EXTENDING THE INSTANCE-BASED DATA MODEL: SEMANTICS, PERFORMANCE AND SECURITY CONSIDERATIONS

JIANMIN SU

001311

Extending the instance-based data model: Semantics, performance and security considerations

by

Jianmin Su

A thesis submitted to the School of Graduate Studies

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science
Memorial University

November 2010

St. John's Newfoundland

Extending the instance-based data model: Semantics, performance and security considerations

Jianmin Su

Memorial University of Newfoundland, School of Graduate Studies, 2010

Abstract

Current databases are typically designed for particular predetermined purposes. However, users may need to use the same dataset for multiple and changing purposes, some of which may not be known when the database is designed. To handle multiple purposes in traditional data models, it is often necessary to construct multiple databases or views. When new information needs arise, additional databases or views may need to be constructed.

The instance-based data model (IBDM) supports instances independent of any classes to which those instances might be assigned. The model adopts a two-layer approach to data organization (instance layer and class layer), so that an instance may belong to more than one class or, alternatively, none of classes defined in a database schema. The model makes it possible to construct multiple and flexible schemas for a dataset to support multiple and changing needs of users. However, previous research on the instance-based model does not address a number of issues related to the strengths of separating instance and class layers in the IBDM in fulfilling the needs of particular applications, including supporting database administration issues such as providing more flexible security policies.

In this thesis, we propose theoretical and practical enhancements to the instance-based model. First, we extend the semantics and implementation methods of data expressed in the instance-based model. The semantic extension of components of the instance-based model clarifies the definition of the model and the implementation of the components simplifies applications to real database systems. Second, we provide a theoretical comparison and an empirical simulation to show that the instance-based model is more efficient than the relational model on some typical queries. Third, we propose a security model to address security issues in multilevel security applications using the instance-based approach. To ensure the model's security, we also provide operating methods and rules for the proposed model. Finally, we evaluate the proposed model and prove that the model is secure. By applying the instance-based model to the multilevel security area, the research forms the foundation for using the instance-based model to construct multiple schemas and to support multiple applications.

Acknowledgment

First and foremost, I would like to thank my supervisor, Dr. Jeffrey Parsons, for his patience, guidance and friendship during the years that we worked closely together. Without his help, the completion of the work of this scope would not have been possible. I would also like to thank the members of my supervisory committee, Dr. Jian Tang and Dr. Krishnamurthy Vidyasankar, for their efforts in reviewing and providing valuable comments on this thesis. I would also like to thank all members of the Parsons Research Group for their generous help and friendships. During my years in graduate study, I was also honored to work with and to learn from a number of individuals. Among those, Dr. George Miminis, Dr. Siwei Lu, Dr. Miklos Bartha, Dr. Harold Wareham, Dr. John Shieh and Dr. Manrique Mata-Montero deserve special thanks. I would also like to take this opportunity to thank Ms. Elaine Boone for her continuous support throughout my program, and the head of the Computer Science Department at Memorial for academic advice and consultation. I would like to thank the entire Computer Science Department at Memorial University, all the faculty and staff members who made my life so comfortable during my years in graduate study. I would also like to thank my circle of friends, too numerous to list here, who have supported me throughout my graduate study. Finally, I would like to thank my family, my parents, my lovely wife Lin and my son Ning, for being so loving and supportive. Without them, I would not be the person I am today, and for that, I thank them.

