A DATABASE MANAGEMENT SYSTEM TO SUPPORT THE INSTANCE-BASED DATA MODEL: DESIGN, IMPLEMENTATION, AND EVALUATION

JIANMIN SU

A Database Management System to Support the Instance-based Data Model: Design, Implementation, and Evaluation

by
Jianmin Su

A thesis submitted to the School of Graduate Studies
in partial fulfillment of the requirements for the degree of Master of Computer Science

Department of Computer Science
Memorial University

May 2003

St. John's Newfoundland

Abstract

The instance-based data model (IBDM) was recently proposed by Parsons and Wand [2000]. One of the key contributions of this model is class independence. The IBDM separates the storage of information about instances from the organization of a schema using classes. This thesis transforms the IBDM from concept to real world implementation. It develops and analyzes possible base data structures for implementing an instance-based DBMS, and creates a flexible query language iQL (instance-based query language) for the data model. Two proofs of concept DBMSs are implemented (one for each of two base data structures) in order to test and verify the analytical results of the research. Also, the instance-based database model is compared with the relational database model to obtain a clear understanding of the advantages provided by the instance-based database model.

Acknowledgment

First I want to thank Dr. Jeffrey Parsons for being my advisor. Under his guidance, I successfully overcame many difficulties and learned a lot about data modeling. I still remember the time when I began my research in Fall 2001. In that semester, we communicated every week. In each meeting, he explained my questions patiently, and I felt my quick progress from his advise. I am also thankful that he spent time to proofread this thesis to reduce my mistakes in grammar.

Second I want to thank Ms. Elaine Boone. She always helped me a lot, either in my study or my life. Also, I want to thank Ms. Jane Foltz for she, as a head of the department, kindly helped me.

Special thanks to Dr. Krishnamurthy Vidyasankar, Dr. Manrique Mata-Montero, Dr. CaoAn Wang, Dr. Siwei Lu, Dr. Paul Gillard, and Dr. Miklos Bartha. With their kind help, I learned much during my program.

Thanks to all computer science staff. Without their help, I could not finish my program.

Finally, I want to thank my family, especially my wife and my son. Without their endless support and love for me, I would never have achieved my current position. I wish they live happily always.

TABLE OF CONTENTS

List of Tables

List of figures

Chapter 1

Introduction

Database theory has developed with the growth of computing applications. Database systems have evolved through three main generations. They are hierarchical database systems, network database systems, and relational database systems. Since Codd's seminal article [Codd 1970], database theory and implementation has evolved significantly. Many database models have appeared since that time. The Relational Data Model is the basis of most commercial database management systems (DBMS) currently in use. In addition, in the past decade, a number of object-oriented (OO) database languages have been proposed (e.g., [Maier 1986] [Albano, Ghelli, and Orsini 1995]). However, OO DBMSs have yet to achieve wide spread adoption in practice.

Most of the data models that underlie today's DBMSs are *class-based*, which means that the data models assume that the instances (e.g., tuples or objects) must belong to classes. In a class-based model, information about instances is stored according to the classes to which they belong. This leads to two categories of problems—either related to design or related to implementation. The design problems include: multiple classification problem, view integration problem, schema evolution problem, and interoperability problem. The implementation problems (operation problems) include: handling exceptional instances problem, reclassifying instances problem, adding and removing

instances problem, removing a class problem, and redefining a class problem [Parsons and Wand 2000].

The two layered data model (referred to in this thesis as the "instance-based data model") [Parsons and Wand 2000] was proposed to address the problems listed above. This model is based on ontological and cognitive principles. It supports instances independent of any class. Using this feature, the instance-based data model (IBDM) allows the classes in the class layer to be changed at any time without causing any instance information to be lost. This is in contrast to class-based data models, in which any class update operation may cause loss of some instances or information about the properties of instances. Also, the IBDM resolves or reduces the design problems mentioned above.

Parsons and Wand proposed the instance-based data model and suggested some methods of implementing this model. In this thesis, we continue their research and focus on developing and evaluating methods of implementing the instance-based data model. We develop some efficient accessing methods for the instance model. We also develop the basic rules of implementing the instance-based model. This research explores and paves the way for the instance-based data model database to progress from concepts to real world implementation. The research develops and analyzes possible base data structures for implementing an instance-based database. It also develops some methods for an instance-based data model database, to design and implement an instance-based database

management system based on the model. The research shows that the instance-based data model can be supported in a real world DBMS implementation, and also demonstrates the methods for the instance-based data model DBMS design and implementation. The thesis includes three major parts:

1. **Analyze the base structure**

In this part of the thesis, we focus on analyzing possible approaches to implementing an instance-based DBMS. This includes the following tasks:

- analyze what will be stored in each layer of the instance-based database;
- identify possible base data structures for implementing an instance-based database;
- compare the computational complexity of each base data structure in the instance based database under various types of update and query operations.

The research provides a method for choosing the base data structure to implement a certain instance-based DBMS in chapter 5.

2. **Analyze and implement DBMS operations**

In this part, we focus on methods for implementing a DBMS to support the IBDM. According to the theory of this model, we

- describe potential operations in the instance-based DBMS;
- develop a language (iQL) for the instance-based DBMS;

- analyze methods for implementing the commands of iQL in the instance-based DBMS;
- develop some rules for implementing some special queries of the instance-based DBMS to resolve ambiguities on these queries.

3. Implement using two base data structures and evaluate

In this part, we implement two instance-based DBMS versions using two different base data structures. We then compare the two database systems to validate the analysis results of the preceding two parts. We also compare iQL (instance-based query language) query commands with SQL query commands to demonstrate how the IBDM has some advantages.

The thesis proceeds as follows. Chapter 2 provides an overview of the concepts of the instance-based model. Chapters 3-5 analyze the possible base structures and give a general method to choose these structures when designing a database system. Chapter 6 discusses how to implement the commands of iQL in the instance-based model. Chapter 7 then implements two database systems and compares them. Chapter 8 discusses some differences between the instance-based model and the relational model. Chapter 9 discusses some conclusions and suggestions for further work.

Chapter 2

Fundamental Concepts of the Instance-based Model

The instance-based model is based on the ontology of Bunge [Bunge 1977; 1979]. It does not rely on the concepts of inherent classification, which is fundamental to class-based models, such as the relational and OO models. This chapter reviews some important concepts of the instance-based model, based on Parsons and Wand [Parsons & Wand 2000].

2.1 Fundamental Concepts of Ontology

2.1.1 Things and Properties

Postulate: The world is made of things that possess properties.

By ontology, things can be concrete or conceptual. So the word "thing" refers either to a specific object that exists in physical reality, or to anything perceived in someone's mind. Ontology also postulates that there are no things without properties, and that properties are always attached to things. It is also important to recognize that not having a property is not a property.

There are two types of properties. *Intrinsic properties* depend on one thing only. *Mutual properties* depend on two or more things. In fact, an intrinsic property describes a thing itself, while a mutual property describes a relation between things. For example, the

color of a person's hair is an intrinsic property. However, 'a person is a professor' is a mutual property, since it depends on the existence of both a person and a faculty. It cannot belong to a person if the faculty does not exist.

An ontological principle connects the existence of a thing with its properties:

Principle: No two things possess exactly the same set of properties.

Whether a property (either intrinsic or mutual) of a thing exists is not decided by humans. However, when people refer to a property of things, there are two meanings: one is that it is possessed by some things, another is people realize a property (or a set of properties) and define this property (or a set of properties) of things as the property (common property) that people call. In the ontology, this property (common property) is called an *Attribute* (by ontology, an attribute is a characteristic assigned by people to things and used to model a thing). In this thesis, consistent with the concepts of the Parsons and Wand [2000] in the instance-based model, we also call this common property a property, but in fact, after this section, any property we refer to is an attribute (unless otherwise indicated).

2.1.2 Classes

The concept of a class is widely used in our life. People refer to classes without conscious thought [Parson and Wand 2000]. For example, when we refer to 'Students', 'Birds', or 'Bread' we refer to classes. A class is a set of things possessing a finite set of common

properties (these are real properties). Things can have one or several properties in common.

Definition: A set of things T is a *class* if and only if there exists a finite set of properties P that is possessed by all members of T.

This section introduced some basic concepts of the ontological theory. Based on these principles, we will introduce the concepts of the instance-based model.

2.2 Fundamental Instance-based Model

2.2.1 Basic Principles Underling the Instance-based Model

The instance-based model depends on the ontological and classification theory views. The basic principles of this model (taken from Parsons and Wand [2000]) are as follows:

Representation Principle 1: The world is viewed as made of things that possess properties.

Representation Principle 2: Classes are abstractions created by humans in order to describe useful similarities among things.

Two conclusions can be derived from the above principles. They are also the bases of the instance-based model.

Corollary 1: Recognizing the existence of things should precede classifying them.

Corollary 2: There is no single "correct" set of classes to model a given domain of instances and properties. The particular choice of classes depends on the application.

The basic principles and the conclusions let us recognize that instances exist independent of any classes. This is the fundamental idea underlying the IBDM.

2.2.2 Two-layered Model

Based on the two principles and their corollaries, the instance-based model proposes a two-layered approach to information modeling, each layer assuming responsibility for representing different aspects of a domain. The *instance layer* represents instances and their properties. The *class layer* describes how the things are classified for certain purposes. Each layer stores information and implements operations as follow:

Instance layer: consists of the instances and their properties necessary to model a particular domain. The operations on the instance base provide the capability to create, maintain, and examine information about the domain of instances.

Class layer: consists of classes that describe similarities among instances in the instance base in terms of their shared properties. The operations on the class base provide the capability to create, maintain, and examine the classes in the class base. Some of these operations may invoke operations on the instance base.

This two-layered model or instance-based model follows the representation principles listed above. First, since the instance layer exists independent of the class layer and an instance stores all information itself, in this model an instance's existence is independent and individual. And since all instances store all information of the real world in the

database, it reflects Representation Principle 1. Second, since the class layer only stores the information about class definitions (in terms of properties), a class's existence does not affect any instance in the instance layer. In addition, a class definition must refer to the information in the instance layer (we will discuss this in chapter 7). A class can be defined only if things exist. Thus, the relevant properties must exist in the instance layer. In addition, there must be instances that possess the properties defining the class. Otherwise, we cannot define this class in this model. This reflects Representation Principle 2.

The next section discusses some differences between the instance-based model and class-based models.

2.3 Basic Differences between the IBDM and Class-based Models

Class-based data models are dependent on two corollaries: (1) We identify every thing by a *specific class* to which it belongs; (2) There exists a preferred set of classes to describe a domain. In a class-based data model, all information about instances is stored in classes. There are no instances that remain unclassified and all instances in a class possess the same properties specified in the class definition. However, in the instance-based model, although there are also instances and classes, the conception is not same. A class is only defined by some common properties of a set of instances. So there are two basic differences between the two models.

2.3.1 Instances Are Independent of Classes

One basic difference between the instance-based model and the class-base models is that the former supports instances independent of classes. In the instance-based model, the class layer only stores the information defining the class in terms of properties, no information about instances is stored in the class layer. An instance belongs to a class only if it possesses a subset of properties that match the class definition. Therefore, in the instance-based model, whether or not a class exists in the class layer has no effect in the instance layer.

2.3.2 Instances in a Class May Possess Different Properties

The second basic difference is that, in the instance-based model, instances may possess some different properties even if they belong to same class. Because all information about an instance is stored in the instance layer, an instance automatically belongs to a class if and only if the instance possesses all properties in the class definition. So different instances may have different properties whether or not they are in same class. This corresponds to the way people use classification in day-to-day life. For example, when people refer to a class 'employee', it means 'all people employed by some company'. These people will have some different properties. Some of them may have some special skills, others may not have these special properties. But when we say 'employee', they are all included. This is in contrast with class-based models, in which an instance only

possesses the properties defined in the class to which it belongs, instances do not possess "extra" properties. Consider the above example of a class 'employee' in a class-based model. If the class does not have a definition that includes the skills possessed by employees, then retrieving an instance of this class cannot provide information about skills even if this instance possesses (in the real world) certain skills.

The two basic differences produce very different results between the instance-based model and the class-based models. For example, in the instance-based model it is possible both that some instances do not belong to any class, and that other instances belong to more than one class. In contrast, in class-based models an instance must belong to a class, and instances generally do not belong to more than one class unless those classes are subclasses of a common superclass.

2.4 Summary

The basic differences between the instance-based model and class-based models are that the instance-based model recognizes first that things exist, and second that classes reflect how humans organize their knowledge about the common properties of individual things.

Chapter 3

Base Data Structures for an Instance-based Database

In this chapter we discuss what information will be stored in each layer of the instance-based database model, and propose several base data structures that can be used in this model.

3.1 Base Information in Each Layer in the Instance-based Database

According to the definition of the instance-based model, the two layers are an instance layer and a class layer. All instance information is stored in the instance layer. The class layer stores only the class information. So in the instance layer only information about instances will be stored: normally, an instance identifier, intrinsic properties, and mutual properties. In the class layer, only the class definition is stored.

3.2 Methods for Storing Data in the Instance-based Data Model

In the instance-based model, a two-layered implementation should provide a mechanism (instance engine) for identifying instances. The properties possessed by instances can also be identified by this mechanism. The mechanism is used for identifying that an instance or a property is unique.

3.2.1 Instance Layer

There are at least two possible approaches to implementing properties in the instance layer. One is to maintain each instance as a list, consisting of the instance identifier, followed by a set of the pointers to the properties that the instances possesses. An intrinsic property can also be maintained as a list, consisting of the property identifier, followed by a set of pairs consisting of the instance identifier and the value of the property for this instance. This base data structure can be simply denoted as follows:

instance identifier {property pointers}

property identifier {(instance identifier, value)}

Another method is to maintain only a part of the above base data structure, that is, maintain each intrinsic property as a list, consisting of the property identifier, followed by a set of pairs consisting of an instance identifier and the value of the property for this instance. This base data structure can be denoted as follows:

property identifier {(instance identifier, value)}

Similarly, there are two methods for storing mutual properties that parallel those described above. Specifically, if two instances possess a mutual property together, we combine their identifiers to construct a new identifier indicating the two instances are related to each other using this mutual property (if more than two instances possess a mutual property together, we also combine their identifiers to get a new identifier of the mutual property). So the two base data structures for storing mutual properties can be

denoted as follows:

First:

instance identifier {pointers to mutual properties}

mutual property identifier {(instance1 identifier, instance2 identifier, value)}

Second:

mutual property identifier {(instance1 identifier, instance2 identifier, value)}

Here, we only show a mutual property jointly possessed by two instances; if a mutual property is jointly possessed by more than two instances, we can combine the identifiers of all instances that jointly possess the mutual property to form a new identifier indicating the instances are related to each other using this mutual property. For example, if there are three instances that jointly possess a mutual property MP1, then we store this mutual property in the form: MP1{(instance1, instance2, instance3, value)}. If there are more than three instances jointly possessing a mutual property, the same approach is used.

Since there is no difference in the methods used to store intrinsic properties and mutual properties in the instance-based model, the word "property" may be used hereafter to refer to either an intrinsic or a mutual property.

3.2.2 Class Layer

There are also at least two possible approaches to store classes in the class layer of the instance-based model. First, we can maintain a class as a list, consisting of the class name

(if class name is unique in class layer) or class identifier followed by the set of property pointers that this class possesses. We denote this approach as follows:

Class name {property pointers}

Second, classes may be used to store both the properties the class possesses, and the instances. This means storing the class name or class identifier followed by two sets – a property pointer set and an instance identifier set. Properties or instances in the respective sets are possessed by the class. This method can be denoted as follows:

Class name ({property pointers}, {instance identifiers})

We have discussed the data storage methods at each layer of the instance-based model. In the next section, we discuss combining these methods to form base data structures of this model.

3.3 Base Data Structures

3.3.1 First Base Data Structure

The first base data structure model is depicted in Figure 1. In this approach, we store each instance in the instance layer as an instance identifier followed by two sets: an intrinsic property pointers set and a mutual property pointers set. In addition, we store each intrinsic property by an identifier followed by a set of pairs of (instance identifiers, value), and each (binary) mutual property by an identifier followed by a set of triples of (instance1 identifier, instance2 identifier, value). In most cases, the two instance

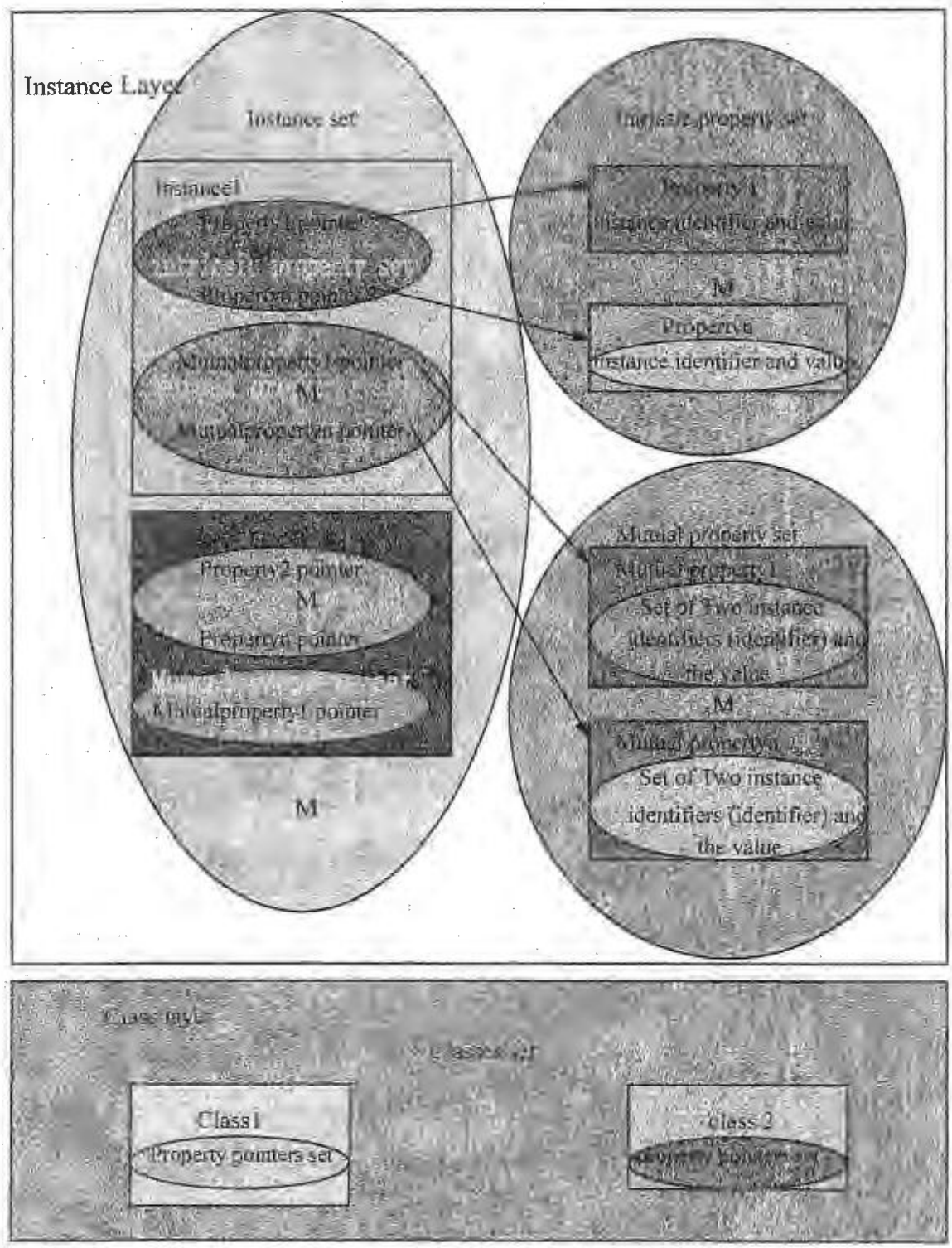


Figure 1: First base data structure

identifiers can combine to form an identifier of the two instances jointly possessing the mutual property (for example, the easy way is using one instance identifier followed by another instance identifier to construct a identifier of the mutual property). So a mutual property can be expressed as a (binary) mutual property identifier followed by a set of pairs of (identifier, value). Finally in the class layer, we store, for each class, the class name or class identifier followed by a property pointers set. We call this base data structure the *first base data structure*.

Figure 2 depicts a simple database organized according to this structure. There are three instances: instance 1, instance 2, and instance 3. Instance1 possesses property1, property2 and property3. Instance2 possesses property1 and property3. Instance3 possesses property1 and property2. Instance1 and instance2 jointly possess mutualproperty1, instance2 and instance3 also jointly possess mutualproperty1. There are two classes: class1 and class2. Each class has some intrinsic properties and mutual properties in its definition: class1 is defined by intrinsic properties property1 and property2, and mutual property mutualproperty1; class2 is defined by intrinsic properties property1 and property3, and mutual property mutualproperty1.

