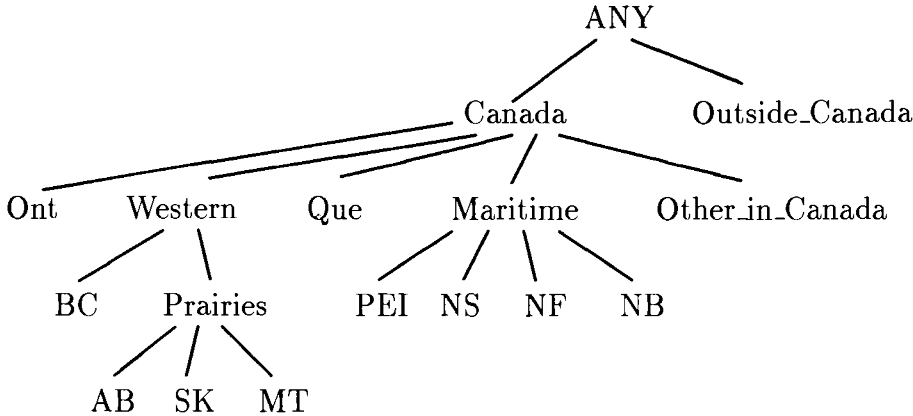

Figure 5.1: An imbalanced concept tree

a concept residing at the current induction level is synchronous with respect to the concepts to be generalized. Each raw concept is treated uniformly throughout the induction process.

However, in some applications, we may encounter concept hierarchies that do not satisfy this assumption; we may have to perform generalization by an imbalanced concept tree. Such a tree is given in Figure 5.1. If we still use the synchronous ascension strategy, we may mingle concepts residing at different levels and result in an incorrect generalization. For instance, if we synchronously ascend the concept tree in Figure 5.1 by one level, "Ont" would be replaced by "Canada", and "AB" by "Prairies". The result thus contains both "Canada" and "Prairies", and since "Canada" logically contains "Prairies", the generalization is incorrect. We therefore need a new ascension strategy.

In order to ensure induction correctness, we need a concept hierarchy that provides us with more information in a more explicit way. We propose an enhancement of the concept hierarchy in which some statistics of the tree is calculated and attached to the tree before generalization is conducted. First, we associate a *conceptual level*

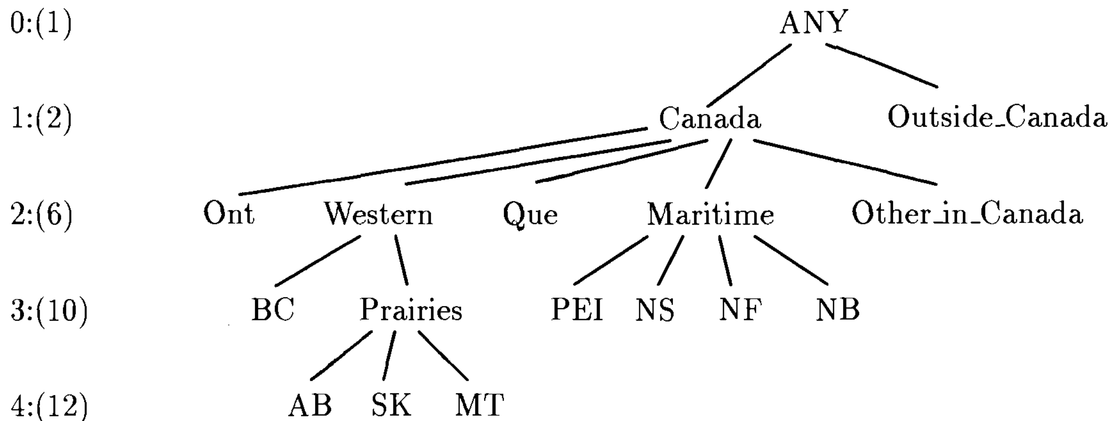

Figure 5.2: The enhanced concept tree

number to each and every node of the tree, which equals to the depth of this node relative to the root. So the root is at level 0, children of the root are at level 1, etc. Second, we associate each conceptual level with an integer value which denotes the number of compatible concepts at this level. Mathematically, this number equals to the number of nodes at this level plus the number of leaf nodes above this level. The calculation can be carried out once a concept tree is formed. In the event that the concept hierarchy should be dynamically adjusted according to the learning task, the calculated information is updated as well. The enhanced version of Figure 5.1 is shown in Figure 5.2. The meaning of *compatibility* is explained in Section 5.1.