TABLE OF CONTENTS

Abstract ii
Acknowledgments iv
Table of Contents v
List of Figures viii

1 Introduction 1
1.1 Background 3
1.1.1 Databases 3
1.1.2 Data Models 4
1.1.3 Ontologies 5
1.2 Problems with Current Data Models 5
1.3 Our Contributions 7
1.4 Thesis Organization 9

2 Data Models 12
2.1 Review of Data Models 12
2.1.1 Flat File Model 12
2.1.2 Hierarchical Model 13
2.1.3 Network Model 14
2.1.4 Relational Model 15
2.1.5 Entity-Relationship (ER) Model 16
2.1.6 Object-Oriented Model 17
2.2 Problems with Current Models 18
2.2.1 Merging Problem 19
2.2.2 Multiple Application Problem 20
2.2.3 Design Problem 22
2.2.4 User Problem 23
2.2.5 Summary 23
2.2.6 Current Solutions 24
2.3 The Instance-Based Data Model (IBDM) 25
2.3.1 Ontology Principles Used in the Instance-Based Data Model 25
2.3.2 Basic Concepts of the Instance-Based Data Model 26
2.4 Advantages of IBDM 30
2.5 Summary 32

3 Ontology Principles 33
3.1 Basic Ontological Principles 33
3.2 Things 34
3.3 Properties and Attributes 37
3.3.1 Properties 37
3.3.2 Attributes 39
3.4 Relations between Instances 40
3.5 Class and Kind 42
3.5.1 Class 42
3.5.2 Kind 43
3.6 Summary 44

4 The Semantic Extension of the Instance-Based Data Model 45
4.1 Instance Identifiers 46
4.1.1 Semantics of Instance Identifiers 47
4.1.2 Creating an Instance Identifier 49
4.2 Properties and Their Relationships 51
4.3 Relationships Between Instances 56
4.4 Class Identifier 60
4.5 Representing Data Semantics to Support Queries 61
4.6 Integrity Rules 62
4.7 Summary 64

5 Query Performance in the IBDM 66
5.1 Introduction 67
5.2 Environment of the Comparison 68
5.3 Select and Project Queries 70
5.4 Join Query 75
5.4.1 Join on the Key Attribute 75
5.4.2 Join on non-Key Attribute 75
5.5 Test 76
5.5.1 Environment of the Test 78
5.5.2 Results of the Test 80
5.5.3 Summary of the Test 83
5.6 Summary 83

6 Multilevel Security Model 84
6.1 The Research Field Overview 85

6.2 Problems with the Traditional Models 95
6.2.1 Data redundancy 95
6.2.2 Null value inference problem 97
6.2.3 Sensitive key value problem 99

7 The Instance-based Multilevel Security Model (IBMSM) 102
7.1 IBMSM 102
7.2 Data Interpretation in IBMSM 106
7.2.1 Property P_i and its Security Level C_j 107
7.2.2 An Instance's View and its Security Levels 107
7.3 Data Access and Integrity Rules 111
7.4 Two-Layered Access Control 115
7.5 Operations 117
7.5.1 Insertion 117
7.5.2 Deletion 121
7.5.3 Select 124
7.6 Security 126
7.7 Structure and Implementation Methods of IBMSM 135
7.8 Summary 139

8 Power and Security 140
8.1 Duplicate Records Problem 140
8.2 Null Values 143
8.2 Sensitive Data in the Key Attributes 144
8.2.1 Multilevel Key Attributes Problem 144
8.2.2 Key Loophole Problem 145
8.4 Summary 149