3.3.2 Second Base Data Structure

The second base data structure model is depicted in Figure 3. In this approach, we store an intrinsic property identifier followed by a set of pairs of (instance identifier, value),

and a (binary) mutual property identifier followed by a set of triples of (instance1 identifier, instance2 identifier, value). As in the first data structure, the two instance identifiers can combine to form an identifier of the two instances jointly possessing the

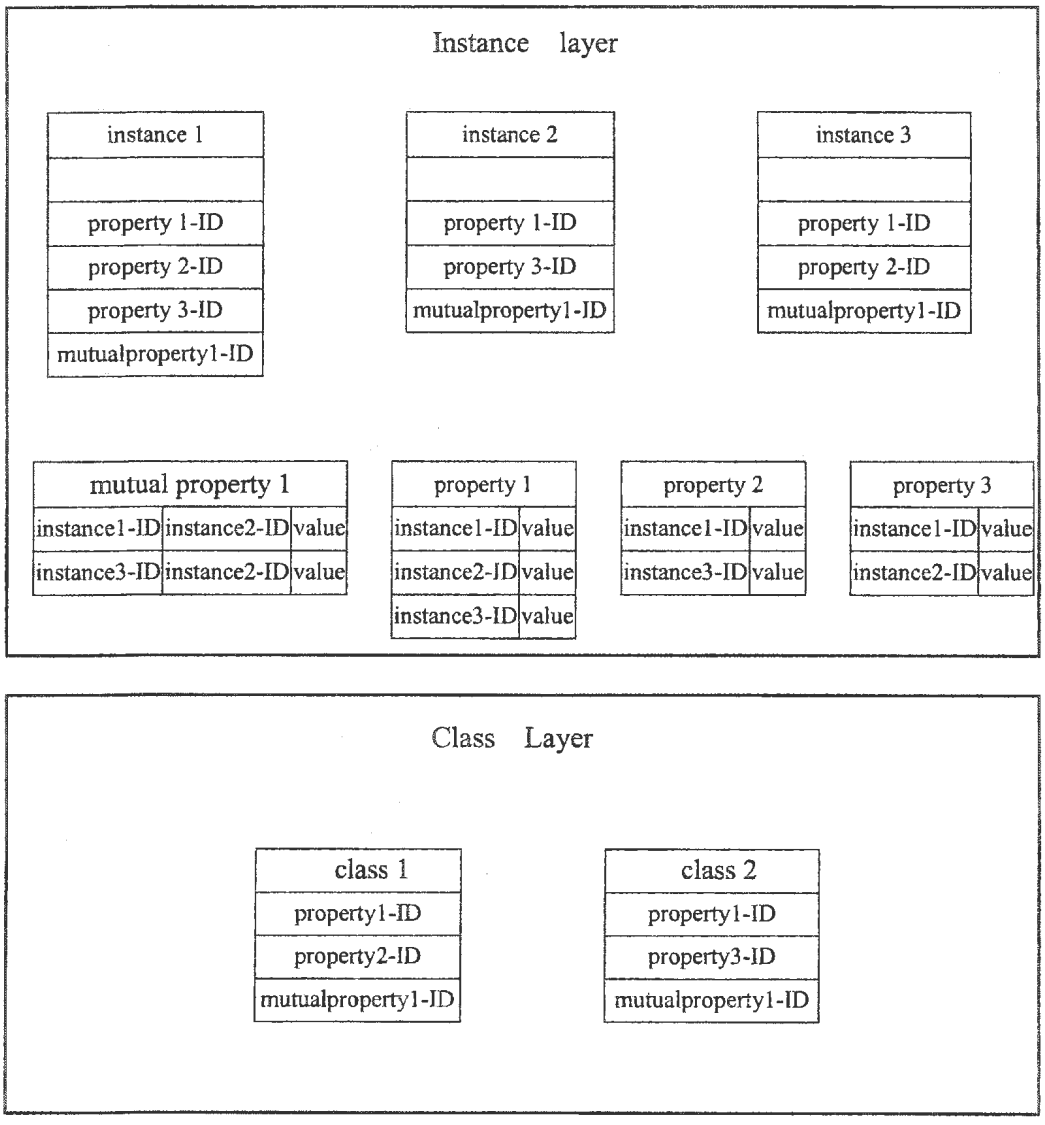


Figure 2: An example of the first base data structure

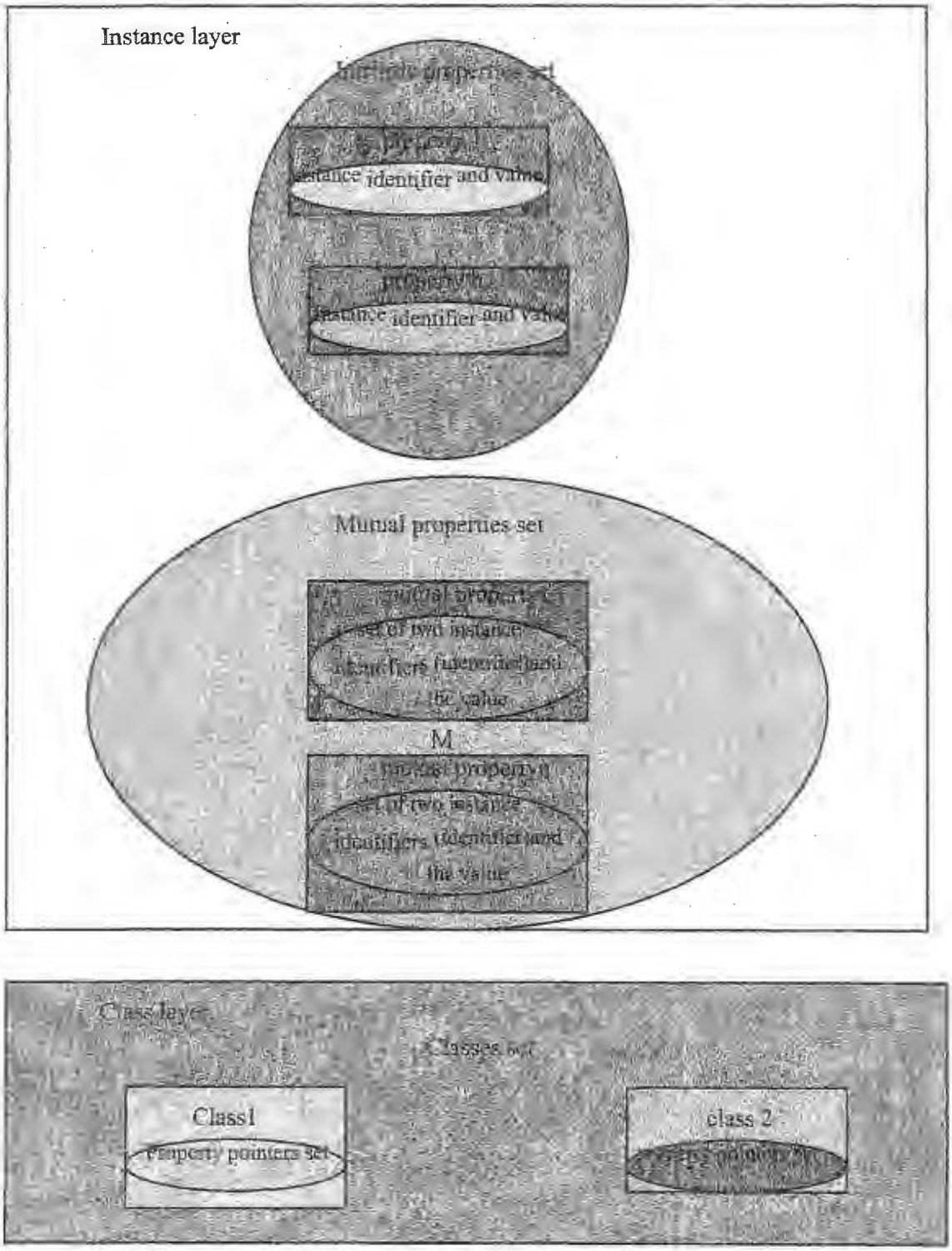


Figure 3: Second base data structure

mutual property. So a mutual property can be expressed as a (binary) mutual property identifier followed by a set of pairs of (identifier, value). As in the base data structure 1, we store the class name or class identifier followed by a property pointer set in the class layer. We call this base data structure the *second base data structure.*

Figure 4 depicts the same database contents as figure 2, but organized according to the second base data structure.

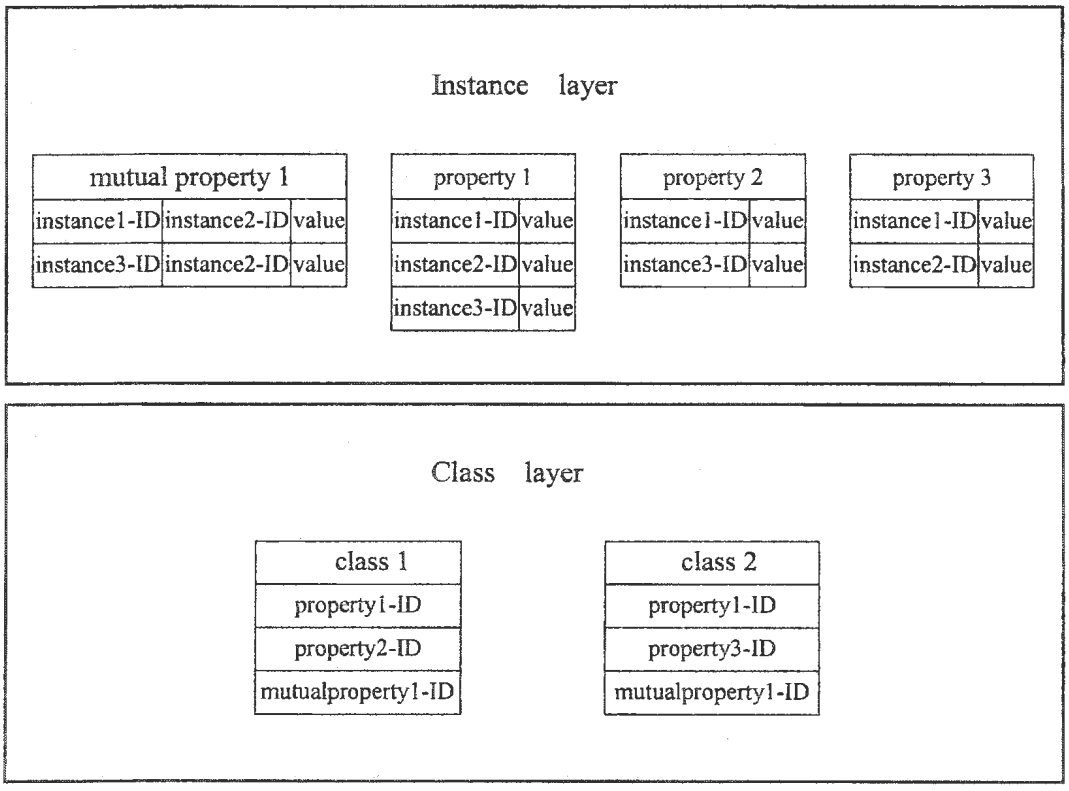


Figure 4: An example of the second base data structure

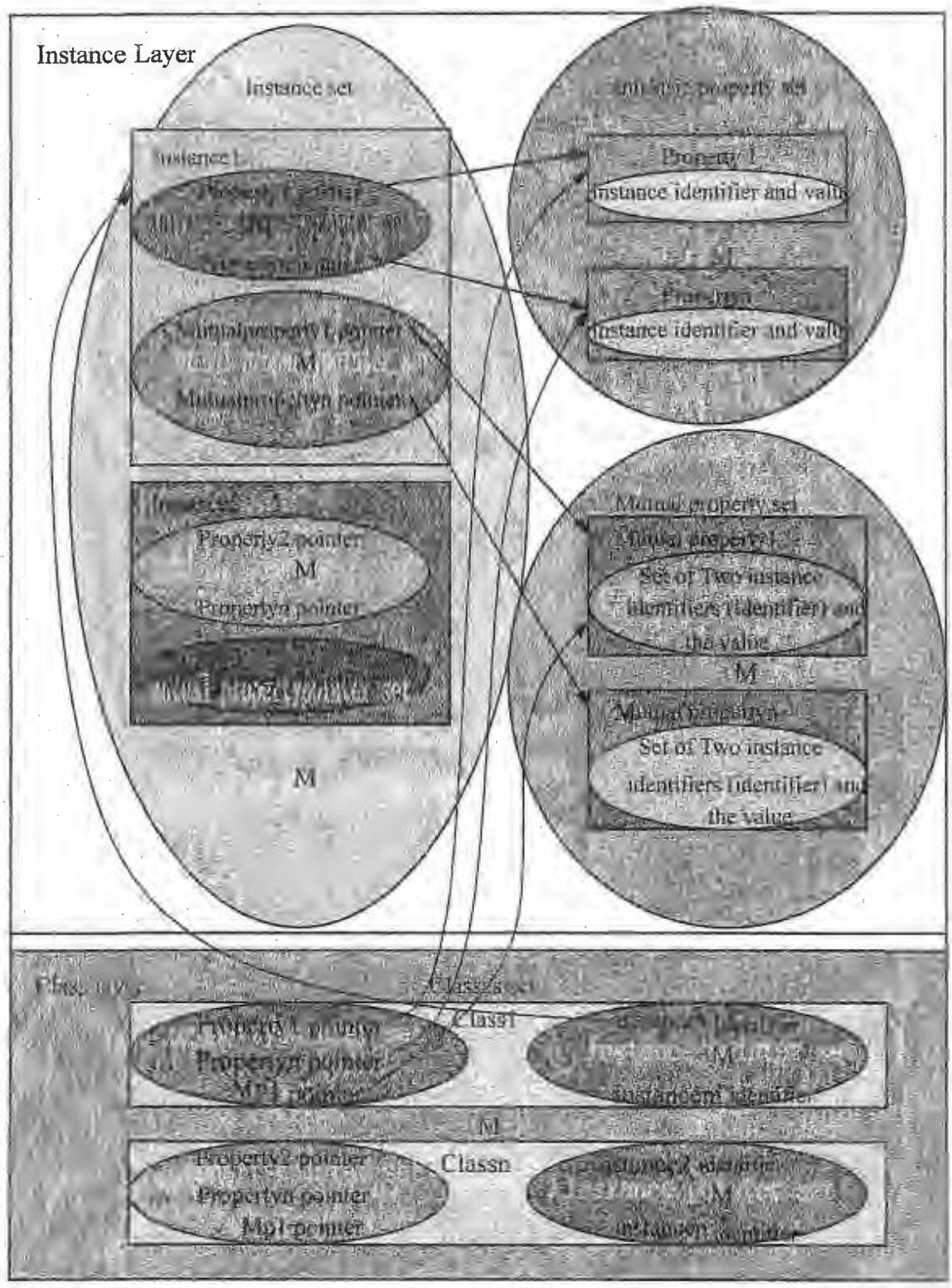


Figure 5: Third base data structure

3.3.3 Third Base Data Structure

The third base data structure model is depicted in Figure 5. In this approach, we store for each instance in the instance layer an instance identifier followed by two sets: an intrinsic property pointers set and a mutual property pointers set. In addition, we store an intrinsic property identifier followed by a set of pairs of (instance identifiers, value), and a (binary) mutual property identifier followed by a set of triples of (instance1 identifier, instance2 identifier, value). As in the above approaches, the two instance identifiers can combine to form an identifier of the two instances jointly possessing the mutual property. So a mutual property can be expressed as a (binary) mutual property identifier followed by a set of pairs of (identifier, value). For each class, we also store the class name or class identifier followed by two sets: a property pointer set and an instance identifier set. In this case, the first set is the definition of the class. The second set is the set of all instance identifiers of the instances that belong to this class. We call this base data structure the *third base data structure.*

Figure 6 depicts the same database contents as shown in Figure 2 organized according to the third base data structure.

From the above analysis, we know that since an instance-based DBMS must support two independent layers, the instance layer and the class layer, any base data structure used by such a DBMS must store the information of instances and classes separately. In each of the above three base data structures, the database is stored in two parts: an instance

layer that stores instance information and a class layer that stores class information.

Based on these requirements of the base data structure, other possible structures are depicted in Figures 7 to 9; however, they have no advantages for implementing the instance-based model. The structure shown in figure 7 cannot be directly implemented because any programming language used to implement an instance-based DBMS stores data based on the basic data types supported by the language. We only have methods for storing data with a single known data type in a structure. There is no efficient method that stores data of different data types to one structure directly. If we store data as in Figure 7, since the types of the properties may be different, different instances may need different structures to store. But before an instance is entered into a database we do not know which types of data (properties) will be in the instance structure. For example, suppose we store the instances as Figure 7, and each instance possesses several properties. Since different properties may have different types, they may need a different number of bits to store. For instance, an instance may possess the first property as a string type and the second property as an integer. However, another instance may possess properties that are all decimal fractions. Therefore, if storing the instance identifier followed by a set of pairs of property identifier and value in one file, there is no way to indicate the beginning of the second property of an instance. Even if we set the space for storing the first property as equal to the length of biggest property of the database, the problem still exists. In the future, we may add a property the length of which is longer than any property that existed

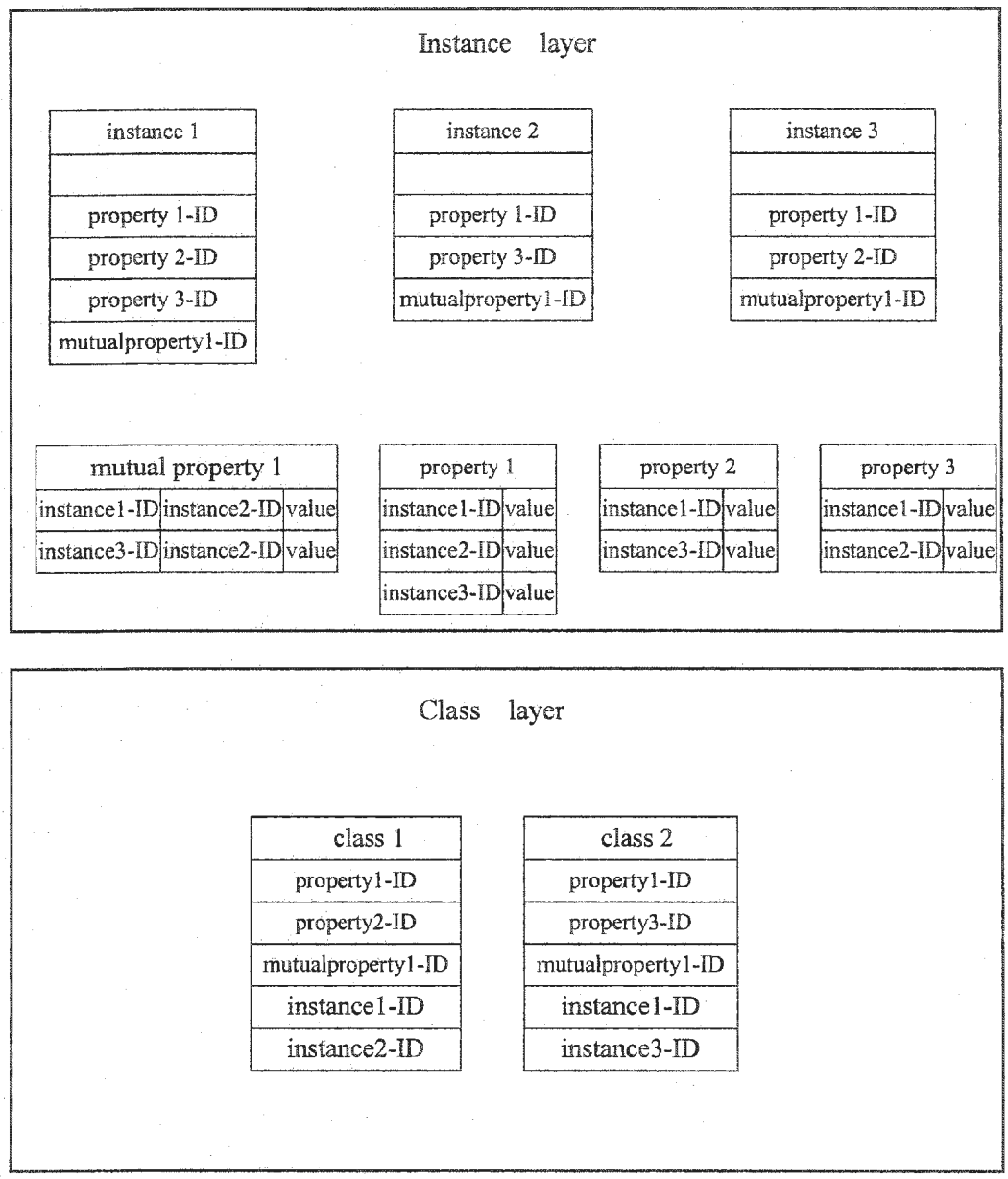


Figure 6: An example of the third base data structure

in the database after setting the space of the first property. We call this problem the *unknown structure problem*. It is difficult to access data that has an unknown structure in current programming languages. The structure shown in Figure 9 has the same problems

as in Figure 7. Therefore, we only consider the first three data structures in the remainder of this thesis.

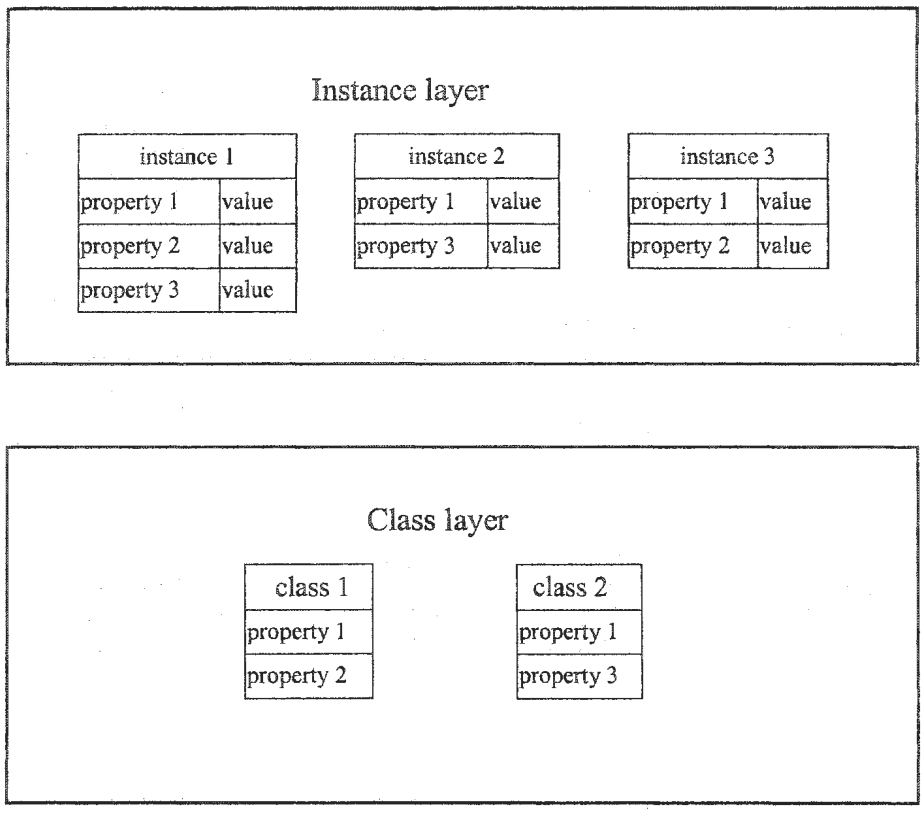


Figure 7: First example of other possible base data structures

The three base data structures are methods that can be used to store data in a DBMS that supports the instance-based model. However, choosing the best structure to support an instance-based DBMS is not trivial, since the relative efficiency of updates and queries varies between the base structures. The next chapter compares the three base data structures by analyzing the complexity of update and query operations.