5.1 Maintaining Generalization Correctness

As has been mentioned, the synchronous tree ascension method does not work well on imbalanced trees and will result in incorrect rules. For a learned rule to be logically sound, it must be expressed in terms of *compatible* concepts. Two concepts are *compatible* if they can appear in a generalized result at the same time and the result

makes sense. Let us examine the compatibility of concept "Ont" with any other concepts in the concept tree of Figure 5.2. On the one hand, "Ont" represents one of the many provinces and appears in the database before generalization as a raw datum, so it is naturally compatible with other province names represented as leaf nodes in the tree. On the other hand, due to many factors, "Ont" plays a more important role in the grant information database and attracts more funds than does the three maritime provinces combined. From the statistical point of view, we also want "Ont" to be compatible with "Maritime"; this is where human knowledge(bias) can influence the learning result. In like manner, we want "Ont" to be compatible with "BC", "Prairies", "Other_in_Canada", and "Outside_Canada". To generalize, we state that a leaf node concept is compatible with any other concept residing at the same level or at a lower level. The method for calculating the number of compatible concepts of each level we mentioned previously captures this concept.

To guarantee that the learned rule is always expressed in terms of compatible concepts, we can modify the existing synchronous ascension method or we can design a new ascension control mechanism. We discuss both scenarios. Cai presented a method following the former approach [2]. This method does a compatibility check and adjustment after the generalization by seeing whether one generalized concept covers other concepts of the same attribute. Concept A covers concept B if concept B is a descendent of concept A in the concept tree. If one generalized concept covers another concept, that is, the two concepts are not compatible due to their logical containment relationship, the covered concept is then replaced by the generalized concept.

There are two drawbacks to this method. It entails computation to check the

compatibility, and it may lead to over-generalization. To see how over-generalization may occur, let us take the concept tree in Figure 5.2 as an example. Because the synchronous induction treats every raw datum equally by replacing it by its parent concept in the tree and "Ont" is a raw datum, "Ont" will be replaced by "Canada" after the first round of generalization. Since "Canada" covers all concepts except "Outside_Canada", Cai's method will adjust the generalization result by substituting "Canada" for any concept other than "Outside_Canada", and results in an over-generalized rule that contains "Canada" and "Outside_Canada" only.

We now look at the another approach to ensuring the coherence of the learning result–the *asynchronous ascension method.* We have seen from the above example that logical inconsistency stems from the synchronousness in treating each raw datum. In an imbalanced tree, raw concepts reside at different levels and so deserve different treatment; this is the idea of asynchronous ascension. In this method, we examine the location of each concept prior to ascension. Depending on its location, a concept is substituted for by its parent, or remains unchanged in the current round of generalization. The ascension of all concepts is asynchronous.

Now that we have enhanced the concept tree with conceptual level numbers, the implementation of the asynchronous ascension is easy. We demonstrate the method using the same concept tree as in Figure 5.2. In our example concept tree, "Canada" will be at conceptual level 1, "Western" at 2, "Prairies" at 3, "Alberta" at 4, etc.. We use a counter to record the current conceptual level which is initialized to be the maximum depth of the concept tree. With each ascension, we only substitute for a concept whose conceptual level number is the same as the current conceptual level and leave it intact otherwise. The current conceptual level is decremented after each

```
PROCEDURE Async(c1)
/* c1 is the concept to be generalized. The procedure returns the
 * generalized concept
 */
BEGIN
  Search the concept tree for concept c1, its conceptual level
  number c1_level, and its parent concept c2;
  IF (c1_level == CurrentLevel)
  THEN return(c2)
  ELSE return(c1);
END
```

Figure 5.3: The **Async** procedure

round of ascension. The induction goes on until a desired level is reached. In this example, at the first round of generalization, the current conceptual level is 4, so we should only replace concepts that reside at level 4 with their corresponding parent concept at level 3. Therefore, concepts such as “SK”, “MT”, and “AB” are replaced by “Prairies”. Concepts like “Ont.”, “BC”, “NF”, etc. not residing at the current conceptual level are not effected. “BC”, “Prairies”, “PEI”, etc., are substituted for only when a second round of generalization is conducted.

In a nutshell, the enhanced concept hierarchy allows lower leveled concepts to “catch up” during generalization, and disallows concepts at different levels to be generalized at the same time. Compared with Cai's method, our method not only guarantees induction correctness but also has a lower complexity because the expensive coverage check is eliminated. The method is summarized in the **Async** procedure below in Figure 5.3.