9 Conclusions and Future Work 151
9.1 Conclusions 151
9.2 Contributions 154
9.3 Future Works 155

10 Bibliography 157

11 Appendix: Wisconsin Benchmark Queries and Results 164

List of Figures

Number		page
Figure 1	An Example of Multiple Schemas in the Instance-based Model	7
Figure 2	An Example of the Hierarchical Data Model	13
Figure 3	An Example of the Network Model	14
Figure 4	An Example of the Relational Model	15
Figure 5	An Example of the ER Model	16
Figure 6	Merging Problems	19
Figure 7	Constructing a Higher Level Schema for Multiple Databases	21
Figure 8	Two Layers in the Instance-Based Data Model	27
Figure 9	Data Storage Methods for the Instance Layer	28
Figure 10	Data Storage Methods for the Class Layer	29
Figure 11	Differences Between Views and Schemas	31
Figure 12	How People Refer to Things	36
Figure 13	Class and Kind	44
Figure 14	An Instance Identifier Represents a Globally Recognized Unique Property	48
Figure 15	Implementation System of Instance Identifiers	50
Figure 16	Transforming Joint Properties to Discrete Properties	54
Figure 17	Representing the World at the Instance Level	57
Figure 18	Data Representation at the Instance Level	58
Figure 19	Three-level Structure of a Family	59
Figure 20	Two-level Structure of a Family	59
Figure 21	An Employee Table	69
Figure 22	The Second Data Structure of IBDM	69
Figure 23	Architecture of the Instance-based Database System	77
Figure 24	Attribute Specification of Scalable Wisconsin Benchmark Relations	79
Figure 25	Non-indexed Property Queries	80
Figure 26	Non-Clustered-Index Property Queries	80
Figure 27	Update Operations	81
Figure 28	An Example of the Tuple-level Labeling Model	86
Figure 29	An Example of the Element-level Labeling Model	87
Figure 30	A Query Result for an L_2 User in the Element-level Labeling Model	88
Figure 31	An Example Data of Smith and Winslett Model	88
Figure 32	Typical Data of the SeaView Security Model	90

Figure 33	An Example Table for the Sandhu-Jajodia Model	91
Figure 34	An Example Data of the MRL Model	93
Figure 35	An Example Data of the BCMLS Model	93
Figure 36	Unambiguous Results to the Users in the BCMLS Model	94
Figure 37	A Customer Relation	96
Figure 38	A Customer Relation in BCMLS Model	97
Figure 39	The Null Value Inference Problem	98
Figure 40	Data Redundancy Problem	98
Figure 41	The Sensitive Key Value Problem	99
Figure 42	The Belief Problem	110
Figure 43	Two-layered Controls	116
Figure 44	IBMSM Database States Transformation	129
Figure 45	Different Security Levels of Users Affect Each Other	130
Figure 46	Users Affect Each Other on Different Level Data	130
Figure 47	Affecting on Different Levels	134
Figure 48	Architecture of the Instance-Based Multilevel Security System	136
Figure 49	An Example Data Structure of IBMSM	137
Figure 50	Entity Polyinstantiation and Element Polyinstantiation in MLR Model	141
Figure 51	Multiple Key Attributes Problem in MLR Model	145
Figure 52	Passenger Relation	146
Figure 53	Different Level Views on Passenger Table	147
Figure 54	Mask John's Name in Passenger Relation	147
Figure 55	Hidden Passenger John from Lower Level Users	148

Chapter 1

Introduction

Building a database to support multiple applications is an essential aspect of database systems. However, traditional database models suffer from a serious drawback. Let us take an example. In the relational model, information has to be stored in tables in which each row represents an instance of an entity type (or class) [1]. Any presentation of the data in the database is based on the meaning of these tables (classes). So, in traditional models, for each application, one has to build a separate database. For example, we have bank management systems for bank applications, healthcare systems for hospital applications, sales management systems for retail applications, and so on. When the relational model was first proposed, the requirement for a separate database for each application was not a problem since there was no network connection between different systems. One specific system for one kind of application was reasonable. However, with the development of networks, systems now can be connected to many others. Meanwhile, some applications might need to combine several data sources from different databases after these databases were built. For example, a customer in a retail system may also be a patient with an infectious disease (for example, the swine flu) in a hospital. It will be more efficient if physicians at the hospital can combine the data in the two systems to find out the whereabouts of other customers who had contacted with the patient. Not only would it be inefficient to build a system to combine data for the application but also traditional models have lots of inherited problems when combining data. For example, how to combine the customer's categorical age (e.g. child, adult, and senior) with the patient's age (in the form of an exact numerical age)? It will be more efficient if the two systems can be combined together without any additional steps. However, in traditional models, a system can be built only if we know the applications for which the system will be used. It is difficult also to use a system to support a non-defined application after the system is built. For example, in the traditional models, we cannot query information of a patient from a retail system. That is, traditional models can support an anticipated application of a database, but they cannot support any non-anticipated application of a database after the database was built.