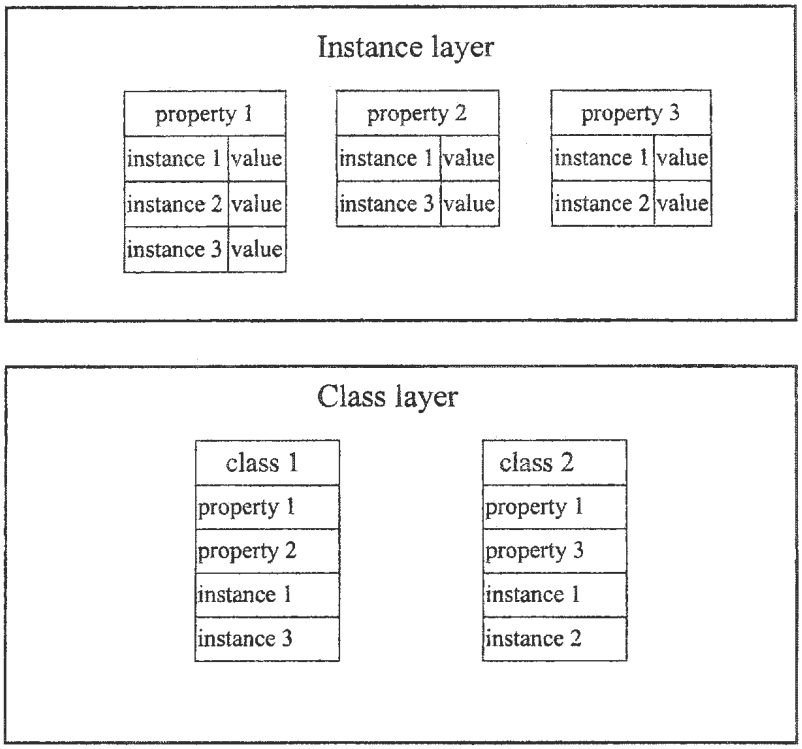


Figure 8: Second example of other possible base data structures

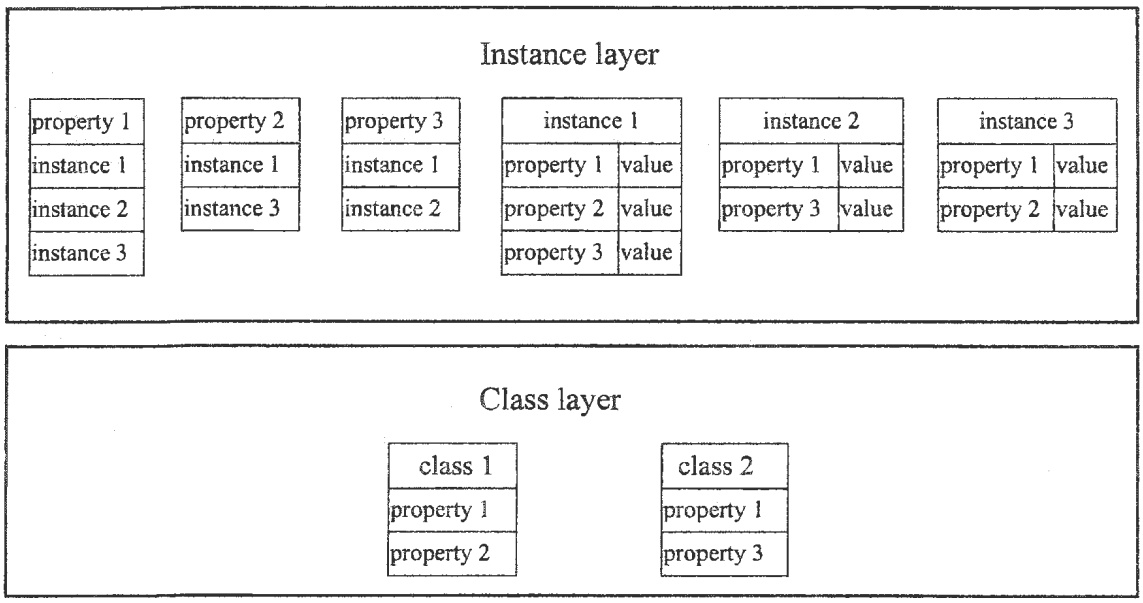


Figure 9: Third example of other possible base data structures

Chapter 4

Comparing Data Operation in Each Base Data Structures

The previous chapter identified three potential base data structures for implementing an instance-based DBMS. Each has its advantages. The first data structure can query instances faster than the second one. The second data structure enables properties to be deleted faster than first one. The third data structure is better than the first or second one for queries about instances that belong to some classes. But the third data structure is not good for updating instances or properties (since we need to maintain the integrity of a database). These are general conclusions from our data structure knowledge. However, to select the most suitable data structure when designing a database system, we must analyze the operations that will apply in this system.

In this chapter we first give some assumptions for comparing the three base data structures, then compare the cost of each type of operation in different base structures. In the instance-based model, database operations are of three types: the first involves queries in the instance layer, the second involves queries between two layers, and the third involves update operations in database. Our comparison is likewise divided into three parts.

4.1 Environment of Comparison

For ease of comparison, some environment variables in an instance-based database need to be assumed[1].

In the instance layer, we first assume that for each base data structure, the database always stores this structure using a B^+ tree [Comer 1979] [Bayer and Unterauer 1977] [Knuth 1973], which means it uses a B^+ tree to store each type of data. That is, in a system based on the first or third base data structure, its instance layer will include three directories: the instance directory that includes all instance files, and each instance file stores the instance identifier followed by the pointers to the properties possessed by this instance; the property directory that includes all property files, and each property file stores the property name followed by a set of (instance identifier, value) pairs for the instances possessing the property; the mutual property directory that includes all mutual property files, and each mutual property file stores the mutual property name followed by a set of pairs of an identifier (two instance identifiers combine to form it) and a value of the two instances jointly possessing the mutual property. In each directory, all files are stored as a B^+ tree. Also, any file itself stores the elements in it as a B^+ tree.

In an instance-based database system based on the second base data structure, the instance layer includes only two directories: the property directory and the mutual

1. The results that each operation needs time on each base data structure depend on these assumptions. However, the comparative results of this chapter do not depend on them.

property directory. We assume the structure of each directory is as described above.

In the class layer the assumptions are the same. Classes are stored in a B^+ tree, and each class also uses a B^+ tree to store its elements. That means in an instance-based system using the first or second base data structure, we store all class files in one directory, the class directory, as a B^+ tree. Each file stores only a class definition. Also, the structure of each file stores the class definition (property pointers) as a B^+ tree. However, in a system based on the third base data structure, a class file includes two parts: one is the class definition, another is a set of identifiers of the instances belonging to the class. Therefore, in this base data structure, there are two B^+ trees in one class.

Here, we assume everything is stored as a B^+ tree. Of course, in a real instance-based database, instances or classes may not be stored as a B^+ tree. They may be stored as a hashing table [Knuth 1973], a linked list, or any other data structures (e.g., grid file structure [Nievergelt 1984]). The assumptions here only provide a unified environment for the comparison.

We also assume that the memory of the system is big enough to contain one node datum, that is B (B is the node size). In addition, we assume that all the data types are the same as the data type of the instance identifier. We use the access algorithm proposed in [Kanellakis 1996].

Before beginning the comparison, we introduce some notation for our analysis.

Let (**X**, **P**) denote the instance base where **X** is a set of instances and **P** a set of properties

possessed by instances in **X**. The average number of properties possessed by an instance in **X** is denoted by I_p. The number of instances in the database is I_d. The average number of instances possessed by a class is C_I. The average number of instances possessing a property in **P** is denoted by P_I. The number of properties in the database is P_d. $\wp$ denotes the power set of a set. A class is defined by a subset of properties: $C \in \wp(\mathbf{P})$. The class base is a set of classes. The average number of properties in a class is denoted by C_p. The number of classes in the database is C_d.

This notation is summarized in table1.

Table 1 The denotation of the instance-based database

Symbol	Denotation
I_d	The number of instances in the database
I_p	The average number of properties possessed by an instance in the database
P_d	The number of properties in the database
P_I	The average number of instances possessing a property in the database
C_d	The number of classes in the database
C_I	The average number of instances possessed by a class
C_p	The average number of properties in a class definition

4.2 Comparison of Query Complexity

The queries in the instance layer are the following: properties that an instance possesses (Form of a instance); instances that possess a certain property (Scope of a property); instance pairs linked by a given mutual property (Lscope); instances that are linked with a

given instance via a given mutual property (LInst) [Parsons and Wand 2000].

4.2.1 Properties that an Instance Possesses (Form)

This operation is called Form in the instance-based model. The form of $x \in \mathbf{X}$ is a function Form: $\mathbf{X} \rightarrow \wp(\mathbf{P})$ such that Form (x)={ $P \in \mathbf{P}$ | x possesses P }. In the different base data structures, the operations for implementing the function Form(x) and the corresponding complexity are as follows.

1.1 In the first data structure (DS1, refer to Figure 1), the operations are as follows. First, find this instance in the database, this needs time $\log I_d$. Then, get the pointers of the properties this instance possesses, this needs time I_P/B. The last step is to get the values of every property the instance possesses. For each property, this step needs to find this property, then get the value according to the instance identifier. The time needed is $\log P_d+\log P_I$. So, to get values of all the properties of this instance requires time $I_P\times(\log P_d+\log P_I)$. Therefore, the total time required for the Form operation in DS1 is: $\log I_d+ I_P/B+ I_P\times(\log P_d+\log P_I)$.

1.2 In the second data structure (DS2, refer to Figure 3), the operation must retrieve all properties to determine whether or not each property is possessed by the instance, so this operation needs time $P_d\times\log P_I$.

1.3 In the third data structure (DS3, refer to Figure 5), the data structure is the same as DS1 in the instance layer, so the query time is the same as DS1; that is, $\log I_d+ I_P/B+$

$I_P \times (\log P_d + \log P_I)$.

4.2.2 Instances that Possess a Property (Scope)

This operation is called Scope. The scope of $P \in \mathbf{P}$ is a function Scope(P): $\mathbf{P} \rightarrow \wp(\mathbf{X})$ such that Scope (P)={ $x \in \mathbf{X}$ | x possesses P }. In all three data structures, to perform this query, it is necessary to first find this property in the database (the time is $\log P_d$), then get all the instances that possess this property (the time is P_I/B). So this operation needs time $\log P_d + P_I/B$.

4.2.3 Instances Linked by a Given Mutual Property

This operation is called Linked Scope. The linked scope of a mutual property $P \in \mathbf{P}$ is a function LScope(P): $\mathbf{P} \rightarrow \wp(\mathbf{X})$ such that LScope(P)={$S \in \wp(\mathbf{X})$ | P is a mutual property of all $x \in S$}. Since mutual properties are stored by the same structure in the three base data structures, the time needed for this operation in each of the three data structures is $\log P_d + P_I/B$.

4.2.4 Instances that Share a Mutual Property with a Given Instance

This operation is called Linked instances. The linked instances of an instance x is a function Linst: $\mathbf{X} \otimes \mathbf{P} \rightarrow \wp(\mathbf{X})$ such that Linst(x, P)={$y \in \mathbf{X}$ | P is a mutual property of x and y}. As with the Linked Scope operation, in the three data structures, the time needed for this operation is the same. Since a mutual property uses a combined identifier of two

instance identifiers, there is no method to decide which identifier we need unless we retrieve all identifiers. So the operation is in two steps: find the mutual property (the time is $\log P_d$) then get all identifiers (the time is P_I/B). Therefore, this operation needs time $\log P_d + P_I/B$.

From the above, it is clear that for queries in the instance layer, the first and third data structures are superior to the second one, since they are faster for querying an instance form and are equivalent on scope and mutual property queries (when querying the instance layer, if the first or third base data structures are not faster than the second base data structure, they can use the same method as the second base data structure to query).

We next compare the data structures with respect to queries between the instance layer and the class layer. There are two queries between instance layer and class layer: instances that belong to a class, and classes to which an instance belongs.

4.2.5 Instances that Belong to a Class

This operation is called Membership. The membership of a class is a function Mem: $\mathbf{C} \rightarrow \wp(\mathbf{X})$ such that $\text{Mem}(C) = \{x \in X \mid x \in \text{Scope}(P) \forall P \in C\}$. The time required to perform this operation is different in the three base data structures.

1.1 In DS1, the class does not store any information about instances. However, the property information is stored in both the instance layer and the class layer. Therefore, the following operations are required for the query. First, find the properties possessed

by the class, i.e., query the class definition. This involves finding the class first (the time is $\log C_d$), then retrieving all property pointers in the class definition (the time is C_P/B). Thus, this step requires time $\log C_d+C_P/B$. Second, for an instance it is necessary to check whether it possesses each property in the class definition. If there are some instances that possess all of the properties, then these instances belong to the class. In the second step, it is not necessary to check all instances in the database. We only check the instances that possess at least one property in the class definition. This step uses the following method:

1. Get the set of instances that possess the first property in the class definition. That is a scope of the property. It needs time $\log P_d+P_I/B$;
2. Check whether an instance of the set possesses the second property in the class definition. If any instance does not possess the property, delete it from the instance set. For one instance, this check involves finding the property first (the time is $\log P_d$), then checking whether the instance possesses the property (the time is $\log P_I$). Therefore, to check all the instances (P_I) in the instance set needs time $(\log P_d+\log P_I)\times P_I$;
3. Check other properties in the class definition. Since only the (C_P-1) remaining properties that in the class definition need to be checked, therefore, step 2 and step 3 need time $(C_P-1)\times(\log P_d+\log P_I)\times P_I$.

So, the second step need time is the sum of steps 1-3. That is

$(\log P_d+P_I/B)+(C_P-1)\times(\log P_d+\log P_I)\times P_I$. The total time needed for this operation is the sum of the above steps:

$$\text{Log}C_d+C_P/B+(\log P_d+P_I/B)+(C_P-1)\times(\log P_d+\log P_I)\times P_I.$$

1.2 In the second data structure, the operation is the same as the first one.

1.3 In the third data structure, the class stores a set of instance identifiers that all instances belong to the class. So the query operation is to find the class in the class layer (the time is $\log C_d$), then get the all instances that belong to this class (the time is C_I/B). So the time required for this operation is $\log C_d+C_I/B$.

4.2.6 Classes to which an Instance Belongs

This operation is called Sort. A sort of an instance $x \in \mathbf{X}$ is a function Sort: $\mathbf{X} \rightarrow \wp(\mathbf{C})$ such that Sort(x) = $\{C \in \mathbf{C} \mid x \text{ is a membership of } C\}$. The time needed in the three base data structures is as follows.

1.1 In DS1, it is necessary to compare all properties possessed by the instance with all properties possessed by each class in the class layer. If there are some properties possessed by this instance that constitute all the properties that define a class, then this instance belongs to the class. It is necessary to check all classes in the class layer to find the classes to which this instance belongs. The operation requires the following steps. First, find the instance and get all property identifiers possessed by the instance (the time is $\log I_d+I_P/B$). Second, get all class definitions, this requires

time $C_d \times C_P/B$. The last step is to compare the property identifiers possessed by the instance with each class definition. Since the property pointers in an instance and in a class are all stored by a B^+ tree, that we want to compare the two B^+ trees. To decide whether or not one is a subset of another between these B^+ trees need time is I_P+C_P [Cormen 1989]. Therefore, this step needs time $C_d \times (I_P+C_P)$. So the operation need time is $\log I_d+I_P/B+C_d \times C_P/B+C_d \times (I_P+C_P)$.

1.2 In DS2, this query is more complex than in DS1 because it is necessary to find the properties possessed by an instance in the instance layer first. This step involves querying the properties that an instance possesses. It is necessary to check all properties to find which properties the instance possesses, so the time needed is $P_d \times \log P_I$. Then it is necessary to determine which classes this instance belongs to. The second step is the same as the first data structure operation in the comparison step, it needs time $C_d \times C_P/B+C_d \times (I_P+C_P)$. Thus, the total time needed for this operation is $P_d \times \log P_I+ C_d \times C_P/B+C_d \times (I_P+C_P)$.

1.3 In DS3, the class stores all instances that belong to the class. So the only operation is to check all classes to find which class that to which this instance belongs. This operation need time is $C_d \times \log C_I$.

It is clear that the third base data structure is superior for querying the instances that involved some classes.

4.2.7 Query in the Class Layer

There are two queries in the class layer: which classes possess a property and which properties define a class. Because the information between class and properties in the class layer is the same in the three data structures of the instance-based model, the time needed in each data structure also is the same. In this case, query time of these two queries has no effect on the results of our comparison.

The first query, query a class definition, only needs to find the class, then retrieve the properties possessed by this class. This operation need time is $\log C_d+C_P/B$. The second query, classes that possess a property, needs to check all classes to find which classes possess this property. This operation need time is $C_d \times \log C_P$.

This completes our comparison of the complexity of the basic query operations in the instance-based database system. Tables 2- 4 summarize the results of the comparison.

Table 2:Query in the instance layer

Query	Query properties that an instance possesses	Query instances that possess a property
first data structure	$\log I_d+ I_P/B+ I_P\times(\log P_d+\log P_I)$	$\log P_d+P_I/B$
Second data structure	$P_d \times \log P_I$	$\log P_d+P_I/B$
third data structure	$\log I_d+ I_P/B+ I_P\times(\log P_d+\log P_I)$	$\log P_d+P_I/B$

Table 3:Query between two layers

Query	Query instances that belong to a class	Query classes to which an instance belongs
first data structure	$\log C_d+C_P/B+(\log P_d+P_I/B)+(C_P-1)\times(\log P_d+\log P_I)\times P_I$	$\log I_d+I_P/B+C_d\times C_P/B+C_d\times(I_P+C_P)$
second data structure	$\log C_d+C_P/B+(\log P_d+P_I/B)+(C_P-1)\times(\log P_d+\log P_I)\times P_I$	$P_d\times\log P_I+ C_d\times C_P/B+C_d\times(I_P+C_P)$
third data structure	$\log C_d+C_I/B$	$C_d \times \log C_I$

Table 4:Query in the class layer

Query	Query classes possess a property	Query a class definition
first data structure	$C_d \times \log C_p$	$\log C_d + C_p/B$
second data structure	$C_d \times \log C_p$	$\log C_d + C_p/B$
third data structure	$C_d \times \log C_p$	$\log C_d + C_p/B$

From the above comparison, we find that the third base data structure is best for querying operations. It is fast in queries that span the instance and class layers (Table 3), and it is equivalent on other query operations with the first base data structure.

4.3 Comparison of Update Complexity

The second major type of comparison in the instance-based database involves update operations in each data structure. There are three types of update operations: update an instance or some property of an instance; update a property; and update a class. Next, we compare the complexity of these operations under each of the three data structures.

4.3.1 Insert or Delete an Instance

Any insert or delete instance operation updates both the instance and also the properties that the instance possesses. That means that when inserting a instance to a database, two steps are needed: first, add the instance, that is a function Add_Inst ::= $\mathbf{X} \rightarrow \mathbf{X} \cup \{x\}$; second, add properties to the instance, this is the function Add_Prop_Inst(x, P) ::= $p(x) \rightarrow p(x) \cup \{P\}$, and add properties to the instance layer if the properties are not in the database before, this is the function Add_Prop(P) ::= $\mathbf{P} \rightarrow \mathbf{P} \cup \{P\}$. The converse

operation is the delete an instance operation, again with two steps: first, delete properties of the instance, that is a function Del_Prop_inst (x, P)::= p(x) → p(x)-{P}, and remove properties from the instance layer if the properties are possessed by no other instances, this is function Del_Prop(P) ::= $\mathbf{P} \rightarrow \mathbf{P}$-{P}; second, remove the instance, that is a function Rem_Inst(x)::=$\mathbf{X} \rightarrow \mathbf{X}$-{x}. Since the data structure is different in the three base data structures, the time required for these operations is different.

1.1 In DS1, if inserting an instance, we need only to insert the instance to the instance layer. This step includes two parts: first, insert instance identifier and the property identifiers of these properties the instance possesses. This part need time is $\log I_d + I_P/B$; second, insert each value of each property possessed by the instance to the properties. This part need time is $I_P \times (\log P_d + \log P_I)$. So this operation need time is

$\log I_d + I_P/B + I_P \times (\log P_d + \log P_I)$.

If we want to delete an instance we need to first locate this instance in the instance layer, which on average requires time $\log I_d$. Then, we need to get all properties that the instance possesses, this step need time is I_P/B. Based on the properties the instance possessed, delete each property value of the instance. That is, find each property (the time is $I_P \times \log P_d$), and delete each value of the instance (the time is $I_P \times \log P_I$). Thus, the total time is $I_P \times \log P_d + I_P \times \log P_I$. If this instance is the last one that possesses a property, then delete the property also. Finally, delete the instance file. Therefore, this operation needs time is

$\log I_d + I_P/B + I_P \times(\log P_I+\log P_d)$.

1.2 In DS2, if inserting an instance, we only need to insert the values of each property the instance possesses to each of the properties, so the time needed is $I_P\times(\log P_d+\log P_I)$. To delete an instance, we need to check the whole instance layer to find the properties possessed by this instance, then delete them. This operation needs time $P_d\times\log P_I$.

1.3 In DS3, if inserting an instance, we also need two steps: first, insert the instance to the instance layer which, like inserting an instance in DS1, needs time $\log I_d + I_P/B + I_P\times(\log P_d+\log P_I)$. Second, we check whether this instance belongs to each class in the class layer. If the instance belongs to any class, add its identifier to the class. For each class, this step needs to compare this instance to the class (the time is I_P+C_P). If it belongs to the class, insert its identifier to this class (the time is $\log C_I$). The total time needed for one class in this step is $I_P+C_P+\log C_I$. So, for all classes in the database, the total time needed for the second step is $C_d\times(I_P+C_P+\log C_I)$. Therefore, the insert operation needs time $\log I_d + I_P/B + I_P\times(\log P_d+\log P_I)+ C_d\times(I_P+C_P+ \log C_I)$.

The delete operation is more complex than in the other base date structures. We need to delete the instance in both the instance layer and the class layer. So there are two steps for this operation. The first is to delete this instance in instance layer, which is the same as deleting an instance in DS1. It needs time $\log I_d+I_P/B+I_P\times(\log P_I+\log P_d)$. The second step is to delete the instance identifier from the classes in the class layer to which the instance belongs. This step checks all classes to determine whether the

instance identifier is in its instance identifier set, and if so, deletes it from the set. This step needs time $C_d \times \log C_I$. So this operation needs time

$$\log I_d + I_P/B + I_P \times (\log P_d + \log P_I) + C_d \times \log C_I.$$

4.3.2 Update an Instance Property Value

In this case, the operation is only in the instance layer and the three data structures need the same operation: find the property first, then find the instance value. The time needed is $\log P_d + \log P_I$.

The two above update operations are the same as in the relation database or the object oriented database operation in that they are all value operations. That means any update operation only changes the value of the database, either instance or property value. These operations do not change the schema of the database. However, the following update operations are not supported in either a relation database or an object oriented database. These operations let the schema change without loss of information. These update operations are compared next.

4.3.3 Insert or Delete a Property of an Instance

This operation is in the instance layer. The 'add a property to an instance' operation is a function Add_Prop_Inst(x, P) ::= $p(x) \rightarrow p(x) \cup \{P\}$. Of course, if the property does not already exist in the instance layer, there is an additional operation at first: add the property

to the instance layer. That is a function Add_Prop(P) ::= $\mathbb{P} \rightarrow \mathbb{P} \cup \{P\}$. The opposite operation is 'delete a property of an instance', that is a function Del_Prop_Inst(x, P) ::= $p(x) \rightarrow p(x) - \{P\}$. An additional operation, remove the property from the instance layer is also needed if the property subsequently belongs to no instance, that is a function Del_Prop(P) ::= $\mathbf{P} \rightarrow \mathbf{P} - \{P\}$. The time needed for this operation is different in the three data structures.

1.1 In DS1, if inserting a property of an instance, there are two steps. First, insert the property to the set of the property pointers that this instance possesses. This step needs to find the instance first, then add the property. The time needed is $\log I_d + \log I_P$. Second, insert the property value of this instance for this property. If no instance possesses this property before, then add this property to the database first. This step needs to find the property first, then add the value. The time needed is $\log P_d + \log P_I$. So the insert operation needs time

$\log I_d + \log I_P + \log P_d + \log P_I$.

There are also two steps in the operation for deleting. First, find the instance in the instance layer, and delete the property pointer from the instance file; second, find the instance identifier in the property file, and delete this (instance identifier, value) pair. Of course, if this instance is the only instance that possesses the property, then we need to delete this property at the same time we delete the value. The first step needs time $\log I_d + \log I_P$. The second step needs time $\log P_d + \log P_I$. So the delete operation

needs time

$$\log I_d + \log I_P + \log P_d + \log P_I.$$

1.2 In DS2, this operation is the same as 'update an instance property value'. In this data structure, we do not care whether update is on the value or the schema itself, since there is no difference in operations on the data structure. The time needed is also the same as update an instance property value, which is $\log P_d + \log P_I$.