5.2 Induction Speedup

The induction algorithm proposed in [2] suggests that ascension of the concept tree be done one level at a time. For learning tasks requiring a high degree of generalization, this single-step ascension has to repeat several times. Precious computation time can be saved if the ascension of multiple levels is done in a single move. [14] furthers [2]'s study in that multiple-level ascension is introduced into the algorithm. Before generalization on an attribute, a *minimal desirable level* is computed and then each values in the attribute is replaced by its corresponding superordinate concept in the concept hierarchy at the minimal desirable level. The enhanced hierarchy we proposed in the last section provides an elegant way to implement the multiple level ascension. The following paragraphs discuss it in detail.

For ease of discussion, we define the *start level* of the induction to be the level with the largest number in the tree, the *finish level* to be the level where induction is supposed to end, and the *induction depth* to be their difference. The ascension process yields maximal efficiency when the ascension of *induction depth* levels is done in one move. In order not to over-generalize, it is imperative to determine at which level ascension should stop prior to the generalization process. The generalization result obtained by multiple level ascension should equal to that obtained by stepping up the concept tree one level at a time.

We have used a threshold value as the sole criteria to decide when to stop further generalization. The threshold method is easy and simple for single-level ascension: before each round of ascension, the number of distinct values of an attribute is compared with the threshold to determine whether or not ascension should be conducted.

Since only one level is climbed up the concept tree after each threshold comparison, there is never the risk of over-generalization. The number of distinct attribute values left after some generalization is a good indicator of whether a certain degree of generalization has been reached and the threshold method is simple, we therefore want to extend it for multiple level ascension control as well.

We want to determine the finish level from the threshold value and knowledge coded in the concept tree. This has become easy now that the enhanced concept tree registers the number of compatible concepts allowed at each level in the tree; this number indicates how many distinct attribute values are left when the induction stops at this level[1] The finish level can be easily determined by identifying the level whose number of compatible concepts is no greater than the threshold. With the finish level known, only concepts residing between the start level and the finish level need be generalized, and the ascension of the concept tree can be done in one move, that is, we substitute for each raw value of this attribute needing generalization by its ancestor residing at the *finish level.*

Let us work through an example to see how the multiple level ascension works. Suppose the learning task is "*Learn characteristic rules regarding grant recipients in relevance to geographic distribution with threshold of 6*". We assume the enhanced concept hierarchy for the `Province` attribute is as shown in Figure 5.2. Using the asynchronous single step ascension technique discussed in the previous section, we

[1]If the database does not include some leaf concept(s), when induction stops the number of distinct values may be smaller than the number of compatible concepts at the finish level. However, this is a rare situation in a very large database, especially when the concept hierarchy is generated automatically from the raw data.

would first replace concepts "AB", "SK", and "MT" by "Prairies" and do bookkeeping such as merging identical instances. Since the number of distinct values for this attribute is still greater than the threshold value of 6, the second round of ascension is initiated and concept "Prairies" is replaced by "Western" and concept "PEI" is replaced by "Maritime", etc. By now, the only possible distinct values for `Province` are "Ont", "Western", "Que", "Maritime", "Other_in_Canada" and "Outside_Canada", and this number is no greater than the given threshold. Therefore, the induction process stops at this level, and the learning task is finished.

Using the multiple level ascension method, we first determine the finish level by comparing the number of compatible concepts associated with each level with the threshold, and it is easy to find the level whose number of compatible concepts is no greater than the threshold value; this level is the finish level. In this example, the finish level can be identified as level 2. Since induction starts at the raw concepts residing at level 4, so the induction depth is calculated to be 2. During the one-move ascension, we replace each raw concept residing between level 3 and level 4 inclusive by its ancestor residing at level 2. "AB" will be replaced by "Western", "BC" by "Western", "PEI" by "Maritime", etc. We get the same induction result as that obtained by single level ascension. To make our method more robust, we can integrate this multiple-level ascension method and the asynchronous ascension control method developed in the previous section. Instead of comparing the level number of a concept being considered with the current conceptual level number, we simply see if the concept resides below the finish level. Concepts residing below the finish level are substituted for by their ancestors at the finish level; otherwise, these concepts are not changed. This integrated algorithm, **Async*** is formalized as follows in Figure 5.4.

```
PROCEDURE Async*(c1)
/* c1 is the concept to be generalized. The procedure returns the
 * generalized concept
 */
BEGIN
  Search the concept tree for concept c1, and its conceptual level
  number c1_level;
  IF (c1_level > FinishLevel)
  THEN retrieve ancestor c2 of c1 residing at FinishLevel; return(c2)
  ELSE return(c1);
END
```

Figure 5.4: The **Async*** procedure

In the discussion so far, we have assumed that the concept hierarchy remains constant throughout the induction. In some learning situations, the concept hierarchy has to be re-structured according to the specific query in order to make the learning result more meaningful[14]. In this case, the *finish level* cannot be determined before the query is processed and single level ascension is still needed. However, as soon as the concept hierarchy becomes stabilized after some adjustment, the *finish level* can be determined and multiple-level ascension can still be utilized.

The multiple ascension strategy is more time efficient than the single ascension strategy in that only one visit to the concept hierarchy is needed for each attribute value and that merging identical instances is performed only once. This reduction of overhead is significant when the number of instances is huge, which is the norm for a very large database.

```
ALGORITHM OOLCHR*;
INPUT: (1) an object-oriented database,
       (2) a set of concept trees,
       (3) the learning task, and
       (4) the threshold value(T).
OUTPUT: A characteristic rule learned from the database.
BEGIN
  Step 1. Select the task-relevant instances into set S.
  Step 2. Call Procedure OOinduction*(S,T).
  Step 3. Transform the final relation into a predicate formula.
END
```

Figure 5.5: The **OOLCHR*** Algorithm

5.3 The OOLCHR* Algorithm

The algorithms we developed in this chapter improve the attribute-oriented induction method in general and are not database model dependent; they can be used in the relational induction as well as in the induction in object-oriented databases we discussed in Chapter 4. Here we add our newly designed asynchronous multiple-level ascension method to the **OOLCHR** algorithm presented in Chapter 4, and the new algorithm, **OOLCHR*** is complete for performing learning characteristic rules from databases where complex attributes exist.

```
PROCEDURE Async*(c1)
/* c1 is the concept to be generalized. The procedure returns the
 * generalized concept */
BEGIN
  Search the concept tree for concept c1, and its conceptual level
  number c1_level;
  IF (c1_level > FinishLevel)
  THEN retrieve ancestor c2 of c1 residing at FinishLevel; return(c2)
  ELSE return(c1);
END
PROCEDURE Resolve*(A);
/* The argument A is an attribute name on which generalization
 * is done in this procedure */
BEGIN
  CASE A OF
  Reference: IF (*A has not been dereferenced)
     THEN FOR (each attribute of *A) DO Resolve(attribute)
     ELSE FOR (each non-reference attribute) DO Resolve(attribute);
        Set: FOR (each element of A) DO Resolve(element);
             Delete repetitive elements from A;
       List: FOR (each element of A) DO Resolve(element);
  Primitive: FOR (each instance in set S) DO
               Replace the value c1 of attribute A by Async(c1);
             Delete repetitive instances in set S;
END
PROCEDURE OOinduction*(S, T);
/* S is set of instances of class relevant to learning task*/
BEGIN
  FOR (each task-relevant attribute A of the instance class) DO
    WHILE (the number of distinct values of A > T) DO
      IF there does not exist a concept tree for A
      THEN mark Ai as ''irrelevant'' and exit
      ELSE call Resolve(A);
  WHILE |S| > T DO
      Select the attribute containing substantially more distinct
      values or with a better reduction ratio, and replace each
      value of them by its corresponding parent in the concept
      tree, and eliminate redundant instances in S;
END
```

Figure 5.6: The **OOinduction*** procedure

CHAPTER 6

Concluding Remarks

6.1 Conclusions

Learning from examples is a very useful learning technique in artificial intelligence and is the theoretical foundation of many learning algorithms for knowledge discovery in databases. The attribute-oriented induction method for excavating data regularities and rules from databases is computationally superior to many other learning algorithms in that it factors down the version space. To meet the ever increasing demand of real world applications, the new generation database systems such as the object-oriented database systems have been gradually adopted, and it is of significance to extend knowledge discovery methods designed solely with the relational database in mind to handle learning in object-oriented databases.

In this thesis, we examined features of an object-oriented database and their effects on the attribute-oriented induction method and then proposed strategies to apply

the induction technique in the presence of complex attributes. The algorithm we presented, **OOLCHR**, can learn characteristic rules from object-oriented databases having complex attributes such as sets, lists, and arrays.

To guarantee that the generalization process maintains logical soundness, a mechanism for deciding which concepts should be replaced by their ancestors and which concepts should be left unchanged was also proposed, making use of the enhanced concept hierarchy. Interestingly, the enhanced concept hierarchy also provides an elegant way to implement the multiple level tree ascension proposed in [14]. The two improvements were formalized in the **Async*** algorithm. The algorithm works on an enhanced concept tree that contains more information about the attribute domain.