1.3 In DS3, there is a difference between updating a property value and updating the property itself. In the third data structure, the class stores some instance identifiers in the class layer. Therefore, if a property of an instance is deleted, this instance may no longer belong to some classes. So if deleting an instance property, we must check all classes to determine whether or not the instance still belongs to each class. If the instance no longer belongs to a class, delete the instance identifier from the set of the instance identifiers in the class. So for the third data structure, the delete operation requires two steps. The first step is to find this property in the instance layer and delete the property pointer in the instance file, then delete the pair (instance identifier, value) in the property file. This step is the same as in DS1. The time needed is $\log I_d + \log I_P + \log P_d + \log P_I$. The second step is to check the instance in the class layer. For each class, if this instance no longer belongs to the class then delete this instance identifier from the class. This operation needs time:

$$\log I_d + \log I_P + \log P_d + \log P_I + C_d \times (\log C_P + \log C_I).$$

In the above, the operation is to implement deleting a property of an instance, so we only check whether the property that is deleted from the instance belongs to some classes. For each class definition that includes this property, we must check whether the instance belonged to the class before. If so, then we delete the instance identifier from the class. We do not check whether this instance belongs to other classes that do not possess the property that is deleted by this instance. An instance belongs to a class if a subset of the properties it possesses equals the class definition. So, if we delete a property from an instance, it can only change the fact that this instance belongs to a class which includes this property in its definition.

If we insert a property to an instance, we also need two steps as in deleting a property of an instance. However, in second step, if a class definition includes the property that we insert to the instance, then we need to check whether a subset of the instance's properties match the class definition. If so, we add the instance to the class. So, the first step of the insert operation is the same as the delete operation, but in the second step, we need to compare the properties possessed by the instance to the class definition if the class includes the property in it definition. The second step needs time $C_d \times (\log C_P + (C_P + I_P))$. So the total time of this operation is:

$$\log I_d + \log I_P + \log P_d + \log P_I + C_d \times (\log C_P + (C_P + I_P)).$$

As in the delete operation, in the second step we only compare the classes whose definition includes the property inserted into the instance with the instance.

4.3.4 Delete a Property from a Database

The instance-based data model allows a property to be deleted from a database without losing information. Since we store property definitions in the instance layer in all the three base data structures, we need to delete a property definition in the instance layer. However, in the class layer, some class definition may include this property pointer. Because the property definition has been deleted, the pointers lose their meanings, so these pointers also need to be deleted. Thus, we apply this operation in both instance layer and class layer. To implement this operation, each data structure has different costs.

1.1 In DS1, a property definition is stored in the instance layer, but its pointer is stored in both the instance layer and the class layer. To implement this operation, first we need to check the property pointer in each instance of the instance layer. If any instance includes the property pointer then delete the pointer (the time is $I_d \times \log I_P$). We also need to delete all values of every instance possessing the property (the time is $\log P_d$). Second, check each class of the class layer, if any class includes the property pointer in its definition (the class includes the property in its definition) then delete this pointer or delete this class* (the time is $C_d \times \log C_P$). Thus, this operation needs time:

$$I_d \times \log I_P + \log P_d + C_d \times \log C_P$$

In DS2, the property definition is also stored in the instance layer, but only the class

* The decision on whether the class should be redefined to now exclude the deleted property or be deleted, should be made by the database administration.

delete it (the time is logPd). The second is to find the property pointers in some class layer and store the property pointer in some class definitions. Therefore, implementing this operation also needs two steps. The first is to find this property in the instance layer and definitions in the class layer and delete it (the time is $C_d \times \log C_P$). This operation needs time: $\log P_d + C_d \times \log C_P$.

1.3 In DS3, this update operation is the same as DS1, because deleting a property from both the instance layer and the class layer will not affect the classification of instances. So the time needed is the same as the DS1: $I_d \times \log I_P + \log P_d + C_d \times \log C_P$.

4.3.5 Update a Property in a Class

Inserting or deleting a property in the class layer does not affect the instance layer in any of the three data structures. But there are some different operations between the first two data structures and the third one. Since in the first two data structures, the instance layer and the class layer are independent of each other, if we update in one layer, it does not affect the other. So in the two first base data structures, there is no operation needed in the instance layer. The operation is only to find the class and insert or delete the property identifier in the class. The time needed is $\log C_d + \log C_P$.

However, in the third data structure there are some relations between the class layer and the instance layer, because in this case the class stores the pointers to the instances that belong to the class. So if we delete or insert a property in the class definition (the

time is $\log C_d+\log C_P$), we also need to check whether an instance belongs to the class (the time is $I_d\times(I_P+C_P)$). The time required for this operation is greater: $\log C_d+\log C_P+I_d\times(I_P+C_P)$.

In general, any operation to update a property in the database can be transformed into two types of operations: update a property of an instance and update a property of a class. So after this chapter, we will not analyze the operation of update a property in the database but analyze update a property in either an instance or a class.

4.4 Summary

At this point, all operations of the instance-based database have been compared. Tables 5 and 6 summarize the results of the update operations in each base data structures.

From the comparisons above, we know that the second base data structure is faster for updating some data than either the first base data structure or the third base data structure. This is because it is faster than the first and the third base data structures when updating some properties of an instance. The first base data structure, also, is faster than the third base data structure when updating some property of an instance and updating some properties of a class. However, the comparison of this chapter is based on each operation separately. We do not know which base data structure is best for a certain system having particular query and update frequency. In the next chapter, we discuss how to select a base data structure based on these considerations.

Table 5:Update instance or property values of an instance

Update	delete or insert a instance	update a property values in a instance
first data structure	$\log I_d + I_P/B + I_P \times(\log P_I+\log P_d)$	$\log P_d + \log P_I$
second data structure	$P_d \times \log P_I$	$\log P_d + \log P_I$
third data structure	$\log I_d+I_P/B+I_P\times(\log P_d+\log P_I)+C_d\times\log C_I$	$\log P_d + \log P_I$

Table 6:Insert or delete a property in a database

Update	Insert or delete a property of a instance	Insert or delete a property of a class
first data structure	$\log I_d + \log I_P + \log P_d + \log P_I$	$\log C_d+\log C_P$
second data structure	$\log P_d+\log P_I$	$\log C_d+\log C_P$
third data structure	$\log I_d + \log I_P + \log P_d + \log P_I + C_d\times(\log C_P+\log C_I)$	$\log C_d+\log C_P+I_d\times(I_P+C_P)$

Chapter 5

Choosing Base Data Structures

Different databases serve different purposes. Sometimes we store data to a database only to support queries. The data in such databases are changed infrequently, if at all. This type of system is usually used for some public service areas. In these databases, queries are much more frequent than updates. However, in other cases, the data in a database are changed frequently. Common examples include transaction processing systems (i.e., a sales system, or a reservation system). In these databases, the most common operations are updates. So in the real world, the prevalence of query versus update operations is highly variable and application specific.

Since the frequency of operations on a database can vary, and since chapter 4 showed that base data structures incur different costs for query versus update operations, to choose a suitable base data structure for an instance-based DMBS, we must analyze the three base data structures accordingly.

5.1 Mathematical Models for Choosing Base Data Structures

Before specifying the mathematic models, we define several variables used in the analysis. These variables are listed in table 7:

Table7: The denotation of the operations of database

Variables	Denote the meaning
Q_I	Proportion of database operations comprised of 'instance form' queries;
Q_P	Proportion of database operations comprised of 'instances possessing a property' operations
$Q_{C\text{-}I}$	Proportion of database operations comprised of 'instances of a class' operations
$Q_{I\text{-}C}$	Proportion of database operation comprised of 'classes of an instance belongs to' operations
U_I	Proportion of database operation comprised of 'update an instance' operations
$U_{I\text{-}P}$	Proportion of database operation comprised of 'update a property of an instance' operations
$U_{C\text{-}P}$	Proportion of database operation comprised of 'update a property of a class' operations

Since we want to compare the different costs of the three base data structures, we will not consider further the operations that have the same costs in all three base structures. According to the comparison in chapter 4, the class query operations in Table 4 incur the same cost for all three base data structures. Therefore, we only consider Tables 2, 3, 5 and 6 for calculating the different cost of each base structure. We also do not include the operation 'query instances which possess a property', as all three base data structures have the same cost for this operation. In each operation, if there are some steps of an operation that have the same cost in all three base data structures, we delete these costs in each structure. For example, the operation 'query instances that belong to a class' has a step 'finding the class', and in all three base data structures there is a term, $\log C_d$, that

represents the cost of that step, so we eliminate this for the purpose of comparison. After canceling items of the three base data structures with the same costs, we count the sum of the product of the remaining operations multiplied by the proportion of each operation occurrence in all the operations of the database. The results represent the costs of each base data structure. We list them as follows.

First base data structure:

$$(\log I_d+I_P/B+I_P\times(\log P_d+\log P_I))\times Q_I+(C_P/B+(\log P_d+P_I/B)+(C_P-1)\times(\log P_d+\log P_I)\times P_I)\times Q_{C\text{-}I}+$$
$$(\log I_d+I_P/B+C_d\times C_P/B+C_d\times(I_P+C_P))\times Q_{I\text{-}C}+(\log I_d+I_P/B+I_P\times(\log P_I+\log P_d))\times U_I+(\log I_d+$$
$$\log I_P)\times U_{I\text{-}P}+(\log C_d+\log C_P)\times U_{C\text{-}P} \qquad \text{(i)}$$

Second base data structure:

$$P_d\times\log P_I\times Q_I+(C_P/B+(\log P_d+P_I/B)+(C_P-1)\times(\log P_d+\log P_I)\times P_I)\times Q_{C\text{-}I}+(P_d\times\log P_I+\times C_P/B+$$
$$C_d\times(I_P+C_P))\times Q_{I\text{-}C}+ P_d\times\log P_I\times U_I+(\log P_d+\log P_I)\times U_{I\text{-}P}+(\log C_d+\log C_P)\times U_{C\text{-}P} \qquad \text{(ii)}$$

Third base data structure:

$$(\log I_d+ I_P/B+ I_P\times(\log P_d+\log P_I))\times Q_I + C_I/B\times Q_{C\text{-}I}+ C_d\times\log C_I\times Q_{I\text{-}C}+ (\log I_d+I_P/B+$$
$$I_P\times(\log P_d+\log P_I) +C_d\times\log C_I)\times U_I+ (\log I_d + \log I_P+C_d\times(\log C_P+\log C_I))\times U_{I\text{-}P}+$$
$$(\log C_d+\log C_P+I_d\times(I_P+C_P))\times U_{C\text{-}P} \qquad \text{(iii)}$$

If we compute these three expressions, the result with the lowest total cost indicates the "best" data structure to choose for an instance-based DBMS. In practice, however, the relative proportions of queries and updates of various types are not known. The next section discusses a general and useful comparison approach.

5.2 General Methods for Choosing Base Data Structures

From the three expressions in the previous section, we can in principle accurately choose the best one of the three base data structures when designing a database system. However, the values of many variables need to be known in order to compute the results. In practice, many of these variables will not be known in advance. The expressions above, therefore, are not immediately useful for determining how best to implement an instance-based database system. Instead, we focus on developing some general methods for choosing the base data structures.

For our analysis, database systems can be classified into three types. One is when a database is very small. The second is when the update operations are very frequent. The third is when query operations are very frequent.

5.2.1 Database is Very Small

In this case, 'small' means that all data of the database can be loaded into the system memory in one access operation. So, if there are some data structures such that the data can be loaded into the memory in one access operation, then we should choose these data structures. In our data structures, DS2 requires the least memory, and DS1 needs less memory than DS3.

In some cases, there can be more than one base data structure that makes the database size small. The criterion for choosing a data structure in this case is the query or update

frequency.

In most cases, it will be impossible to load all data of the database at one access operation. In this case, we do not need to think about storing data in the least space, but need to consider all the conditions in the mathematical model.

When deciding which base data structures are most suitable for a database, we first consider what types of update operations are frequently performed in a database. If we observe the operations in a relational database or an object oriented database, we find that users mostly update some attributes values of an instance or an instance itself. In the instance-based database, we allow update operations on classes or instance properties (schema changes). However, these operations may not be very frequent in a real database. Therefore we do not consider update to the schema in the following analysis.

5.2.2 Query Operation Dominate

We define this type of database to mean that we query data or update the property values of instances possess much more than delete instances, properties or classes. From the comparison of chapter 4, we know that for query operations, the third base data structure will be faster than the second or first base data structure, since it is faster in querying instances belonging to classes. Also the first data structure is better than the second data structure for querying instances. Therefore, we compare the third base data structure with first base data structure to get the conditions that direct us to choose the third base data

structure. We do not compare the third base data structure with the second base data structure in this case, because in the instance layer the second base data structure only is a part of the first data structure, and in the class layer they are same. So for any query or update involving the values of properties, the cost of the second data structure is no less than the first data structure.

First, we delete all clause of update attribute property or class property from the mathematical models above. Then we find that DS1 and DS3 are different in two parts. One is querying instances of some classes, and another is updating an instance. So if

$(C_P/B+(\log P_d+P_I/B)+(C_P-1)\times(\log P_d+\log P_I)\times P_I)\times Q_{C\text{-}I}+(\log I_d+I_P/B+C_d\times C_P/B+C_d\times(I_P+C_P))\times Q_{I\text{-}C}$ ~~$+(\log I_d+I_P/B+I_P\times(\log P_I+\log P_d))\times U_I$~~ $> C_I/B\times Q_{C\text{-}I}+C_d\times\log C_I\times Q_{I\text{-}C}+$ ~~$(\log I_d+I_P/B+I_P\times(\log P_d+\log P_I)+C_d\times\log C_I)\times U_I$~~ (1)

then the third data structure will require less time. We can analyze expression (1) in general as following:

In most cases, Cp< Ip, $I_d\approx C_d\times C_I\approx P_d\times P_I/C$, $P_I\approx C_I$ and in the current case B is bigger than 64K/4 or 32K/4 (B is the node size of the B^+ tree, it is less than the size of the system's memory). So $\log C_I$ is much smaller than C_I, and I_P is also much less than B. This means:

$(C_P/B+(\log P_d+P_I/B)+(C_P-1)\times(\log P_d+\log P_I)\times P_I)\times Q_{C\text{-}I}\approx(C_P\times P_I)\times Q_{C\text{-}I}\approx(C_P\times C_I)\times Q_{C\text{-}I}$

and

$(\log I_d+I_P/B+C_d\times C_P/B+C_d\times(I_P+C_P))\times Q_{I\text{-}C}\approx(\log I_d+2\times C_d\times C_P)\times Q_{I\text{-}C}$

So, we can reduce expression (1) to

$(C_P \times C_I) \times Q_{C\text{-}I} + (\log I_d + 2 \times C_d \times C_P) \times Q_{I\text{-}C} > C_I/B \times Q_{C\text{-}I} + C_d \times \log C_I \times Q_{I\text{-}C} + C_d \times \log C_I \times U_I$

Since $C_I/B \times Q_{C\text{-}I} << (C_P \times C_I) \times Q_{C\text{-}I}$, so we do not consider $C_I/B \times Q_{C\text{-}I}$. Then expression (1) can be approximately rewritten as

$(C_P \times C_I) \times Q_{C\text{-}I} + (\log I_d + 2 \times C_d \times C_P) \times Q_{I\text{-}C} > C_d \times \log C_I \times Q_{I\text{-}C} + C_d \times \log C_I \times U_I$ (2)

$C_P \times C_I \times Q_{C\text{-}I} + (\log I_d + 2 \times C_d \times C_P - C_d \times \log C_I) \times Q_{I\text{-}C} > C_d \times \log C_I \times U_I$ (3)

In a large database system, assume C_P=10, C_I=1000, C_d=20-100 (we assume C_d=50), $I_d \approx$ 1,000,000-1,000,000,000.

So $\log I_d \approx$ 3-4 and $\log C_I \approx$ 1-2

Then expression (3) means

$10 \times 1000/C \times Q_{C\text{-}I} + (3 + 2 \times C_d \times 10 - C_d)/C_d \times Q_{I\text{-}C} > U_I$,

or

$20 \times (Q_{C\text{-}I} + Q_{I\text{-}C}) > U_I$, $(Q_{C\text{-}I} + Q_{I\text{-}C}) > \frac{1}{20} * U_I$ (4)

which is our result in the 'query is much more' case. That is, the sum of the proportion of the queries of the database operations comprised of operations involving instances some classes are bigger than $\frac{1}{20}$ the proportion of database operations comprised of 'update an instance', then the third base data structure is better. Otherwise, the first base data structure is better.

From the previous comparison, we do not care about the query operations on the instance layer, because they are the same in these operations for two data structures.

5.2.3 Update Operation Dominate

In this case, the choice is the first or the second data structure. Since update operations are much more frequent than query operations, we need not be concerned about the difference between the query operations of the two data structures. It is only necessary to compare the difference in the complexity of update operations. According to the previous analysis, the time needed for update operations in the two data structures can be expressed in the follow expressions,

DS1 $(\log I_d + I_P/B + I_P \times (\log P_I + \log P_d)) \times U_I$ (5)

DS2 $P_d \times \log P_I \times U_I$ (6)

In general $P_d >> I_P$, and $\log I_d << P_d$. Therefore, expression (5) < expression (6). That means the second base data structure is slower than the first base data structure for both queries and updates. So the first data structure is the good choice for this case.

We have compared the three base data structures in different application cases. To summarize, we suggest the following.

1. If the database is small and only one base data structure (the second base data structure) can store the database in one access, then the second base data structure is the best choice. Otherwise:
2. If query operations are more common in a database system, and the update operation is not more than twenty times the sum of the proportion of the 'instances of a class' operations and the 'classes an instance belongs to' operations, then the third base data

structure is the best choice. Otherwise:

3. The first base data structure is the best choice.

This chapter completes the analysis of the base data structures. We have proposed three base data structures and given some suggestions for selecting them. In the next chapter, we begin to discuss how to implement an instance-based database system.

Chapter 6

iQL Language

Prior to the implementation of relational database management systems, no standard query language existed, with different systems using different query methods. Any user wanting to query different systems needed to understand the structure of different databases. This inhibited user acceptance of new database systems, since understanding a new system required a significant investment of time and effort. The SQL language is considered one of the major reasons for the success of relational databases in the commercial world. Because it became a standard for relational databases, users were less concerned about migrating their database applications from other types of database systems to relational systems. If users became dissatisfied with the particular relational DBMS product they chose to use, converting to another relational DBMS would not be expected to be too expensive and time consuming, since both systems would follow the same language standards. Another advantage of using such a standard is that users can access data stored in two or more relational DBMSs using the same statements in a database application program without having to change the statements.

In this chapter, we propose an SQL-like language, called iQL (instance-based query language), for the instance-based database model. This language supports most of standard SQL, and provides some additional queries not supported by SQL.

6.1 Query

6.1.1 SQL-like Query Capability

The SQL standard depends on the relational data model. Any SQL query command refers to classes (relations, tables). This is evident from the general form of an SQL query command:

Select <*attribute and function list*>
From <*table list*>
[*Where* <*condition*>]
[*Group by* <*grouping attribute(s)*>]
[*Having* <*group condition*>]
[*Order by* <*attribute list*>];

This general form indicates that: a query can consist of up to six clauses, but only the first two, select and from, are mandatory. The select clause lists the attributes to be retrieved or functions to be computed. The from-clause specifies all relations (classes) needed in the query. The last three clauses only relate to how to output the results set, and do not affect the query capability. Moreover, any attribute in a query command must belong to one class in the from-clause. For example, a query

Select first_name, birthday
From employee

indicates that the query operates on the class employee. Only instances belonging to the relation (class) employee will be selected, and the attributes first_name and birthday must be defined with respect to relation (class) employee.

The instance-based data model supports all SQL query operations applied in the relational model, and relational model queries have counterparts in the instance-based

model [Parsons & Wand 2000]. For example, a simple query, what is the student name and student number if student name is 'John', is expressed in SQL as:

Select Name, Student_number from Student where Name='John';

The instance-based model also uses this expression on this type of query. This query can be explained as an expression in the instance-based model:

Form((Mem (Student) ∩ Scope (Name='John')) ∩ Scope(Student_number), {Name, Student_number})

This query has several steps: First, retrieve the property Name to find which instance possesses value 'John', that is Scope (Name | Name='John'). There is no method for finding the value 'John' directly in any base data structure in the instance-based data model, so this step needs to retrieve all keys (that is instance ID) to get an instance set. Second, retrieve the instances of the class student (**Mem**(Student)). Third, check which instances retrieved in the first step belong to the class Student. If any instance does not possess all properties in the class Student definition, delete it from the instances set. Next, if the class Student definition does not include property Student_number, then check all instances in the set. If any instance does not possess the property, delete it from the instance set. Finally, retrieve the values of property Name and Student_number for each instance in the instances set, and output it. The above four steps implement the query. However, in a real database, optimization methods may be different, so the order of the steps may not be the same. This example includes two basic types of operations of the

relational model, that is select and project.

Another example illustrates the join operation. Consider a query to find the Name of students who take the course Comp1700. This example query is expressed by a standard SQL as:

Select Student_Name from Student, Course where Student_ID=Course.S_ID and Course.number=Comp1700;

In this case, the iQL language is somewhat different from the standard SQL. In the relational model, we express the relationships between the classes (Student and Course) using a foreign key. Any class C1 refers to another class C2 if and only if there is a foreign key of C1 that references C2 (or there is a foreign key of C2 that references C1), and any relation between the instances is expressed by the classes. However in the instance-based model, one instance relates to another instance since they share a mutual property. It does not matter whether they belong to classes. Thus in this model, one instance may share more than one mutual property with another instance. Alternatively, an instance may not relate to other instances. So if a query needs to refer to some mutual properties, they must be indicated. To illustrate, the above query will be expressed in iQL as:

Select Name from Student, Course sharing Take_Course

where Course_number = Comp1700;

Here, the word "sharing" indicates the mutual property on which the query is based. This

query can be explained as an expression:

Form (Mem(Student) ∩ Linst ((Mem(Course) ∩ Scope (Course_number=Comp1700)), Take_course) ∩ Scope(Name),{Name})

This query can be implemented in several steps: First, retrieve the property Course_number to find which instances possess value Comp1700, that is Scope (Course_number | Course_number = 'Comp1700'). This returns a set of instances (possibly one). Second, check which instances in the set belong to the class Course, this step deletes instances which do not belong to the class Course from the set. Third, get another instance sharing the mutual property Take_course with each instance in the set. This step retrieves a new set of instances. The next step is checking which instances from this set belong to the class Student. If any instance does not belong to the class Student, delete it from the instance set. Then, if class Student definition does not include the property Name, check which instances possess this property. If any instance does not, delete it from the set. Finally, to get the results, we retrieve the values of property Name for each instance in the instance set, and output it.

From above two examples, we see that, in the instance-based model, a query command can be decomposed into a sequence of basic query operations discussed in Chapter 4 to get the desired results.

In this section, we have discussed the same queries that the instance-based database model shares with the relational model. In the next section, we discuss two types of

queries that are unique to the instance-based model.

6.1.2 Unique Query Capability

Since the instance-based model supplies two layers in a database, with instances independent of any class and classes independent of any instance, the model has more powerful query capabilities than SQL. We provide two types of queries that are not available in class based models.

6.1.2.1 Property Query

In class based models, any query is related to classes. So in the above general query model, any query command needs a from-clause in the standard SQL. In class models, there is no method to retrieve from a database all instances that have certain properties, because all instances may be distributed over many classes, and every property is defined with respect to a class. In a class-based model, it is necessary only that a property identifier is unique within a class. Thus in a database, it may be that some properties that are different properties have the same names, and some properties that are the same properties (i.e. they have the same semantics) have different names in different classes. So even querying all classes, there is no guaranteed way of combining them to get correct results.

In the instance-based model, the instance layer stores all instance information. Any instance or property identifier is assigned by the instance engine. This engine guarantees

that each identifier is unique in a database. It is therefore possible to query the whole instance layer to find which instances possess some properties. All instances that possess a property must store it using the same name. So the query will be solved by retrieval at the instance layer. For example, if querying which instances possess the property 'age', the command is showed as follow:

Select age;

Using any base data structure of the instance-based model, the operation involves checking the instance layer to find the property 'age', and output all (instance; value) pairs stored in the property 'age'. Of course, a query can add some conditions for selection. For example, a query

Select age Where age=30;

is also allowed in the instance-based model. More generally, any restriction clause in the SQL model can be added to an iQL command as a condition.

In iQL, a query that only has one clause (select clause) is enough to retrieve results. Other clauses are optional conditions of a query. This is not same as SQL, in which a query must have at least two clauses (select and from).

6.1.2.2 Limited and Unlimited Query

To implement a join operation in SQL, the from clause must declare all classes (relations) and the where clause must declare all properties used to join these classes (otherwise, a query *Select * from a, b;* produces a Cartesian product. Such a result can be very large,

and generally is not needed). The instance-based model supports two types of join operations, which we call limited queries and unlimited queries. A query that declares the mutual properties on which the join is based is called a **limited query**. This type of query mirrors an SQL query that includes some join operations, and has already been discussed. However, iQL supports a type of join query that we call an **unlimited query**, in which the query command does not declare mutual properties of the join. Obviously, SQL (and class based models in general) does not support unlimited query, because a DBMS that supports SQL will not be able to perform a join unless the join conditions are specified in the query. In the instance-based model, two instances jointly relate to each other via mutual properties, and mutual properties are stored by the instance layer. Therefore, instances can be related without reference to classes, and no foreign key is needed.

For example, the limited queries of the above examples may be shown as below:

Select P1, P2 sharing mutualproperty1; (4)

Expression (4) means "if there are two instances, one possessing P1 another possessing P2, and the two instances share a mutual property mutualproperty1, then output the pair consisting of the values of P1 and P2". The limited query is analogous to the join operation in the relational model. However, an unlimited query is different. For example, the unlimited queries of the above examples may be shown as below:

Select P1, P2 From C1, C2; or

Select P1, P2; (5)

Expression (5) means "if there are two instances, one (instance1) possessing P1 and another (instance2) possessing P2, and the two instances share some mutual properties, then output the triple of (value of instance1 possessing P1, name of the mutual property shared by two instances, value of instance2 possessing P2". The name "unlimited query" means that no mutual property is declared or defined in the query. The primary advantage of unlimited query is that it offers "join" query in which no limiting conditions are specified.

In this section we introduced a concept of unlimited query. The next section we present two rules for implementing unlimited query.

6.2 Rules for Implementing Unlimited Query

In the instance-based model, each instance is stored in the instance layer independently. Two cases may arise. First, sometimes some instances possess a certain set of properties; however, some other instances possess a subset of the properties of the set but are linked via mutual properties to some instances that possess another subset of the properties of the set. For example, suppose there are three instances: a student has a property Weight (75KG), a building has a property Color (yellow), a washing machine has two properties Weight (50KG) and Color (white), and the database stores information that the student has an office located in the building with mutual property Office-in. The question arises, what will be the result of a query such as:

Select Color, Weight;?

The results will be values of the properties of an instance (washing machine) or the related values that the two instances (student and building) possess each property. This type of problem is called the 'covering' problem. To get a clear semantics for such a query, we must define a rule in the instance-based model.

Rule 1. (Cover Rule) If two or more properties appear in an unlimited query command and there are some instances that possess all the properties, then only the values of the instances that possess all the properties will list to the results.

For example, if there are some data in an instance-based database as in Figure 10. An unlimited query

Select Property1, Property2;

will return the result:

Property1	Property2
a	c

Instance1 possesses Property1 and Property2, so even though instance4 and instance2 are linked with mutualproperty3, the result will not include the values of these properties for the linked instances. We need Rule 1 because, in the instance-based model, all instances are stored in the instance layer, and each instance possesses its properties independently. Each property identifier is unique in the whole database system. Therefore, there may be a case such that an instance is jointly related to another instance, and each one possesses a subset of properties in the select clause of our query, but there is another instance

possessing all properties possessed by the previous two instances. The first rule gives a method to resolve potential ambiguity in this situation. It reduces many complexities of unlimited queries, and gives them a clear semantics.

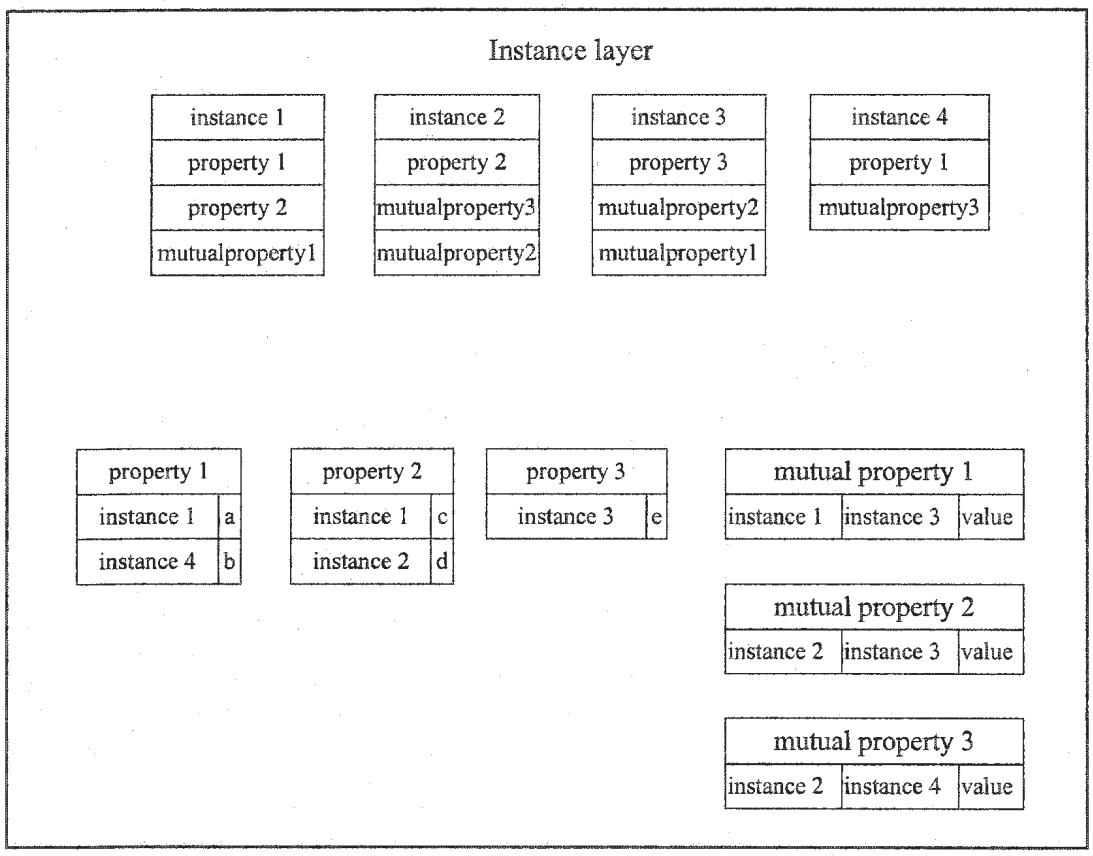


Figure 10: An example of the database where there exists a 'Covering' problem

In the instance-based model, to implement query operation, there exists another problem, which we call the 'order' problem. The 'order' problem reflects the second case mentioned above. That is, sometimes there is an instance set such that all instances jointly relate to another with different mutual properties (e.g. in Figure 11, instance1 and instance2 jointly possess mutualproperty1, instance2 and instance3 jointly possess

mutualproperty2, instance1 and instance4 jointly possess mutualproperty3, and so on). However, there may be more than one instance possessing a subset of properties that appear in the select clause of a query. For example, consider some data stored in the instance layer as shown in Figure 11. We pose the question, what will be the difference between two queries such as:

Select Property2, Property1, Property3; and

Select Property1, Property2, Property3;?

To support unlimited queries, there are two rules to be applied.

Rule 2. (Order Rule) We check whether an instance links to another instance according to the order of the properties in the select clause.

By this rule, the above query

Select Property2, Property1, Property3;

will give the results:

Property2		Property1		Property3
b	mutualproperty1	a	mutualproperty3	d

But an unlimited query

Select Property1, Property2, Property3;

will get results

Property1		Property2		Property3
a	mutualproperty1	b	mutualproperty2	c

In the two queries above, the properties are not in the same order at the select clause, so the results are not the same. A real world example is shown in Figure 12, there are five

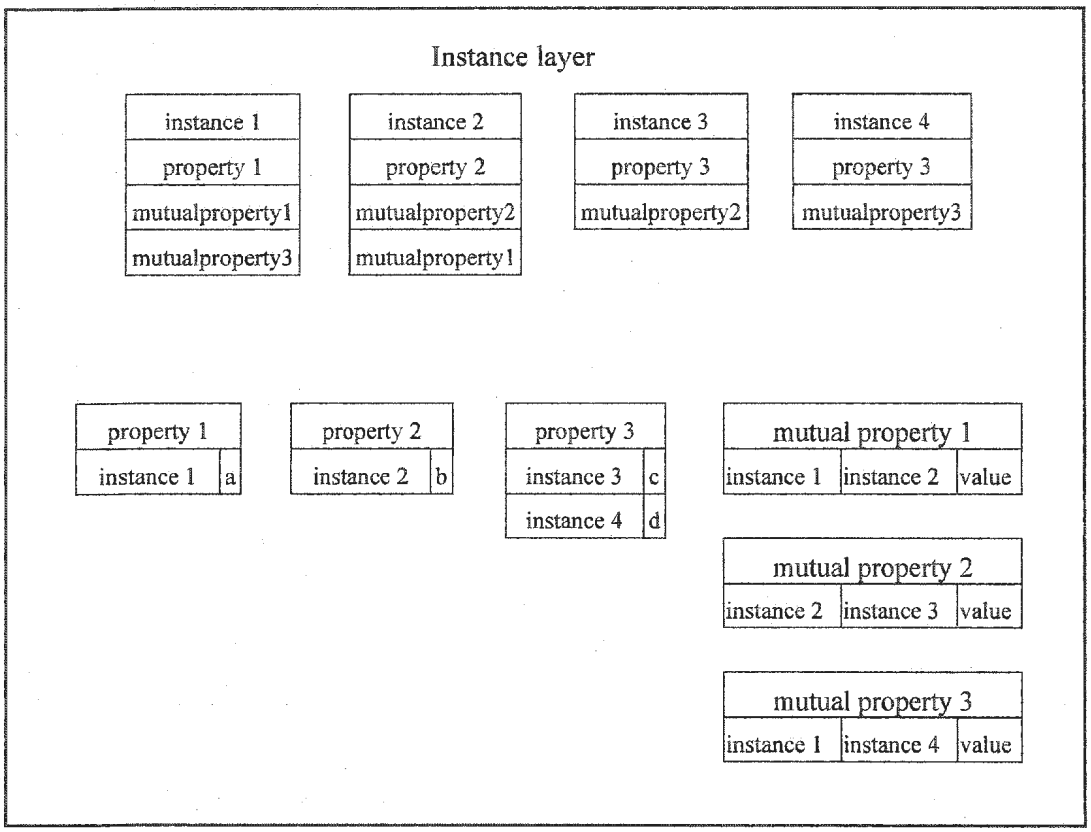


Figure 11: An example of the database where there exists a 'Order' problem

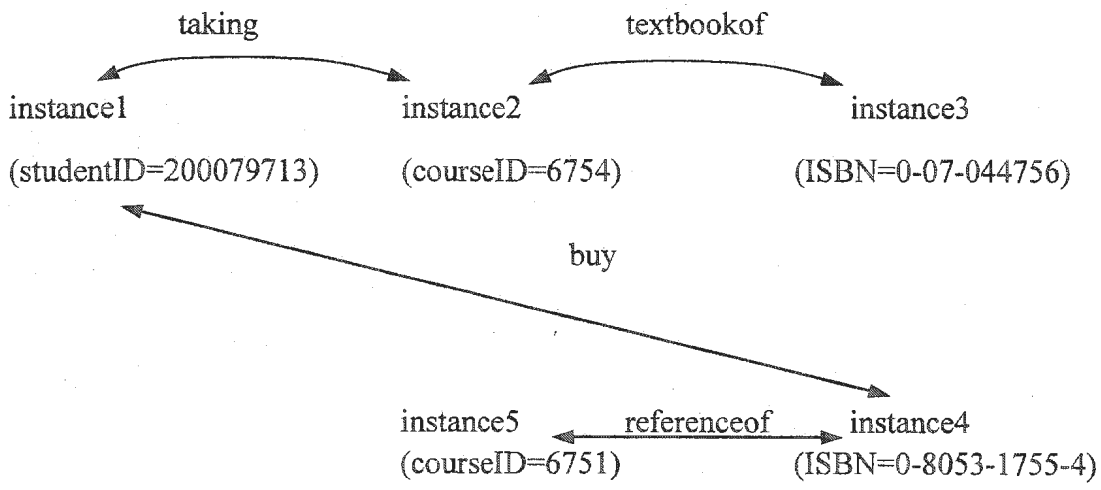


Figure 12: Methods for mutual properties select

instances: instance1 is a student, it possesses a studentID (200079713); instance2 and instance5 are courses, they respectively possess courseID 6754 and 6751; instance3 and instance4 are books, they respectively have ISBN 0-07-044756 and 0-8053-1755-4. The relations between the instances are showed by arrows, and the names of relations (mutual properties) are above each arrow. According to rule2, an unlimited query:

Select studentID, courseID, ISBN;

will get result:

studentID		courseID		ISBN
200079713	taking	6754	textbookof	0-07-044756

Even if the instance1 is related to instance4 (the student buy the book) and instance4 is related to instance5 (the book is a reference of the course), they still not list to the result, because the order of the relations are not in the order of properties in the select clause.

Rule 2 (Order rule) is important for the instance-based model, because the model can express very complex relations between instances. The order rule gives a condition for retrieving mutual properties, that is, which mutual property is the first choice and which is the second and so on. The system returns the results based on this condition.

Rule 3. (Range Rule) Only 'direct' mutual properties are used for retrieving.

Here, direct mutual properties means each instance that will be retrieved must possess at least one property which is in the select clause. The instance-based model can contain very complex relations between instances. An instance may be related to many other instances, and sometimes there may be an instance that is related to all other instances of

a database. We call this problem a 'range' problem. So we must define which instances are the set that we will retrieve when we implement a query. Rule 3 just gives the range. This rule indicates that a query will only check the mutual properties between the instances that must possess at least one property in the command. For example, if there are data as in Figure 13, a query:

Select Property1, Property2;

will return results

Property1		Property2
a	mutualproperty1	c

Although instance4 is linked to instance3 with mutualproperty2 and instance3 is linked to instance1 with mutualproperty1, it is not a direct link. Therefore, the results will not include it.

In Figure 14, we use real data showing the meaning of direct mutual property. A student (possessing a property Name) may buy or borrow a book (possessing a property ISBN). A student may also take a course (possessing a property courseID), and a course may use a book as a textbook. Then the results of a query *Select Name, ISBN;* should include only the direct associations between people and books. The results should not include the names of students and ISBNs of books where for students who take some courses that use some books, because a 'student takes a course' is not a direct mutual property. In Figure 14, a bold arrow shows the direct mutual property, while a thin arrow shows an indirect

mutual property.

In this section we provided three rules for implementing the unlimited query. In the next two sections, we discuss how to implement query and update operations.

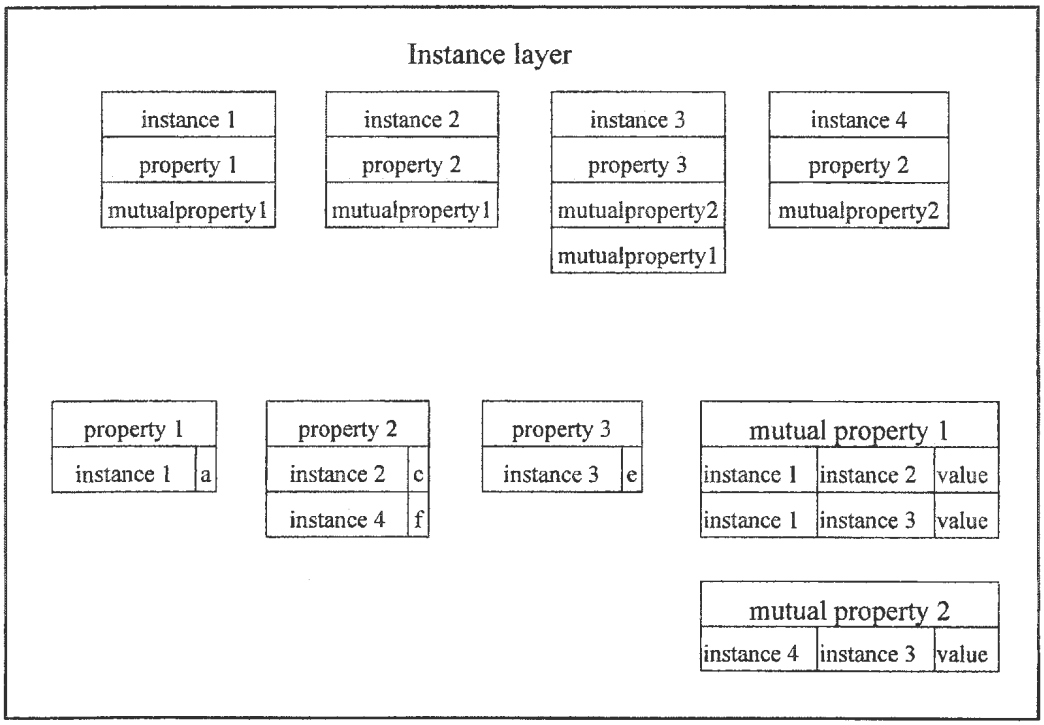
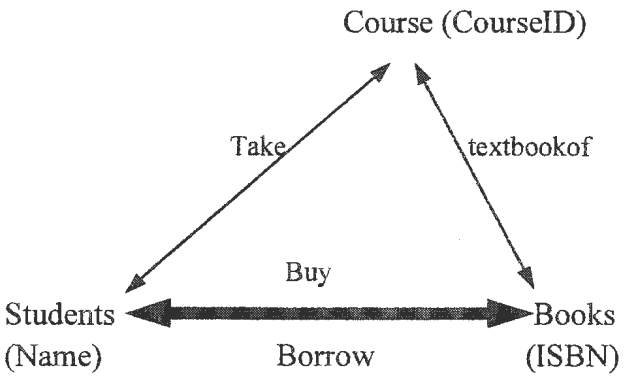


Figure 13: An example of the database having 'Range' problem



Query: Select Name, ISBN;

results:

Name		ISBN
John	buy	0-768-03141-8
Brown	borrow	0-786-03141-6
Smith	borrow	0-669-03141-9
Rabbit	buy	0-889-03141-2

Figure 14: A direct linked mutual property

6.3 Query Implementation

The three rules reduce some complexity of unlimited query. However, implementing this type of query is costly. Assume an unlimited query is: select P_1, P_2, P_3, …, P_n; the steps of implementation are as below:

First, find the set (set1) of instances that possess properties from P_1 to P_m ($1<m<n$). Second, if there is a property P_{m+1} ($m+1<n$) such that no instance in set1 possesses it, and there are some instances (set2) that possess P_{m+1} to P_{m+g} ($m<m+g<n$), then check whether there is a mutual property jointly possessed by a pair of instances s_1 and s_2 ($s_1 \in$ set1, $s_2 \in$ set2). If there is a pair of instances jointly related with a mutual property mp, add the pair to a linked instances set $S_1\{s_1, mp, s_2 \mid s_1 \in set1, s_2 \in set2\}$. Here, if the pair of instances jointly possesses more than one mutual property, we only list one. Third, if there are some instances (set3) possessing P_{m+g+1} to P_p ($m+g<p<n$), we repeat the second step. However, when we check, we only check whether there is a mutual property between the second element of a pair in S_1 and the instances in set3 to form a new linked instances set $S_2\{ s_1, mp, s'_2, mp_1, s_3 \mid s_1, mp, s'_2 \in S_1$, and mp_1 is a mutual property$\}$. Repeat the third step until all properties in the select clause are possessed by the linked instances set. Then, the results will be the values of the each element (that is a line of linked instances) possessing the properties in the select clause. If there is no element in the linked instances set, then return null.

This algorithm adheres to Rule1, Rule2 and Rule3. The first part of each step uses

Rule 1 to check the results, the last part of each step uses Rule 2 and Rule 3 to check the results. In a commercial implementation of an instance-based DBMS, there may be methods for optimizing each query, but the main process is as above.

The algorithm lets the unlimited query produce results analogous to the multi-join operation in the relational model. The first step is to find the classes (instance sets, that is tables in the relational model), the second step is to find the conditions of join. However, in the unlimited query, there are two differences compared to the join opertion in the relational model: first, the instance sets (classes or tables) are not indicated by users, they are automatically decided by the properties in the select clause and data in the database; second, the relationships between the two sets are different. In the instance-based model, the relationships are based on instances, so even if they are in the same set (selected in the first step) they may have different relationships with the instance in another set. Therefore, the meaning of the unlimited query is different. A result in an unlimited query means: either some instances possess the properties, or some instances possess part of the properties and are linked to each other with some mutual properties. For example, if a database stores some information about a person (name, age) and a department (location, city), and a person may work, study, visit a department, then an unlimited query: *select name, city;* will give results as show in Figure 15. From Figure 15, we interpret the semantics of this query to be the associations between people and cities.

Name		City
Johnson	study-at	St. John's
Brown	study-at	Halifax
Smith	works-at	St. John's
Williams	visit-to	Toronto
Rabbit	visit-to	Ottawa

Figure 15: An example of the result of an unlimited query

Unlimited queries can help us if we do not know the schema of a database. However, this operation is very costly. In many cases, a query may need some condition for reducing the number of instances that need to be checked in the query, or to limit the links that need to be checked. All these conditions are applied to the query by adding a from-clause, where-clause and sharing-clause in the instance-based model. The from-clause and where-clause are the same as the standard SQL in the relational database model. Although the relation between instances and classes is different, the two clauses give the same meaning in queries on both the instance-based model and class-based models. Only the sharing-clause is unique to the instance-based model.

In the relational model, a single property may be stored by many entity types. In fact, many of these properties are not intrinsic properties. They are mutual properties between two instances. However, there is no mutual property concept in the relational model. The information that an instance is related to another instance (or one class is related to another class) is stored by a foreign key. Many classical papers of the relational model

explain this method (e.g., [Markowitz & Makowsky 1990] [Johanneson 1994] [Date and Hopewell 1970]). A join operation on different entity types just links instances using these properties, and if a query involves more than one table (or class), the join operation is needed by the query command. If there is no join condition, the join will not be implemented. This condition is added to the where clause in the form property1=property2, typically involving different relations, not a select condition of the form property1=value. In the instance-based model, an instance is stored in the instance layer, and an instance is related to other instances with mutual properties. Therefore, if we need to add some conditions involving an instance's relationship to others, the query command must include some mutual properties. These mutual properties are included with the sharing clause, which follows the from clause in our iQL query model.