Finally, we presented a complete algorithm for learning characteristic rules from an object-oriented database, the **OOLCHR*** algorithm. It integrates the general improvement to the induction method and the special treatment of complex attributes.

Since learning in relational databases has a longer history and some of the techniques have been quite mature, a question in order would be whether or not we could convert learning in an OODB to that in a relational database? Our study has indicated that it is practible to convert a learning task in an OODB to that in a relational database and has provided transformation methods. This conversion can be significant if we consider that presently available commercial relational databases are still superior to commercial object-oriented databases in terms of performance. It may well be worthwhile to conduct a learning task in a relational database and then have the learning result converted.

Theoretically, however, doing learning directly in an OODB has advantages unparalleled by that in a relational database. First, as we have pointed out, a real-world entity is directly represented as an object, and complex objects are directly supported by an OODB, while in a relational database, not all complex object can be directly represented and sometimes complicated counter-intuitive data structures have to be used. That is to say that sometimes for some applications the OODB model has to be used to achieve design clarity and understandability. Secondly, an OODB differs from a relational database in that it significantly reduces the so-called *impedence mismatch*(explained in the next section). This reduction is expected to boost the performance of OODBs substantially over relational databases. Our study presented a way to conduct learning in an OODB directly without the need for conversion which may be time-costly.

6.2 Future Research

To date, there has been little report on knowledge discovery methods specially designed for object-oriented databases. Undoubtedly, many existing learning algorithms are model-independent and can also be used for object-oriented databases. However, since object-oriented database systems have many advanced features that other database systems lack, a direct application of these algorithms may not take full advantage of these features for maximal performance. One of the good features of object-oriented database systems is the reduction of *impedance mismatch* that relational database systems suffer.

Take the attribute-oriented induction method for example. Since some operations

in the learning process such as *collect task relevant tuples*, *dropping an attribute*, and *merging identical tuples* appear to be the select, project, and join operation or a combination of them, it may seem that if the attribute-oriented induction method is used in a relational database system, we may make use of the well-optimized set operations provided by the query language to make the learning process computationally efficient. This is hardly the case in reality. Since the query language is usually incomplete as a programming language and implementation of the learning algorithm needs the capabilities of a programming language, generally both a query language and a programming language are needed to implement the learning algorithm in a relational database system. The performance of the learning may suffer greatly due to the *impedance mismatch* between the two languages. The following paragraphs explain what *impedance mismatch* is.

The process of knowledge discovery in databases can be regarded as an application of the database. The implementation of the learning procedure usually has to be written in a programming language. The query language, as is provided with relational database systems, is not complete as a programming language and therefore cannot adequately implement the learning algorithm. This means that for relational database systems, two languages, a query language and a programming language, must be used to realize any learning activities. Relational database systems generally provide a declarative query language with no control constructs, variables, or other programming features. The programmer writes applications in a programming language with its own data structures, and uses the query language to transfer data back and forth between the program data environment and the database data environment. The existence of two environments can be a problem in that it is necessary to translate data between the two representations in a database application, and that

the transfer of data or of control between the two environments can be a performance problem[7]. The problems with using two language environments have been collectively called the *impedance mismatch* between the application programming language and the database query language.

On the other hand, an object-oriented database system provides a *database language* that includes a query language plus programming and other capabilities as well. The database language provides only one execution environment, procedural language, and type system. The most important advantage afforded by database languages is a reduction of the impedance mismatch. With this unique feature, object-oriented database systems provide a better environment for conducting learning activities, a very important application of a database.

It is hoped that the attribute-oriented induction method can perform better in an object-oriented database because of the reduction of the impedance mismatch. However, further study regarding the advanced features of object-oriented systems and the implementation issues need to be conducted to reach a conclusion. In this thesis, we examined only the influence of complex attributes on the attribute-oriented induction method in general terms. Study into the other features such as the class inheritance will also be an interesting topic.

REFERENCES

[1] T. M. Anwar, S. B. Navathe, H. W. Beck, Knowledge Mining in Databases: A Unified Approach Through Conceptual Clustering. Database Research and Development Center, Computer and Information Sciences, University of Florida, Fainesville, Florida. November 4, 1991.

[2] Yandong Cai, *Attribute-Oriented Induction in Relational Databases.* MSc thesis, Simon Fraser University, Dec. 1989.