A mutual property expresses two types of information between two instances: one is that the two instances are jointly related to each other, the other is that there is a value resulting from their relation. Therefore, there are two forms that a mutual property can have in the sharing clause. In the first, only a mutual property identifier appears in the sharing clause. This means we only check the mutual property if any instance links to other instances with this mutual property. In this case, the sharing clause is of the form: sharing MP1, MP2, ..., MPn (MP1, MP2, ..., MPn are all mutual properties). In the second form, we declare both a mutual property and a value of the mutual property, which means we need to check the mutual property only if an instance is linked to another

instance with this mutual property and the value. In this case, the sharing clause is of the form: shared MP1=value1, MP1=value2, ..., MPn=valuen (MP1, MP2, ..., MPn are all mutual properties).

As in the relational model, there are many methods for optimizing the condition clause in a real database design. Such methods are beyond the scope of the thesis. However, adding a sharing clause will reduce the complexity of a query.

Each type of query supported by the instance-based model is listed in Table 8.

The instance-based database model supplies more powerful query capability than the relational model. A user can query a database more freely, using the form of unlimited query or linking query in this model. However, we do not think users can use these forms randomly. Since a powerful query will cost more time to get results, we suggest that the user add conditions to a query command as much as possible, instead of only using powerful unrestricted queries. We also discuss how to develop efficient queries in the next chapter.

6.4 Update

Since the instance-based model supports classes independent of any instance, it supports more update operations than the relation model. These update operations are divided into

Table 8 iQL Queries

Commands	Contents
Query instance layer	
select *;	list all instance forms
select P1, ..., Pn ;	list values of P1 to Pn possessed by some instances or some related instances
select * from c1, c2, ..., cn ;	list all instance forms that belong to the class c1 to cn
select P1, ..., Pn from c1, c2, ..., cn ;	list values of P1 to Pn possessed by some instances or some related instances that belong to class c1 to cn
select * where P1=[1]value and[2] P2=value and ... and Pn=value;	list all instance forms for instances that possess P1=value and P2=value and ... and pn=value
select P1, ..., Pm where P1=value and P2=value and ... and Pn=value;	list values of P1 to Pm possessed by some instances or some related instances such that they possess P1=value and P2=value and ... and Pn=value
select P1, ..., Pm from c1, c2, ..., cq where Pg=value' and Pg+1=value and ... and pn=value; (P1, ..., Pm∈P and Pg, ..., Pn∈P)	list values of P1 to Pm possessed by some instances or some related instances that belong to the class c1 to cq and possess Pg=value and Pg+1=value and ... and pn=value
select P1, ..., Pn sharing mp1 and mp2 and ... and mpn;	list values of P1 to Pn possessed by some instances that are related to each other by mutual property mp1,mp2,..., mpn.
select P1, ..., Pn from c1, c2, ..., cq sharing mp1 and mp2 and ... and mpn;	list values of P1 to Pn possessed by some instances or some related instances that belong to class c1, c2, ..., cn, and are related each other by mutual property mp1, mp2,..., mpn.
select P1, ..., Pm from c1, c2, ..., cq where Pg=value and Pg+1=value and...and Pq=value sharing mp1 and mp2 and ... and mpn; (P1, ..., Pm∈P and Pg, ..., Pq∈P)	list values of P1 to Pm possessed by some instances or some related instances that possess Pg=value, and Pg+1=value, and ..., and Pq=value, and are related to each other by mutual property mp1, mp2,..., mpn.

Table 8 (Cont)

query property belong to instance instanceID;	query the form of this instance
query mutualproperty information belong to instance instanceID;	query all mutual properties that this instance possesses with others
query mutualproperty mutualpropertyID <,*value*> sharedby instance InstanceID withothers;	query all instances that sharing a mutual property (*and value*) with the special instance
query instance share mutualproperty mutualpropertyID;	query all instances that share a special mutual property
query mutualproperty *;	query all mutual properties
Query (class layer)	
query class *;	query all classes
query property belongto class className;	query a class definition
query class has instance instanceID;	query all classes that a special instance belongs to

Note: *1. the notation '=' indicates that there we can use all comparison operations, including* =, <,>, $\neq$, $\leqslant$, $\geqslant$.

2. the notation 'and' indicates that there we can use either disjunctive or conjunctive.

three types: update instance, update property (include mutual property update), and update class. All these update operations were discussed in Chapter 4. To implement the update operation, some rules are needed.

6.5 Rules about Update

The instance-based model supports classes independent of any instance, so updating a

class will not affect the instance layer. However, since the instance layer stores all instance information, include property definitions, a property update may affect class definitions. If a property is deleted from the instance layer, its information must be lost from the database. If there is a class definition that includes this property in the class layer, then the database loses the semantics of these classes. So the first update rule is:

Rule 4. If a property (or a mutual property) is deleted from the instance layer, it must be deleted from the class layer. Any class including this property in its definition must be deleted, or its definition must be changed not to include this property.

For example, consider a class Student defined by properties set {Name, Student_ID, Sex, Age}. If, at some time, the property Age is deleted from the instance layer, Rule 4 implies that we either change the class Student definition to property set {Name, Student_ID, Sex} or delete the class Student. Otherwise, database integrity will be lost, since a class will refer to a property that no longer exists. The rule is based on the three base data structures. In these data structures, the class layer does not store property definitions. Therefore, Rule 4 maintains the integrity of DBMS.

In the instance-based database model, there is not a static database schema, the instance layer stores all instance information and the properties information. So another rule is:

Rule 5. A property exists in a database if and only if some instances possess this property (At the time we checking).

This rule indicates that property existence depends on instances. It also expresses an advantage of the instance-based model: there is no need to manage any properties when no instance possesses them, and there is also no need to manage any classes if no instances belong to them[1].

The basic update operations, such as update an instance, update a class, and update a property, were discussed in chapter 4. We call these basic update operations 'simple' update operations. Other update operations either are a simple operation or a query operation plus some simple update operation set, so are easily implemented. We do not discuss them again, but summarize them in Table 9.

This chapter discussed how to implement the iQL language in the instance-based model. We have provided two types of powerful queries that give the instance-based model advantages over class-based (notably, relational) models. Although we did not discuss the optimum methods for implementing this language, if we follow the base rules of the implementation and use a correct base data structure, we can implement an instance-based database system. In the next chapter, we will apply these rules to discuss how to implement an instance-based database, test and compare two implementations based on different base data structures.

1. This rule is according to the "pure" model. In a real database of this model, for efficient query or update we may maintain a class definition even if no instance belongs to it.

Table 9 Update operation

Command	Contents
Insert (instance layer)	
insert instance ins_ID (property1, value1, property2, value2, ···, propertyn, valuen);	insert an instance (users indicate an instance ID)
insert instance (property1, value1, property2, value2, ···, propertyn, valuen);	insert an instance (users do not indicate an instance ID)
insert property (pro1, value1, pro2, value2, ···, pron, valuen) into instance ins_ID;	insert some properties to an instance
insert mutualproperty mutualproID shared by instance insID1, insID2 value value1;	insert a relation of two instances and the value of the relation
Insert (class layer)	
insert class ClassName (property1, property2,···, propertyn);	insert a class
insert property (property1, property2, ···, propertyn) into class className;	insert a set of properties (mutual properties) to a class
Delete (instance layer)	
delete instance ins_ID;	delete an instance
delete instance from c1, c2, ···, cn ;	delete some special instances that belong to class c1, and c2, and …, and cn
delete instance where P1=value1 and P2=value1 and ··· and Pn=value1;	delete some special instances that possess P1=value1, and P2=value1, and …, and Pn=value1
delete instance from c1, c2, ···, cq where Pp=value1 and Pp+1=value1 and ···and Pn=value;	delete some special instances that belong to class class c1, and c2, and …, and cq and possess Pp=value1 and Pp+1=value1 and … and Pn=value1
delete property (properties set) from instance ins_ID ;	delete some properties from special instance
delete property (properties set);	delete properties from database
delete mutualproperty mutualproID shared by instance insID1,insID2;	delete a relation of two instances sharing a mutual property
delete mutualproperty mutualproID;	delete the mutual property
Delete (class layer)	
delete class className;	delete a special class
delete property (properties set) from class className;	delete some properties from a class

Chapter 7

Implementing, Testing and Comparing

In this chapter we describe prototype implementations of two instance-based database management systems based on the first two base data structures. The implementations include query and updating operations to demonstrate that this system can produce correct results. We also compare and validate results of the cost of operations using each of the two base data structures. Finally, we discuss how to design efficient query methods in the instance-based model.

7.1 Implementing an Instance-based Database System

7.1.1 Programming Languages for the Implementation

Many programming languages could be used to implement the instance-based database system, such as JAVA, C, or C++. In this research we have chosen to use JAVA as the implementation language in order to take advantage of JAVA's cross-platform portability.

7.1.2 Structure of Instance-based DBMS

The instance-based model is different from the relational model with respect to the design of database systems. The instance-based database system has a dynamic schema. It is managed by system managers and users. For example, system managers may define some classes for the security of systems and some for the common information query, while

users may define special classes for faster query or update data[1]. Nevertheless, designing

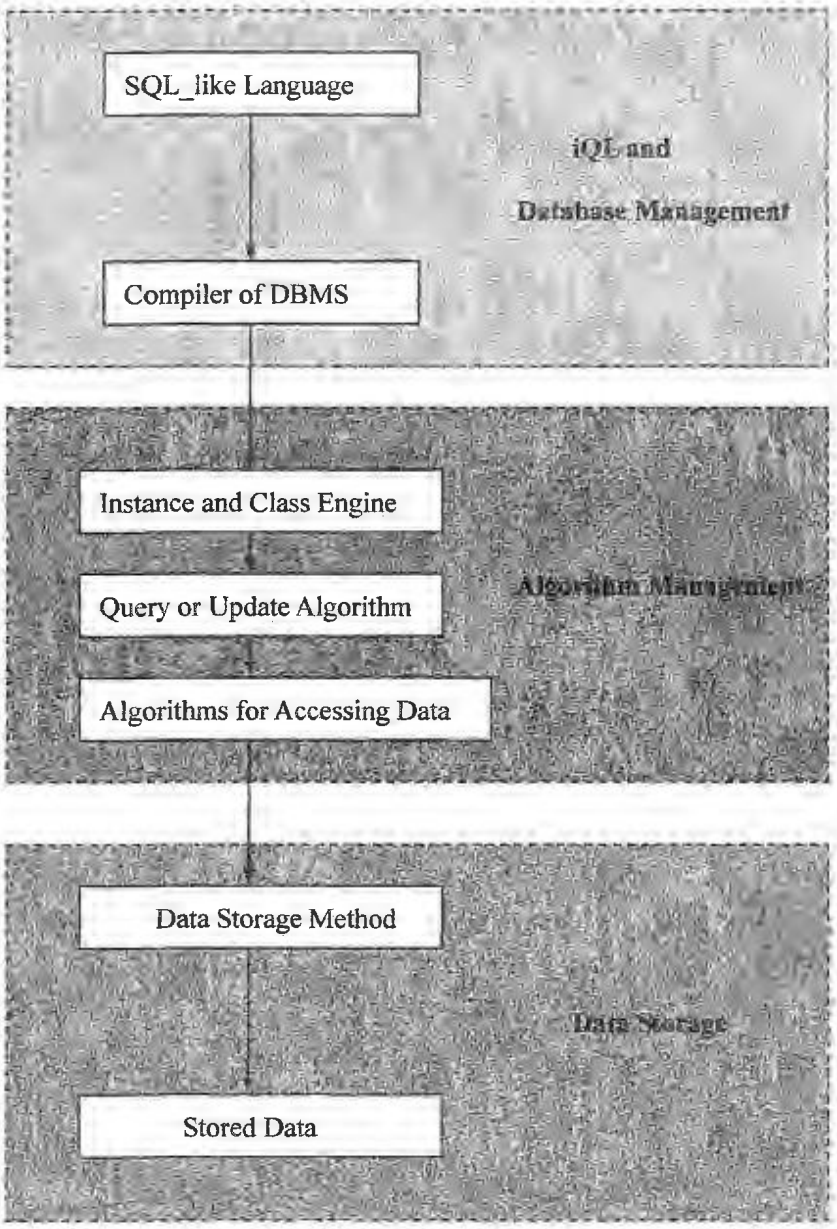


Figure 16: Instance-based DBMS structure

1. In instance-based model, the security needs some special approaches. In this thesis we do not consider this kind of approaches.

an instance-based database is much simpler than designing a relational database. The components in our implementations are shown in Figure 16.

Our instance-based database management system consists of three parts. These three portions are (from the bottom to the top of Figure 16): data storage, algorithm management, iQL language and database management. The components of each portion are as follows:

Stored Data: This component includes two parts. The first is the instance layer data, which includes instances, intrinsic properties and mutual properties, as well as the values of intrinsic and mutual properties for all instances. The second part is the class layer data. It includes only class definitions in the first and second base data structures.

Data Storage Methods: This area also includes two parts. One part is the method used to store the instance layer information. This includes a method to store instances themselves, a method to store intrinsic properties, and a method to store mutual properties. The second part is the method used to store the class layer information. This includes a method to store class definitions.

Algorithms for Accessing Data: This component is related to the data storage method. Because data access methods are based on data structures, it also includes two parts: methods to access instance layer information and methods to access the class layer. The first part includes some algorithms to access instance layer information. First, there is an algorithm that specifies how to insert, delete, and retrieve instance information. Second, there is an algorithm that specifies how to insert, delete, and retrieve intrinsic property information. Third, there is an algorithm that specifies how to insert, delete, and retrieve

mutual property information. The second part includes an algorithm to access class layer information. This includes inserting, deleting and retrieving the definition of classes. Each algorithm above must adhere to the base rules of the instance-based model.

Query or Update Algorithm: This component includes the methods and algorithms that are presented in chapter 6. Since this affects the efficiency of a query or update, optimization methods will need to be applied in this component for a database system based on this model. Such methods are beyond the scope of this thesis.

Instance and Class Engine: The instance engine manages the instance layer. In fact, the instance engine is not only algorithms to query or update the data in the instance layer but also an identification tool for the instance layer. It creates a unique identifier for each instance, intrinsic property, and mutual property in the database. For example, it creates a unique instance identifier for each instance when it is first added to the database system. Also, it deletes the identifier when the instance is deleted from the database system. In the same way, the class engine manages the class layer. It is not only algorithms about querying or updating the data in the class layer but also an identification tool for this layer. It creates a unique class identifier for each class.

Compiler: This component includes a compiler which compiles iQL commands.

iQL language: This component implements the standard language of the instance-based database. It supports standard SQL, and also has extended query and update abilities that pertain to instances independent of any classes, as presented in

chapter 6.

The last two parts are the same in the implementations for each of the two base data structures.

7.1.3 Steps for Implementing an Instance-based Database System

The instance-based database system design follows the structure shown in Figure 16, from the bottom to the top. The steps are:

1. Select a base data structure: According to the discussion in the chapter 5, different base data structures will vary in efficiency for query and update operations.
2. Design the data storage structure and data access methods: In this step, the first activity is to design a structure for storing data. This structure declares for each layer where data will be stored, and which type of files will be used for data storage. The second activity is to design data access methods for each type of file. Different data access methods may have different cost and components. For example, a hash table is very fast for accessing, but incurs a high cost in storage space. In contrast, linked lists are very slow for accessing, but need relatively little storage space.
3. Design an instance engine and a class engine: A different instance engine and class engine will be designed for different data storage structures in this step. The engines guarantee that each instance, property and class are unique in the

database.

4. Implement algorithms: This step involves implementing all algorithms for query and update. It may also involve optimizing these algorithms for different base data structures.

5. Design a compiler for the iQL language. Since this language is the standard language of the instance-based database model, the compiler does not need to design for every instance-based DBMS. The iQL compiler is a standard compiler for the instance-based database system, and can be used by any instance-based database system implementation.

The five steps above are needed to implement an instance database system. However, we note that, in the instance-based database model, if no instance has been added to a database system, then neither a database nor a database schema exists. In contrast, in the relational model, even if there are no records in the database, the schema can exist, and relationships between tables can exist.

7.2 Implementing two Database Systems Using two Base Data Structures

In this section, we will describe our implementation of two database systems using the first two base data structures presented in Chapter 3. The basic structures and data are described next.

All base data are stored as files, either at the instance layer or at the class layer. Each

type of file is stored in one directory. So in the first base data structure, there are four directories in the database. They are instance, property, mutual property and class. However in the second base data structure, there are only three directories in the database: intrinsic property, mutual property and class respectively. In general, each class stores only one class definition and a class definition includes only a few properties or mutual properties (e.g., ten to twenty properties in a class). Therefore, each class file stores the class definition information (a class is defined by properties or mutual properties) in a linked list. Similarly, each instance file stores the property pointers or mutual property pointers that the instance possesses in a linked list.

For each property file and mutual property file, things are quite different. An intrinsic property file may store a few or many values, perhaps thousands or millions, possessed by instances. A mutual property file can also store a few or many values shared by some instances. Therefore, for fast querying and updating, in each intrinsic property file and mutual property file, after comparing the advantages of each ordered indexing structure, we decided to store data as a B^+ tree [Silberschatz, Korth and Sudarshan 1996]. In the tree of an intrinsic property file, the instance identifier is the key. And in the mutual property file, two instance identifiers, the pair sharing the mutual property, are combined to form a key. All methods for accessing data are based on the above structures. After implementing all algorithms for each base data structure, we can build a standard iQL language and a compiler for this language in the instance-based database model. Of course, each

algorithm will need some optimization methods added for fast querying or updating operation to be practically or commercially feasible. However, such enhancements are beyond the scope of this thesis.

At the end of the design, data were added into each of the database systems to initialize them. We added the data shown in Appendix 1 to each database system (the data are mostly same as the Figure 7.6 in [Elmasri and Navathe 2000]). In Appendix 1, an instance is expressed as an instance identifier followed by the pairs of intrinsic property identifier and value of this instance possesses the property. A relation between instances is expressed as a mutual property of two instances. A class is expressed as a class identifier followed by the property identifiers of the properties in the class definition. After adding data to each database system, two instance-based database systems have been built. Each of them is based on one of the two base data structure presented earlier.

7.3 Testing

The purpose of testing is to check the query and update capabilities of the implemented databases. All commands in Table 8 and Table 9 in Chapter 6 are tested and some of the results are listed as examples in Appendix 2.

Some commands of iQL language listed in Table 9 are not implemented in this project. They can be described as composite commands. Since they are all based on the others commands in the Table 9, all commands of this type can be implemented in the

instance-based database system. This is because in this model, any instance, or property, or class has a unique identifier. So combining each type will produce no confusion. For example, a composite command:

Delete instance from C1 where P1=A and P2 in (select P2 from C2 where P3=B),

is composed of two basic commands. One command selects P2 from C2 where P3=B while another command deletes the instance from C1 where P1=A and P2=results of the first command.

From the results list, we know that commands of query or update are implemented correctly. However the detail of the implementation of mutual properties has not been discussed in this thesis. Because the mutual property concepts are different from the concepts of the relation in the relational model, we provide some details about implementing mutual properties in the instance model in the next section.

7.4 Implementing Mutual Properties

Thus far, we have not considered how to answer queries such as " who has a supervisor?" in the instance-based model in the example databases. In the previous chapters, we did not consider in detail how to implement mutual properties. It seems that the mutual property stores information that is not suited for this type of query. This is because, if only a mutual property identifier is stored in an instance when the instance possesses this mutual property with another instance, then there is no information about which instance

is an "active" instance and which is a "passive" instance of the mutual property[1]. There is no method to differentiate the instances if the query involves a mutual property itself. However, in the instance-based database system, we can answer this query correctly. In this section, we consider how to store mutual properties information in each layer.

Two types of information about mutual properties need to be stored. The first involves the way in which a mutual property stores instances. Instead of randomly storing instances to form the key in a mutual property, in our implementation, the mutual property always stores an instance which is an 'active' instance in the first column. Another instance which is a 'passive' instance is stored in the second column when they are combined to form a key of the mutual property. This will give information to the mutual property itself about which instance is an active instance. For example, a mutual property Supervise may store its value as shown in Figure 17.

Supervises		
instance 1	instance 2	1999, 08, 20
instance 1	instance 3	2000, 01,05
instance 4	instance 3	1997, 07, 01

Figure 17: Mutual property Supervise

In Figure 17, all instances in the first column are employees, that means instance1 and

1. We do not think there are an "active" and a "passive" side for all mutual property in the real world. However, for efficiency we can define the sides when we defining a mutual property.

instance4 are employees. The instances in the second column are supervisors, which means instance2 and instance3 are supervisors. And the third column contains values of a supervisor related to an employee (e.g., date on which supervisor commenced). There is no confusion.

The second kind of information is about the way in which the instances or classes store mutual property information. From the above, we know that instances possessing a mutual property must be divided into two sets (for binary mutual property). The first set is active instances. Another set is passive instances. The question becomes how to store this information to each set of instances. By ontology, when two things interact, one may cause the other to change. Changes to things are manifested as changes to properties. We can find a definition in ontology as below:

Definition: *Thing X acts on thing Y if and only if the states that Y traverses for a given subset of M (M is a set of time instances) when X is present are different from the states that Y would traverse if the thing X did not exist. Things X and Y interact if at least one acts on the other.* [Wand Storey and Weber 1999]

That means if two instances interact then at least one gets a new state. For example, consider an instance1 that is a student (first state), and another instance2 that is a book. If the student borrows the book, the instance1 enters a new state 'borrower' and the instance2 enters new state 'borrowed book'. This change is showed in Figure 18.

According to the ontology, a state of an instance is a set of properties. So an instance

changes to a new state by interacting with another. That means if an instance is jointly related with another instance, it must possess some properties.

By the mutual property definition, a mutual property expresses the fact that an instance is related to another instance. So if an instance possesses a mutual property with another instance, this means the two instances interact. Therefore, each instances, either the active instance or the passive instance, must have a set of special properties since they belong to a side of the mutual property. We define that the active instance possesses a set of properties named "attaching another instance with the mutual property", and express this set of properties as one notation. That is the mutual property identifier plus a "->".

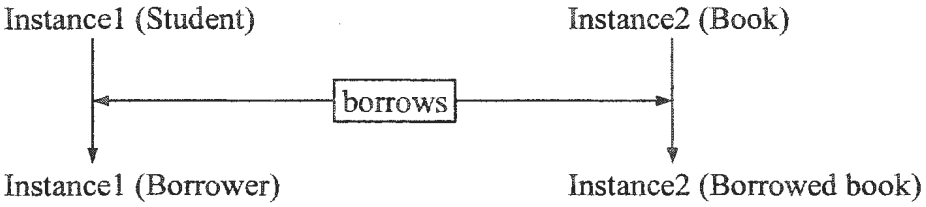


Figure 18: Instances change their states by acquiring a mutual property

Also we define that the passive instance possesses a set of properties named "attached by another instance with the mutual property", and express this set of properties with the mutual property identifier plus a "<-". For example, in Figure 17, instance3 is a supervisor of instance1, so instance3 possesses an additional property 'Supervise->', and instance1 possesses a additional property 'Supervise<-'. And if instance3 is a supervisor

of more than one instance, it only possesses one property 'Supervise->'. The same is true for a passive instance. By the ontology definition, we also know that the existence of additional properties is based on the mutual property. If an instance no longer possesses a mutual property with other instances it loses the additional property of this mutual property at the same time. Since mutual properties have this property, we get a solution for storing mutual property information into an instance. If an instance possesses a mutual property with other instances then we store the additional property to the instance.