[3] Yandong Cai, Nick Cercone and Jiawei Han, Learning Characteristic Rules from Relational Databases, in F. Gardin and G. Mauri(eds.), *Computational Intelligence, II*, Elsevier Science Publisher B.V. (North-Holland), 1990, 187-196

[4] Yandong Cai, Nick Cercone, and Jiawei Han, Attribute-Oriented Induction in Relational Databases, *Proceedings of IJCAI-89 Workshop on Knowledge Discovery in Databases*, Detroit, Michigan, August 1989, 26-36.

[5] Yandong Cai, Nick Cercone, and Jiawei Han, An Attribute-Oriented Approach for Learning Classification Rules from Relational Databases, *Proceedings of the 6th International Conference on Data Engineering*, Los Angeles, CA, Feburary 1990, 281-288.

[6] Yandong Cai, Nick Cercone, and Jiawei Han, Learning in Relational Databases: an Attribute-Oriented Approach, *Comput. Intell.* 7, 119-132(1991).

[7] R.G.G.Cattel, *Object Data Management.* Addison-Wesley Publishing Company, 1991.

[8] Thomas G. Dietterich, Ryszard S. Michalski, Inductive Learning of Structural Descriptions: Evaluation Criteria and Comparative Review of Selected Methods, *Artificial Intelligence*, 16(1981) pp257-294.

[9] Thomas G. Dietterich, Ryszard S. Michalski, A comparative Review of Selected Methods for Learning from Examples, *in Michalski et. al. (eds.), Machine Learning: An Artificial Intelligence Approach, Vol. 1, Morgan Kaufmann, 1983, 41-82.*

[10] W. J. Frawley, G. Piatetsky-Shapiro, C. J. Matheus, Knowledge Discovery in Databases: An Overview, in G. Piatetsky-Shapiro and W. J. Frawley (eds.), *Knowledge Discovery in Databases*, AAAI/MIT Press, 1991, 1-27.

[11] Jiawei Han, Yandong Cai, and Nick Cercone, Knowledge Discovery in Databases: An Attribute-Oriented Approach, *Proceedings of the 18th VLDB Conference*, Vancouver, British Columbia, Canada, 1992, 547-559.

[12] Jiawei Han, Yue Huang, and Nick Cercone, *Intelligent Query Answering Using Discovered Database Knowledge.* School of Computing Science, Simon Fraser University. Canada.

[13] Jiawei Han, Yandong Cai, and Nick Cercone, Discovery of Quantitative Rules from Large Databases, in Z. W. Ras, M. Zemankova, and M. L. Emrich(eds.), *Methodologies for Intelligent Systems, vol. 5*, Elsevier Science Publishing Co., Inc., 1990, 157-165

[14] Jiawei Han, Yandong Cai, Nick Cercone and Yue Huang. DBLEARN: A knowledge Discovery System for Large Databases. Proceedings of the ISMM International Conference, Information and Knowledge Management CIKM-92, Baltimore, MD, USA, November 8-11, 1992, 473-481.

[15] Min Ke, Moonis Ali, A Knowledge-Directed Induction Methodology for Intelligent Database Systems, *International Journal of Expert Systems*, Vol. 4, No. 1, 1991, 71-115.

[16] Won Kim, Object-Oriented Databases: Definition and Research Directions, *IEEE Transactions on Knowledge and Data Engineering*, Vol2, No.3, September 1990, 327-341.

[17] H. F. Korth and A. Silberschatz, *Database System Concepts*, 2nd ed., McGraw-Hill, Inc., 1991.

[18] Ryszard S. Michalski and Yves Kodratoff, Research in Machine Learning: Recent Progress, Classification of Methods, and Future Directions, *Machine Learning: An Artificial Intelligence Approach*, Vol. 1, Morgan Kaufmann, 1983, chapter 1.

[19] G. Piatetsky-Shapiro, Discovery, Analysis, and Presentation of Strong Rules, in G. Piatetsky-Shapiro and W. J. Frawley (eds.), *Knowledge Discovery in Databases*, AAAI/MIT Press, 1991, 229-248.

[20] A. Silberschatz, M. Stonebraker, and J. Ullman, Database Systems: Achievements and Opportunities, *Comm. ACM*, 34(10), 1991, 94-109.

[21] W. Ziarko, The Discovery, Analysis, and Representation of Data Dependencies in Databases, in G. Piatetsky-Shapiro and W. J. Frawley (eds.), *Knowledge Discovery in Databases*, AAAI/MIT Press, 1991, 195-212.