In the class layer, we also need a method for storing mutual property information in a class definition. By the ontology, a class classifies a set of things into two subsets. One subset is the things that possess all properties in the class definition. They are the things in the class. Another subset is the things that do not belong to the first subset. They do not belong to the class. Since a mutual property expresses the relation of two sets of instances (e.g. relation of active instances and passive instances), if we only store a mutual property identifier to a class definition, the class definition will not be clear about which instances will be included in the class, active instances or passive instances or both of them. In fact, a class including a mutual property also indicates that it includes only one set of the instances of the mutual property, either active instances or passive instances. For example, a class Students means a set of persons who study in a school. This class may include a mutual property, Study-at, expressing a relation between persons and schools. However, the class only indicates one set of instances, persons, in the class. By this meaning, a class

definition can also store the additional property of a mutual property as a part of the class definition to indicate the mutual property. So if a class Student includes a mutual property Study-at, the class definition then includes an additional property Study-at->.

We know that an instance possessing a mutual property with others may have more than one additional property. However, only two property identifiers, mutual property identifier plus a "->" and mutual property identifier plus a "<-", are considered to apply to instances or classes. Since the forms of the two identifiers are very similar to the form of the mutual property identifier, it is very convenient for algorithms to determine to which mutual property the additional property is related.

When we apply these two parts of the method for including mutual property information in each base data structures (the second base data structure does not store an instance's information as a file, so only classes store mutual property information in the second part), all commands give correct results. The example commands and results are listed in Table 10.

Table 10 Results of a query involving a mutual property

```
SQL>select FNAME from SUPERVISOR;

FNAME
--------------------
Franklin
Jennifer
James

--------------------
Record number is 3
```

In the next section, we compare the two database systems with the results that we derived in chapter 4.

7.5 Comparing two Database Systems

After implementing the instance-based database systems according to the two base data structures, two systems were built. We added more data to the example in Appendix 1 to facilitate our comparison (each database system now stores 217 instances for the comparison that follow, the attached disk include all data). The commands are compared using a PC system running Windows 98. We used a separate program to record each command's running time. The results are shown in Tables 11, 12, and 13.

From Tables 11 and 12, we see that the database system which is based on the second base data structure is not faster than the one based on the first base data structure for query operations. And if query commands are related to instances or classes, the database system based on the first data structure is faster than the second one. If query operations are only related to properties or mutual properties, then the two systems require the same time for querying. The comparisons of all these queries empirically confirm the analysis in Chapter 4. However, we can see that the update operations highlighted in Table 13 are not as the same as the analysis results. By checking the implementation methods of each database, we find that this difference is not coming from the implementation methods themselves, but from the iQL language compiler. The compiler always checks whether or

not the command is correct first. When checking whether or not the instances in these commands exist, the first data structure requires less time than the second data structure.

Table 11: Test of queries (types shared with the relational model)

commands	time	
	first structure	second structure
Query		
select *;	55.8	241.46
select * from DEPARTMENT;	0.33	2.2
select * from PROJECT;	0.55	3.24
select * from EMPLOYEE;	51.25	214.98
select * where DNUMBER=5;	0.22	0.44
select * from DEPARTMENT where DNUMBER=5;	0.11	0.77
select DNAME,DNUMBER;	0.44	0.5
select FNAME,SSN;	22.63	22.85
select LNAME,SSN from EMPLOYEE;	31.31	252.1
select LNAME,SSN from EMPLOYEE where BDATE='10-NOV-27';	10.71	226.9
select DNAME,DNUMBER from DEPARTMENT where DNUMBER=5;	0.28	3.08
select LNAME,SSN where BDATE='19-JUL-58';	1.43	1.54
select DNAME,DNUMBER where DNUMBER=5;	0.04	0.05
select FNAME,DNAME;	70.8	112.82
select PNAME,DNAME;	1.43	3.63
select FNAME,DNAME where SSN=888665555;	49.65	93.26
select FNAME, DNAME from EMPLOYEE,DEPARTMENT;	77	226.4
select FNAME, DNAME from EMPLOYEE,DEPARTMENT where SSN=333445555;	54.93	201.47
select FNAME,SSN,DNAME sharing MANAGER;	9.56	9.77
select FNAME,DNAME where SSN=888665555 sharing MANAGER;	9.12	9.39
select FNAME, DNAME from EMPLOYEE,DEPARTMENT sharing MANAGER;	13.56	125.01
select FNAME, DNAME from EMPLOYEE,DEPARTMENT where SSN=333445555 sharing MANAGER;	13.79	123.59

Table 12: Test of special queries of the instance-based model

commands	time	
	first structure	second structure
Query		
query property belongto instance 8;	0.22	0.6
query property belongto class EMPLOYEE;	0.05	0.05
query class has instance 3;	0.11	0.54
query class *;	0.06	0.06
query mutualproperty information belongto instance 4;	0.05	0.55
query instance share mutualproperty MANAGER;	0.05	0.05
query instance share mutualproperty ISEMPLOYEE;	0.61	0.65
query mutualproperty *;	0.05	0.05
query mutualproperty HOURS value sharedby instance 8 withothers;	0.16	0.16

Table 13: Test of updates

commands	time	
	first structure	second structure
Update		
insert instance (SSN,333445555,FNAME, 'Alice', SEX,'F');	0.71	0.6
insert property (MINIT,'F',LNAME,'Smith') into instance 16;	0.17	0.22
insert property (MINIT) into class EMPLOYEE;	0.11	0.11
delete mutualproperty ISEMPLOYEE from class EMPLOYEE;	0.06	0.06
insert mutualproperty (ISEMPLOYEE<-) into class EMPLOYEE;	0.11	0.11
delete property MINIT, LNAME from instance 16;	0.66	0.88
delete property SSN from instance 8;	0.16	0.44
delete mutualproperty RELATIONSHIP sharedby instance 5,13;	0.16	0.22
delete mutualproperty MANAGER sharedby instance 2,11;	0.16	0.22
delete mutualproperty SALARY;	4.45	0.11
delete property MINIT from class EMPLOYEE;	0.17	0.17
delete property MINIT;	5.05	0.11
delete instance 10;	4.66	5.32

Note: Shaded ones indicate results that differ from the analysis in Chapter 4.

So even if the update operations for the database based on the second data structure are faster than the ones based on the first data structure, the results show that the database based on the first data structure is fast in these operations. If we reduce the effect of the checking, we get the results of these update operations shown in Table 14. They are the same as our analysis. We summarize the results of the comparison of two database systems in Figure 19.

When comparing two database systems, we also find that if a query operation or an update operation refers to the instances that belong to classes then the time needed is very big relative to other operations. In the next section, we discuss how to get fast query or update under these cases.

Table 14: Results of reducing the effect of checking

Commands	Time	
	First Structure	Second Structure
insert property (MINIT,'F',LNAME,'Smith') into instance 16;	0.12	0.1
delete property MINIT, LNAME from instance 16;	0.32	0.22
delete property SSN from instance 8;	0.1	0.08
delete mutualproperty RELATIONSHIP sharedby instance 5,13;	0.12	0.08
delete mutualproperty MANAGER sharedby instance 2,11;	0.12	0.08

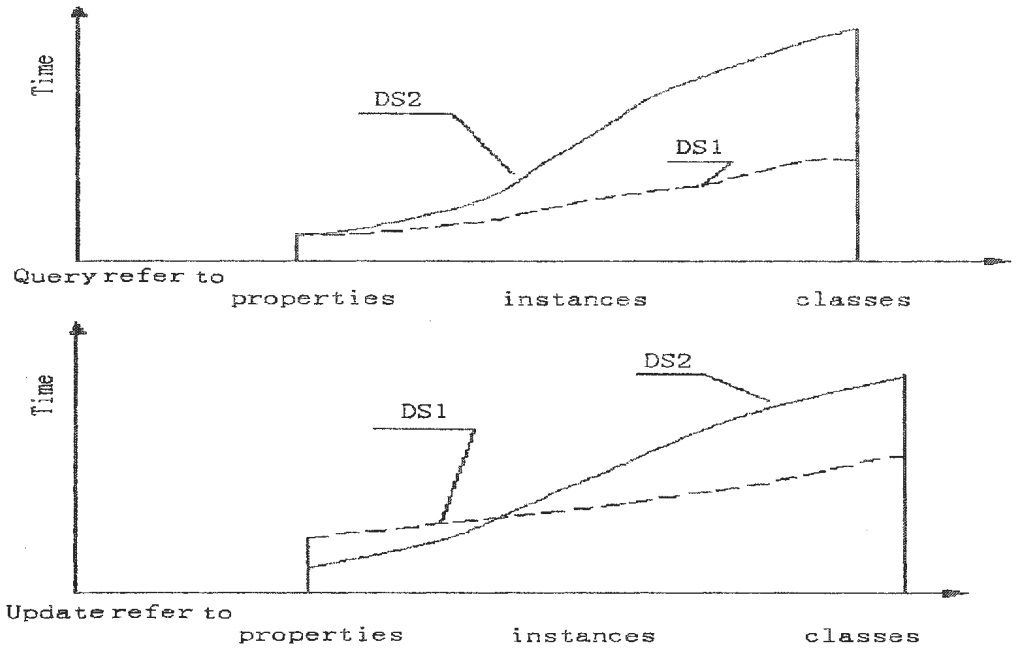


Figure 19: Comparison of two database systems

7.6 Efficient Query and Update Methods

In the relational database model, any operation, query or update, acts on tables. Each record is stored in tables, and each record has a unique key value in a table. A record has static membership in a table. That means a record is in, and only in, a certain table until it is deleted; it does not move to other tables. So a query or update operation can retrieve each table declared in each command to find data (records) that the operation needs, and then the query or update operation will be based on these data to generate results. However, in the instance-based model, the class layer is independent of the instance layer. Therefore, instance membership in classes is dynamic. An instance may belong to more than one class or may not belong to any class. And class membership may change if some

properties of an instance are updated, or some class definitions are updated. So in this model, if a query or update command refers to some instances of a class, we must decide which instance belong to the classes (In third base data structure, after updating a property of an instance or a class, we need to do this to update which instances belonging to which class. So here, we only refer our methods to the first two structures). This time is

$$C_P+(\log P_d+P_I)+(C_P-1)\times(\log P_d+\log P_I)\times P_I \qquad (10)$$

In a real database, the number of the properties, P_d, and the number of instances that possess a property, P_I, are decided by the data in the database, we cannot change them. Only the number of the properties in a class definition, C_P, is the item that we can decide. In the instance-based database system, the class definition is not basic data of the database, it is decided by users. Users (or managers) can change a class definition or update a class any time without losing any information in the database.

To optimize C_P we must realize that the goal of defining a class in the instance-based model is different from class-based models. In the class-based models, classes store all information of instances in a database. So when we define a class, it must include all properties that will store instance information in it. In fact, in class models, all information is stored in classes. We cannot reduce any properties of a class if we want to retain the fact that some instances possess these properties (and the value of the properties). However, in the instance-based model, we repeat that the concept of classes reflects how people organize knowledge about things in the world [Parsons and Wand

1997]. What concept do people refer to when they talking about a class. For example, if people refer to a class 'car', do we think about the names or the colors of cars? Generally, it is not. Rather, it is the concept of a motorized vehicle that is the definition of the class 'car'. In this definition, we find no special property (name, maker, or color) of a car in the definition, only the properties that what is this class 'car' different from other things are included. The instance-based model reflects this recognition. In this model, the instances layer stores all information of instances. Classes do not store any information about instances. A class is defined only because people want to classify instances to a set so that we can operate on this set to get fast query or update in the database. Therefore, if we define a class, we only include into the class definition the special properties that are possessed by the instances of the class and are not totally possessed by other instances. We call these special properties a minimum set of properties possessed by the instances of the class. This concept of a minimum set of properties is the same as the concept of preceding properties [Parsons and Wand 2002]. According to this recognition, when defining a class, we do not add the common properties possessed by both the set of the instances that we want to define and some other instances. For example, a class Employee may only include a property employed-by-company in its definition. It may not include the property Name in it, because other persons must possess the property. In other words, a name does not belong to the minimum set of the common properties of the set of the instances. This recognition is our first method to support fast query or update in the

instance-based model. If we use this method to define the classes in Appendix 1, then the cost of the query or update that refer to some instances of some classes reduces to 50-70% of before. Table 15 shows some example data of the comparison.

Table 15: Results of high efficiency test

Commands	Time	
	Before using the methods	After using the methods
select * from EMPLOYEE;	51.25	12.4
select LNAME,SSN from EMPLOYEE;	31.31	11.5
select FNAME, NAME from EMPLOYEE,DEPARTMENT;	77	22.4
select FNAME, NAME from EMPLOYEE,DEPARTMENT shared MANAGER;	13.56	6.1

The second method of the optimization is that, if the minimum set of the common properties of the instances that we want to classify is still big after using first method, we can create a new property to represent the set of properties both of the class definition and each instances of the class. So the class definition only includes one property and we can access it fast. We name the type of the new property as a *compound property* (in fact, it is a preceding property [Parsons and Wand 2002]), and to retain the integrity of the database we must add an engine (compound property engine) to manage the compound properties before we add any compound property to a database. The compound property engine will be able to update and manage the compound properties. That means this engine has two functions. First, it manages each compound property definition, updates these definitions, and deletes or inserts compound properties. Second, it adds a compound property to an

instance automatically if the instance possesses a subset of properties that are the same as the definition of the compound property, and it deletes the compound property if the instance loses some properties in the definition of the compound property. Of course, the engine will take some time to check properties when updating an instance or a property. However, the number of compound properties and other properties is much smaller than the number of instances that possess a property, so the cost of this engine operation is small.

In the instance-based model, the properties an instance possesses may change over time. So in order to get a fast query or update, users should create a plan to create a good class set for database operations, which means managing the dynamic classes suited for different queries. This is also one of the differences between this model and the class-based models.

In this chapter, we completed the implementation and comparison of two database systems. We also discussed how to store mutual property information in both the instance layer and the class layer. Finally, we proposed two methods for speeding up class related operations. We expect that users will frequently use these methods to manage the dynamic schema. In the next chapter, we compare the differences between the instance-based model and the relational model.

Chapter 8

Characteristics of the Instance-Based Database Model

In the previous chapters, we have discussed some different features of the instance-based model in comparison to the class-based models. In this chapter, we will summary the characteristics of the instance-based model and point out the differences of the instance-based model as compared with the relational model. We compare the characteristics of the instance-based model to those of the relational model because the relational model is dominant in database research and in practice. The characteristics cover two aspects: the design method and the management method. The focus will be on the differences between the two models.

8.1 Characteristics in the Database Design Process

8.1.1 Characteristics in Requirements Collection and Analysis

Using the relational data model, a database designer needs to collect a large amount of data requirements before beginning to design a database. This includes understanding which types of data will be used for the database and which types of relationships will be added to the database. This means a lot of time is required to design a big relational database. Many books or papers that relate to implementing relational databases give suggestions for selecting relations (e.g., [Elmasri and Navathe 2000] [Silberschatz Korth

and Sudarshan 1996]). However, since a database will be designed to operate on some special type of data (that is the data suited for a special schema of the database) and a designer may not have the special knowledge of the data, it is difficult to abstract the schema of the data even for an experienced database designer [Prietula and March 1991; Batini Ceri and Navathe 1992; Storey and Chiang 1990]. And the data collection and analysis process are also quite time-consuming.

In the instance-based model, however, the schema in the database is dynamic, and the relations among the instances are expressed by mutual properties, so we do not need to perform as much work in collecting and analyzing the data. From chapter 5 we know that when we design a database system for the instance-based model, we only need to know two things. The first is whether the database system is big. The second is whether this system will perform updating operations more than querying operations. We do not need to consider which type of data will be in the database. So the instance-based model reduces many complexities associated with the relational model when preparing database system design.

8.1.2 Characteristics in the Conceptual Schema Design

In the relational data model, the first step in implementing a database is to design the database schema. The schema is specified during the database design process and is not expected to change frequently [Elmasri and Navathe 2000]. In fact, a schema is a set of

tables in the relational database model. However, designing schemas of a database system is not an easy task.

There are two aspects that make this process difficult. The first is related to how to classify data to some entities: what are the attributes of each entity and what type of relationships exists between two types of entities. These are sometimes very confusing in database design [Wand, Storey and Weber 2000]. Another aspect is how to combine schemas. The relational model only maintains a name (either a table name or an attribute name) unique in one schema, while a database system may have more than one schema. So there may be many conflicts, such as name conflicts, type conflicts, and domain conflicts in the global database [Elmasri and Navathe 2000] and view integration problems [Parsons and Wand 2000]. For example, the concept of a DEPARTMENT may be an entity type in one schema and an attribute in another.

Therefore, in order to merge data, we need to design a global schema for a database system. However, mapping real data to a global schema is very difficult, because the relations of the data in a large database are very complex. On the other hand, sometimes we do not know whether a type of data (relation) needs to be added to the database if we do not need it currently, but may need it in the future. These factors make schema design both costly (in time) and complex.

In the instance-based model, the instance engine and the class engine manage the two levels respectively. There is only a dynamic schema in a database. The information about

which instances possess a property is stored by only this property, and the relationships between instances are stored by mutual properties. Both instances and properties are unique in the instance level and classes are unique in the class level. So when merging data, the instance-based model will reduce many problems associated with the relational model.

In addition, since in the instance-based model the relation between instances is represented using mutual properties, we only need to add mutual properties to express relations between instances for forming all relations in a database. There is no abstraction needed for expressing the relation between instances. Therefore, we can describe very complex relations between instances with a very simple method: if a relation between instances exists, we add this relation via a mutual property.

8.1.3 Characteristics in the Data Model Mapping

In the relational model, the conceptual schema can be expressed as an ER or EER diagram. This schema needs to be implemented at a logical level, that is data mapping or logical database design. There are many normalization rules for this mapping [Codd 1972] [Fagin 1977; 1979] [Bernstein and Goodman 1980], and also some new rules are still being developed [Date and Fagin 1992] [Levene 1995]. To implement these rules, we need to implement some complex methods. And there is no method to choose these rules. This can cause confusion. For example, when designing a database, a designer may not

use the 'best' rule, but one which is enough for the database. So splitting a conceptual schema into a logical schema needs in-depth understanding of the relationships of the entities. Since the relationships between the entity types are expressed as a foreign key in each table, even though the foreign key is not an attribute of an entity [Wand, Storey and Weber 2000], the meaning of the foreign key can also be very confusing. In addition, not all attributes in a table represent intrinsic properties, some represent mutual properties. However, a mutual property's existence dependencies on the existence of instances that jointly possess this mutual property. If storing a mutual property of two instances as an intrinsic property of one instance, when another instance is deleted from the database, the meaning of this property is lost.

In the instance-based model, since the information of instances is stored in the instance layer, classes (schema) are both dynamic and independent. Therefore, although the mapping still applies for fast querying and updating it does not affect queries or updates on instances. Even if no schema exists, we can retrieve from the database using 'limited' or 'unlimited' query.

8.1.4 Characteristics in the Database Implementation

In the relational model, we must implement two types of languages respectively. The first type is DDL (Data Definition Language). It includes the SDL (Storage Definition Language). We use this language to create the database schemas and empty database files.

The second type is DML (Database Manipulation Language). This is the language with which users manage the data.

In the instance-based model, the schema (classes) is dynamically managed, and it is also independent of instances. So we do not need to design an independent language to manage it. Therefore, we do not differentiate DDL and DML. The implementation will combine them into one language: IDBL (Instance-based Database Language).

In this section, we compare differences in the database design process between class-based and instance-based database models. From these comparisons, we know that the database design in the instance-based model is simpler than in the relational data model. When designing a database using the instance-based model, there is no confusion such as can arise in the relational model. And there is no complex relationship analysis such as required in the relational model. Also, in the relational model, each database is based on a special schema, so different types of data need different database schemas to manage. That is why we can see many database schemas in different application in the relational model. In the instance-based model, however, a database is not based on certain schema, so it can be used to manage any data, which means we only need design one database system for each type of requirement as discussed in Chapter 4. This property of the instance-based model will let us disentangle drastically from a complex database design process.

8.2 Characteristics in the Database Management and Application

One goal in designing a database is to make it easy for users to manage the data. Based on this point, we compare the two data models in terms of how data are managed by users. There are five differences between the models in this regard. We compare them in the following subsections.

8.2.1 Range that a Database System Manages

In the relational model, a database is based on a special schema, so a database system needs to manage static classes. If the database needs to manage extended data or reduce the range of data it manages, the classes may need to be reorganized. This will produce one of the two results: First, only some classes change. In this case, when we change classes, some information may be lost [Parsons and Wand 2000]. Second, we may need to redesign the database system, that means there are many tables or relations that need to be changed. So the old system may not suit the new data, and we need a new database. In this case, we may need to spend a lot of time and money to design the new database. The rescheduling is caused by the dependence of instances and schemas. In a class-based database, instance existence depends on a special schema. However, in the instance-based model, the database schema is dynamic, and the schema (classes) is independent of any instances. The schema (classes) is used to enable fast retrieval of some instances that belong to a certain class. Therefore, a database, once created, can manage any data.

8.2.2 Managing Temporal Data

The second difference between the instance-based model and the relational model is that the properties an instance possesses can be dynamically changed if necessary. In the relational model, records belong to tables. No record can possess a property (attribute) that does not belong to the table in which the record is located. In fact, in this model, a record (instance) is only a set of values in a table. Therefore, if a record is located in a table, it will be in the table until it is deleted from the database. It does not become a record belonging to another table even if changes in the real world necessitate such a transformation. For example, suppose for a database system in a university, there are two entity types: Student and Professor. Suppose that a person first studies at the university and graduates, and then he/she becomes a professor of the university. In a relational database, when he/she graduates from the university, we delete this record (instance) from the entity type Student. We add a record to the entity type Professor since he/she becomes a professor. Of course, some properties of the person are changed when he/she changes from a student to professor. However, it is the same instance, and a data model should allow this semantics to be preserved. However, there are two problems in a relational database: The first is we cannot express that one record is 'continued' by another. When a record is deleted, any information in this record is deleted from the database. The database system itself does not store any information to express the relation between these two records unless some information is stored in second record itself. Another

problem is that the records do not continually reflect the instance in the real world. Some time (after deleting the first record and before adding back the second record) the database loses the information of this instance[1]. In the instance-based model, since instances store their properties, and an instance can change the properties it possesses any time, if we store an instance in the database, it will be in the database until the user does not want to manage it (delete it), and the database will reflect its state over the time. Taking the example above, if a student becomes a professor, we only delete some properties that relate to being a student and add some properties that relate to being a professor. The instance is always in the database. So the instance-based model is more compatible with a real time system, such as a real time data statistical system, etc.

8.2.3 Merge Capability

The third difference between the two database models is that the instance-based model supplies a 'whole range' definition. That means in a database any instance, property, or class has its unique identifier and definition. In the relational model, however, any attribute name is unique only in one table, and a table name is only unique in a schema. Therefore, in a whole database system, there may exist some attributes that have same names but different meaning [Parsons and Wand 2000]. Similarly, some tables that have

1. Some of the information might still be kept if one designs the RDB to reflect subclasses. However, this special design is "rigid" in that it is fixed and can only deal with subsets of properties. Not any general addition and loss of properties.

different names may be the same [Kim et al. 1995] [Sheth and Larson 1990]. This complicates database querying. However, in the instance-based model, any identifier, either instance identifier, property identifier, or class identifier, is unique in the whole database system. So no confusion exists in retrieving from the whole database. That is why we can suggest whole range query and unlimited query in instance-based model. This makes the instance-based model suitable for distributed database systems, which are used more and more today, especially due to the rapid growth of high-speed networks.

8.2.4 Ability to Manage Instance-Specific Data

The fourth difference of two models is that the instance-based model is based on instances. That is it supplies relationships 'based on instances'. In the relational model, any record belongs to a table. The records in a table must have the same attributes. That means a table only maintains the common attributes of the set of instances in it. Any relation is built between tables (that is the *key* we referred in the relational model, such as *foreign key*, *weak key* etc.). That is, any relation is between two sets (each refers to a table) of instances (records). However, the instances in the real world are different. No set of properties that an instance (thing) possesses is the same as the set of properties another instance possesses. And the relationship between the instances also is very complex. That is why it is very difficult to design a database schema in a large database system. In fact, if we do not reduce some attributes or relations between the real data we cannot build the

schema. Therefore, in the relational model, both the attributes of instances and the relations between instances are curtailed. They are curtailed to maintain the same properties or relations if they are in same table (we do not say relational model can not express complex relation, but it is very costly to express that). In the instance-based model, instances are the base of the data. Any instance stores its information. So an instance can store its special properties. And we have a concept of mutual properties. This concept lets us express the relations between instances very easy. To express any relation, we only add a mutual property between these instances that are jointly related each other, whether they are in same class or not. Therefore, the instance-based model has better abilities than the relational model for managing data.

8.2.5 Support for Multiple Views

The fifth difference between the two models is that for different users, the instance-based model supplies more suitable retrieval than the relational model, even for the same data. This is because in the relational model, database schemas are static in a certain database system, so a database only includes the special entity types in the schema. Any retrieval must be through these entities and relations between these entities. In the instance-based model, however, the schema is dynamic. Any user can either retrieve from a database using the schema supplied by some system managers or can create some classes to form the schema to suit his/her special needs. For example, in a university, to retrieve some

people, an academic officer may need to partition the people to students and faculty. But a security officer may need to partition them by different departments. In the instance-based model, it is easy to achieve this goal. Therefore, different users can use different schemas to retrieve the same data in a database system and without difficulty in the instance-based model.

In this chapter, we have pointed the characteristics of the instance-based model with the relational model in different areas. We have found that in several areas, the instance-based model provides a better solution than the relational model. The instance-based model is easy to design and implement. It has greater flexibility. It also has stronger retrieve ability. Of course, the instance-based model has disadvantages in some cases. For example, in order to do a special retrieve, the user may need to create special classes for fast retrieving. This requires that the user understands how to create the suitable classes. Otherwise, the user may only use the classes managed by system managers to implement his/her querying or the classes that the user creates may not be efficient for his/her querying. So this may not provide the best speed to respond to the special requirements. In this case, the flexibility of the instance-based model may not be utilized very well so that the query may become very costly. However, we believe that with more research and development of the instance-based model, more and more advantages of this model will be found and various techniques will be developed for this model.

Chapter 9

Conclusion and Extensions

In this thesis, we have focused on transforming the instance-based database model from concepts to a real world implementation. We first proposed three base data structures for the instance-based database model. Then, we analyzed the different operations on each base data structure to suggest which base data structure is best for a certain database system. In addition, we discussed how to implement an instance-based database system and implemented two instance-based database systems, one based on the first base data structure, and another based on the second base data structure.

The advantages of the instance-based data model come from ""emancipating" the instances from the "tyranny" of the classes' [Parsons & Wand 2000]. The implementation of this feature is that any instance or property in the instance layer has a unique identifier. This lets the instance-based database system support two different queries: whole range query and unlimited query. These two types of queries support greater flexibility for querying a database. Also, these query models are more powerful. Another advantage is the fact that an instance has a unique identifier and that the database can capture temporal issues related to an instance[1]. This lets us continually manage an instance in the instance-based database model.

1. In the instance-based model, the state of an instance can be changed with the time (acquire or lose some properties). We can design some methods to store this information.

From the implementation, we also demonstrated that, if we use a suitable method (e.g., preceding property) to define classes, we can get highly a efficient query or update. Even if this efficiency is not greater than the relational model, this is very encouraging for supporting the instance-based database model, since we can get powerful query or update but at no more cost than the relational model.

There are some possible extensions of the implementation of the instance-based database. First, a database system is frequently used with multiple users. The instance-based database model is also suitable to this case, so how to manage multiple users is an important issue for future research. Second, the existence of an instance-based database is based on the existence of instances. And the existence of an instance is expressed by the fact that it possesses some properties. Further research is needed to examine how to express the semantics of properties clearly, and how to determine what types of properties to keep in a database in order to support efficient query and update. Third, the instance-based model is a new data model. Its concepts are very different from class-based models. So for developing this model to be scalable to real world applications, research is needed to optimize our implementation. For example, further research is needed to determine which data structure is best suited to store real world data and how to optimization the complex query processing.

References

1. [Albano, Ghelli, and Orsini 1995] Albano, A., Ghelli, G, and Orsini, R. 1995. Fibonacci: A programming language for object databases. VLDB Journal 4, 403--444.
2. [Batini Ceri and Navathe 1992] C. Batini, S. Ceri, and S. B. Navathe 1992. Conceptual Database Design: An Entity-Relationship Approach. Benjamin /Cummings ISBN 0-8053-0244-1
3. [Bayer and Unterauer 1977] R. Bayer and K. Unterauer 1977. Prefix B-Trees. ACM Transactions on Database Systems. 2, 1. 11-26.
4. [Bernstein and Goodman 1980] P. A. Bernstein, and N. Goodman, The power of inequality semijoins. Information System. 6, 4 (1981), 255-265.
5. [Bunge 1977] M. Bunge 1977. Treatise on Basic Philosophy: Vol 3: Ontology I: The Future of the World. D. Reidel Publishing Co., Inc., New York, NY.
6. [Bunge 1979] M. Bunge 1979. Treatise on Basic Philosophy: Vol 4: Ontology II: A World of Systems. D. Reidel Publishing Co., Inc., New York, NY.
7. [Codd 1970] E. F. Codd 1970. A Relational Model of Data for Large Shared Data Banks. CACM 13, 6. 377-387.
8. [Codd 1972] E. F. Codd 1971. Further Normalization of the Data Base Relational Model. IBM Research Report, San Jose, California RJ909.
9. [Comer 1979] D. Comer 1979. The ubiquitous B-tree, ACM Computing Surveys 11. 121-137.
10. [Cormen 1989] H. Cormen, E. Leiserson, L. Rivest 1989. Introduction to Algorithms. The MIT Press and McGraw-Hill Book Company, ISBN 0-262-03141-8, 0-07-013143-0.
11. [Date and Hopewell 1970] J. Date and P. Hopewell 1970. File Definition and Logical Data Independence. SIGFIDET Workshop. 117-138
12. [Date and Fagin 1992] J. Date and R. Fagin 1992. Simple Conditions for Guaranteeing Higher Normal Forms in Relational Databases. ACM Transactions on Database Systems. 17, 3. 465-476.
13. [Elmasri and Navathe 2000] Ramez Elmasri and Shamkant B. Navathe 2000 Fundamentals of Database Systems. Addison Wesley Longman, Inc., ISBN 0-8053-1755-4.
14. [Fagin 1977] R. Fagin 1977. Multivalued dependencies and a new normal form for relational databases. ACM Transactions on Database Systems. 2, 3. 262-278.
15. [Fagin 1979] R. Fagin 1979. Normal forms and relational database operators. In Proceedings of the ACM SIGMOD Conference on Management of Data. p153-160,

Boston, MA, 1979.
16. [Johanneson 1994] P. Johannesson 1994. Linguistic Instruments and Qualitative Reasoning for Schema Integration. Third International Conference on Information and Knowledge Management, Ed. N. Adam. Gaithersburg, Maryland, IEEE Press, 1994.
17. [Levene 1995] M. Levene 1995. A lattice view of functional dependencies in incomplete relations. Acta Cybern. 12, 181-207.
18. [Kanellakis 1996] P. Kanellakis, S. Ramaswamy, D. Vengro , and J. Vitter 1996. Indexing for Data Models with Constraints and Classes. Journal of Computer and System Sciences. 52, 3. 589-612.
19. [Kim et al. 1995] W. Kim, I. Choi, S. Gala, M. Scheevel 1995. On Resolving Schematic Heterogeneity in Multidatabase Systems. Modern database systems, Addison-Wesley Publishing Company, New York, NY, 521-550.
20. [Knuth 1973] D. E. Knuth 1973. The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley, ISBN 0-201-03803-X.
21. [Maier 1986] D. Maier, J. Stein, A. Otis, and A. Purdy 1986. Development of an Object-Oriented DBMS. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. 472-482
22. [Markowitz and Makowsky 1990] V. M. Markowitz and J. A. Makowsky 1990. Identifying Extended Entity-Relationship Object Structures in Relational Schemas. IEEE Transactions on Software Engineering. 16, 8. p.777-790.
23. [Nievergelt 1984] J. Nievergelt and H. Hinterberger 1984. The grid file: an adaptable, symmetric multikey file structure. ACM Transaction on Database Systems. 9, 1, 38--71.
24. [Parsons and Wand 1997] J. Parsons and Y. Wand 1997. Choosing Classes in Conceptual Modeling. Communications of the ACM. 40, 6. 63-69.
25. [Parsons & Wand 2000] J. Parsons and Y. Wand 2000. Emancipating Instances from the Tyranny of Classes in Information Modeling. ACM Transactions on Database Systems. 25, 2. 228-268.
26. [Parsons and Wand 2002] J. Parsons and Y. Wand 2002. Property-Based Semantic Reconciliation of Heterogeneous Information Sources. 21st International Conference on Conceptual Modeling, Tampere, Finland, 2002, Proceedings. 351-364.
27. [Prietula and March 1991] M. J. Prietula and S. T. March 1991. Form and Substance in Physical Database Design: An Empirical Study. Information Systems Research. 2, 4. p. 287-314.
28. [Sheth and Larson 1990] A. P. Sheth, J. A. Larson 1990. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. ACM Computing Surveys 22, 3. 183-236.
29. [Silberschatz Korth and Sudarshan 1996] A. Silberschatz, H. F. Korth, and S. Sudarshan 1996. Database System Concepts. McGraw-Hill, ISBN 0-07-044756-X.

30. [Storey and Chiang 1990] V. C. Storey, R. H. L. Chiang, D. Dey, R. C. Goldstein, S. Sudaresan 1990. Database design with common sense business reasoning and learning. ACM Transactions on Database Systems. 22, 4. 471 - 512.
31. [Wand Storey and Weber 1999] Y. Wand, V.C. Storey and Weber 1999. R., An Ontological Analysis of the Relationship Conceptual Modeling. ACM Transactions on Database Systems. 24, 4. 494-528.

Appendix 1

Date Stored By the Sample Database

Instances: Instances are expressed as an instance identifier followed by a set of pairs of a property identifier and value of the instance possesses this property

1 (NAME, 'Research', DNUMBER, 5);
2 (NAME, 'Headquarters', DNUMBER, 1);
3 (NAME, 'Administration', DNUMBER, 4);
4 (FNAME, 'John', MINIT, 'B', LNAME, 'Smith', SSN, 123456789, BDATE, '09-JUN-55', SEX, 'M');
5 (FNAME, 'Franklin', MINIT, 'T', LNAME, 'Wong', SSN, 333445555, BDATE, '08-DEC-45', SEX, 'M');
6 (FNAME, 'Alicia', MINIT, 'J', LNAME, 'Zelaya', SSN, 999887777, BDATE, '19-JUL-58', SEX, 'F');
7 (FNAME, 'Jennifer', MINIT, 'S', LNAME, 'Wallace', SSN, 987654321, BDATE, '20-JUN-31', SEX, 'F');
8 (FNAME, 'Ramesh', MINIT, 'K', LNAME, 'Narayan', SSN, 666884444, BDATE, '15-SEP-52', SEX, 'M');
9 (FNAME, 'Joyce', MINIT, 'A', LNAME, 'English', SSN, 453453453, BDATE, '31-JUL-62', SEX, 'F');
10 (FNAME, 'Ahmad', MINIT, 'V', LNAME, 'Jabbar', SSN, 987987987, BDATE, '29-MAR-59', SEX, 'M');
11 (FNAME, 'James', MINIT, 'E', LNAME, 'Borg', SSN, 888665555, BDATE, '10-NOV-27', SEX, 'M');
12 (FNAME, 'Alice', SEX,'F', BDATE,'05-APR-76');
13 (FNAME, 'Theodore', SEX,'M', BDATE,'25-OCT-73');
14 (FNAME, 'Joy', SEX,'F', BDATE,'03-MAY-48');
15 (FNAME, 'Abner', SEX,'M', BDATE,'29-FEB-32');
16 (FNAME, 'Michael', SEX,'M', BDATE,'01-JAN-78');
17 (FNAME, 'Alice', SEX,'F', BDATE,'31-DEC-78');
18 (FNAME, 'Elizabeth', SEX,'F', BDATE,'05-MAY-57');
19 (NAME, 'ProductX', PNUMBER, 1, LOCATION, 'Bellaire');
20 (NAME, 'ProductY', PNUMBER, 2, LOCATION, 'Sugarland');
21 (NAME, 'ProductZ', PNUMBER, 3, LOCATION, 'Houston');
22 (NAME, 'Computerization', PNUMBER, 10, LOCATION, 'Stafford');
23 (NAME, 'Reorganization', PNUMBER, 20, LOCATION, 'Houston');
24 (NAME, 'Newbenefits', PNUMBER, 30, LOCATION, 'Stafford');

Relations: A relation between two instances is expressed as a mutual property that these two instances jointly possess this mutual property and value if necessary.

Mutual property ISEMPLOYEE shared by instance 4,1 value '09-JAN-85';
Mutual property ISEMPLOYEE shared by instance 5,1 value '08-DEC-75';
Mutual property ISEMPLOYEE shared by instance 6,3 value '19-JUL-88';
Mutual property ISEMPLOYEE shared by instance 7,3 value '19-JUL-88';
Mutual property ISEMPLOYEE shared by instance 8,1 value '15-SEP-82';
Mutual property ISEMPLOYEE shared by instance 9,1 value '31-JUL-92';
Mutual property ISEMPLOYEE shared by instance 10,3 value '29-MAR-89';
Mutual property ISEMPLOYEE shared by instance 11,2 value '10-NOV-57';
Mutual property RELATIONSHIP shared by instance 5,12 value 'DAUGHTER';
Mutual property RELATIONSHIP shared by instance 5,13 value 'SON';
Mutual property RELATIONSHIP shared by instance 5,14 value 'SPOUSE';
Mutual property RELATIONSHIP shared by instance 7,15 value 'SPOUSE';
Mutual property RELATIONSHIP shared by instance 4,16 value 'SON';
Mutual property RELATIONSHIP shared by instance 4,17 value 'DAUGHTER';
Mutual property RELATIONSHIP shared by instance 4,18 value 'SPOUSE';
Mutual property CONTROLBY shared by instance 19,1 value '1';
Mutual property CONTROLBY shared by instance 20,1 value '2';
Mutual property CONTROLBY shared by instance 21,1 value '2';
Mutual property CONTROLBY shared by instance 22,3 value '2';
Mutual property CONTROLBY shared by instance 23,2 value '1';
Mutual property CONTROLBY shared by instance 24,3 value '1';
Mutual property WORKS_ON shared by instance 4,19;
Mutual property WORKS_ON shared by instance 4,20;
Mutual property WORKS_ON shared by instance 8,21;
Mutual property WORKS_ON shared by instance 9,19;
Mutual property WORKS_ON shared by instance 9,20;
Mutual property WORKS_ON shared by instance 5,20;
Mutual property WORKS_ON shared by instance 5,21;
Mutual property WORKS_ON shared by instance 5,22;
Mutual property WORKS_ON shared by instance 5,23;
Mutual property WORKS_ON shared by instance 6,24;
Mutual property WORKS_ON shared by instance 6,22;
Mutual property WORKS_ON shared by instance 10,22;
Mutual property WORKS_ON shared by instance 10,24;

Mutual property WORKS_ON shared by instance 7,24;
Mutual property WORKS_ON shared by instance 7,23;
Mutual property HOURS shared by instance 4,19 value 32.5;
Mutual property HOURS shared by instance 4,20 value 7.5;
Mutual property HOURS shared by instance 8,21 value 40.0;
Mutual property HOURS shared by instance 9,19 value 20.0;
Mutual property HOURS shared by instance 9,20 value 20.0;
Mutual property HOURS shared by instance 5,20 value 10.0;
Mutual property HOURS shared by instance 5,21 value 10.0;
Mutual property HOURS shared by instance 5,22 value 10.0;
Mutual property HOURS shared by instance 5,23 value 10.0;
Mutual property HOURS shared by instance 6,24 value 30.0;
Mutual property HOURS shared by instance 6,22 value 10.0;
Mutual property HOURS shared by instance 10,22 value 35.0;
Mutual property HOURS shared by instance 10,24 value 5.0;
Mutual property HOURS shared by instance 7,24 value 20.0;
Mutual property HOURS shared by instance 7,23 value 15.0;
Mutual property MANAGER shared by instance 1,5;
Mutual property MANAGER shared by instance 3,7;
Mutual property MANAGER shared by instance 2,11;
Mutual property HASSUPERVISOR shared by instance 4,5 value '01-JAN-89';
Mutual property HASSUPERVISOR shared by instance 5,11 value '01-JAN-85';
Mutual property HASSUPERVISOR shared by instance 6,7 value '01-JAN-85';
Mutual property HASSUPERVISOR shared by instance 7,11 value '01-JAN-86';
Mutual property HASSUPERVISOR shared by instance 8,5 value '01-JAN-88';
Mutual property HASSUPERVISOR shared by instance 9,5 value '01-JAN-82';
Mutual property HASSUPERVISOR shared by instance 10,7 value '01-JAN-85';
Mutual property SALARY shared by instance 4,1 value 30000;
Mutual property SALARY shared by instance 5,1 value 40000;
Mutual property SALARY shared by instance 6,3 value 25000;
Mutual property SALARY shared by instance 7,3 value 43000;
Mutual property SALARY shared by instance 8,1 value 38000;
Mutual property SALARY shared by instance 9,1 value 25000;
Mutual property SALARY shared by instance 10,3 value 25000;
Mutual property SALARY shared by instance 11,2 value 55000;

Classes:
class DEPARTMENT (NAME, DNUMBER);
class EMPLOYEE (FNAME, MINIT, LNAME, SSN, BDATE);
class PROJECT (NAME,PNUMBER,LOCATION);
class SUPERVISOR (HASSUPERVISOR<-);

Appendix 2

Some Results of The Test

1. Queries

```
SQL>select * where DNUMBER=5;

1(NAME, Research        , DNUMBER, 5)

-----------------------------
Record number is 1
```

```
SQL>select * from DEPARTMENT;

1(NAME, Research        , DNUMBER, 5)
2(NAME, Headquarters    , DNUMBER, 1)
3(NAME, Administration  , DNUMBER, 4)

-----------------------------
Record number is 3
```

```
SQL>select * from DEPARTMENT where DNUMBER=1;

2(NAME, Headquarters    , DNUMBER, 1)

-----------------------------
Record number is 1
```

```
SQL>select NAME,DNUMBER;

NAME                          DNUMBER
-------------------------------------
Research                 5
Headquarters             1
Administration           4

-------------------------------------
Record number is 3
```

```
SQL>select LNAME,SSN where BDATE='19-JUL-58';

LNAME                          SSN
------------------------------------
Zelaya                    999887777

------------------------------------
Record number is 1
```

```
SQL>select LNAME,SSN from EMPLOYEE;

LNAME                               SSN
-----------------------------------------
Smith                         123456789
Wong                          333445555
Zelaya                        999887777
Wallace                       987654321
Narayan                       666884444
English                       453453453
Jabbar                        987987987
Borg                          888665555

-----------------------------------------
Record number is 8
```

```
SQL>select NAME,DNUMBER from DEPARTMENT where DNUMBER=5;

NAME                          DNUMBER
--------------------------------------
Research                 5

--------------------------------------
Record number is 1
```

```
SQL>select FNAME,DNUMBER where SSN=888665555;

FNAME                              DNUMBER
-------------------------------------------
James                    1

-------------------------------------------
Record number is 1
```

```
SQL>select FNAME,DNUMBER;

FNAME                                    DNUMBER
-----------------------------------------------
John                                          5
Franklin                                      5
Alicia                                        4
Jennifer                                      4
Ramesh                                        5
Joyce                                         5
Ahmad                                         4
James                                         1

-----------------------------------------------
Record number is 8
```

```
SQL>select FNAME, DNUMBER from EMPLOYEE,DEPARTMENT where SSN=333445555;

FNAME                DNUMBER
-----------------------------------
Franklin                   5

-----------------------------------
Record number is 1
```

```
SQL>select FNAME,NAME where SSN=888665555 shared MANAGER;

FNAME                          NAME
-------------------------------------
James                    Headquarters

-------------------------------------
Record number is 1
```

```
SQL>select FNAME, NAME from EMPLOYEE,DEPARTMENT where SSN=333445555 shared MANAG
ER;

FNAME                NAME
-----------------------------------
Franklin         Research

-----------------------------------
Record number is 1
```

```
SQL>select FNAME, NAME from EMPLOYEE,DEPARTMENT shared MANAGER;

FNAME                    NAME
-------------------------------------
Franklin          Research
Jennifer          Administration
James             Headquarters

-------------------------------------
Record number is 3
```

```
SQL>query property belongto instance 3;

3(NAME, Administration , DNUMBER, 4)

-----------------------------
Record number is 1
```

```
SQL>query property belongto class EMPLOYEE;

EMPLOYEE ( FNAME, LNAME, SSN, BDATE, ADDRESS, EMPLOYEDBY(-, SALARY(- )
```

```
SQL>query class *;

DEPARTMENT ( NAME, DNUMBER, EMPLOYEDBY->, SALARY->, CONTROLBY-> )
EMPLOYEE ( FNAME, LNAME, SSN, BDATE, ADDRESS, EMPLOYEDBY(-, SALARY(- )
PROJECT ( NAME, PNUMBER, PLOCATION, CONTROLBY(-, WORKS_ON-> )
SUPERVISOR ( FNAME, SSN, ADDRESS, HASSUPERVISOR-> )
```

```
SQL>query class has instance 3;

instance 3 belongs to: DEPARTMENT;
```

```
SQL>query instance share mutualproperty MANAGER;

The mutualproperty MANAGER shared by instances :
1 and 5;
2 and 11;
3 and 7;

-------------------------------
There are 3 pair instances share the mutual property MANAGER.
```

2. Update

```
SQL>insert property (MINIT,'F',LNAME,'Smith') into instance 16;

Properties have been inserted to the instance 16.
```

```
SQL>delete property MINIT, LNAME from instance 16;

Property MINIT has been deleted from the instance 16.
Property LNAME has been deleted from the instance 16.
```

```
SQL>insert property (MINIT) into class EMPLOYEE;

Properties have been inserted to the class EMPLOYEE.
```

```
SQL>delete property MINIT from class EMPLOYEE;

Property MINIT has been deleted from the class EMPLOYEE.
```

```
SQL>delete mutualproperty SALARY;

The mutual property has been deleted from the database.
```

```
SQL>delete property MINIT;

Property MINIT has been deleted from the databse.
```

```
SQL>delete instance 10;

there are some mutual properties in this instance, you need delete mutual proper
ty first.
Do you want delete all mutual properties of the instance 10?
 if yes, input y, else input n exit.
y
The insance 10 has been deleted.
```

```
SQL>delete class EMPLOYEE;

Class EMPLOYEE has been deleted.
```