To overcome the problems of traditional models, Parsons and Wand proposed a new model, the instance-based data model (IBDM) [2], by applying ontological theory [3] [4] to data modeling. The model adopts a two-layer approach so that an instance may belong to more than one class. By using a two-layered approach, the model makes it possible to construct multiple and flexible schemas for a dataset to support multiple or changing purposes. However, previous research only focused on the abstract model. How the semantics of components should be and how they should be implemented not been considered. In the absence of these considerations, it is difficult to apply the model to any real application. In this thesis, we address several issues of the instance-based data model and their implementation methods according to Bunge's ontology. The research provides clear understanding of the components of the instance-based data model so that they can be implemented easily. We also propose a new security model, the instance-based multilevel security model [5], based on the instance based model, to solve problems in the current multilevel security data models. The research also demonstrates that the instance-based data model can be applied to some special areas to make databases more suitable for applications in these areas.

1.1 Background

1.1.1 *Databases*

By definition, a database is a structured collection of related data [1]. Despite this general definition, currently databases usually have many restrictions. For example, a database is designed for a special purpose. It can be used only for the applications it is intended to be used for. For example, when we use a bank machine (essentially a computer or computer-based machine) to deposit/withdraw money, we are dealing with a bank database system. Or if we check our utility bill online, we are dealing with a supplier's database system. We cannot deposit/withdraw money to/from our bank account by using a supplier's database.

As a component of computer software, database technologies are important in all areas where computers are used [6]. With the increase of related computer usage, the use of database technology has increased in recent decades as well.

1.1.2 Data Models

A data model is a collection of concepts that can be used to describe the structure of a database [7]. Most commonly, the structure of a database includes types of data, relationships between data and constraints that hold on data.

Based on the constructs data models provide to users, we categorize data models in two types: conceptual data models and physical data models [8]. Conceptual data models provide higher level constructs that are closely related to how users perceive data. For example, conceptual data models express what kinds of entities are stored in a database and what kind of relationships exist between entities. Conceptual models, such as the entity-relational model [9], are widely used by either database designers or end users. On the other hand, physical data models provide lower level concepts that describe data stored in computers. They describe how the information represented in the conceptual model is actually implemented, how the information exchange requirements are implemented, and how the data entities and their relationships are maintained. The physical data model usually is used to calculate storage estimates and may include specific storage allocation details for a given database system. Since the physical data models are so closely related to computer technology, in most cases, they are only used by computer specialists. In this thesis, we largely deal with conceptual data models. Only Chapter 4 refers to physical data models.

1.1.3 Ontologies

Ontology is the study of being or existence and its basic categories and relationships [3]. Ontology attempts to describe what things exist, and how these things can be related together or can be grouped according to similarities and differences. There are several ontological theories used in the computer science and information systems field. Different ontologies may be used for different objectives. For example, domain specific ontology [10] and upper ontology [11] are mostly used in the area of information science. In this research we use a general ontology, Bunge's ontology [3] [4], because it has been used to analyze information systems modeling concepts and has produced useful results [2] [12] [13] [14] [15] [16] [17] [18] [19].

1.2 Problems with Current Data Models

Current databases are designed for particular purposes. However, users may need to use the same dataset for multiple purposes. For example, in a university, the head of a department may be interested in the academic information of a student; library staff may need to know the information about the books people borrowed; and an officer in the campus security department may need information about the names of the faculty members who are authorized to enter lecture halls during a period of time. To handle these purposes in traditional database models, it is often necessary to construct multiple databases. As the need for information increases, more databases need to be constructed.

Enabling a database to support multiple applications is an essential issue for the effective use of database technologies. For example, the relational model [23] provides "views" to support multiple purposes. However, views in the relational model and other traditional models are only subsets of the schemas of the database schema. That is, a view can only provide information that is included in the schema of a database. It cannot provide information that is beyond the schema.

The idea of a global schema has been proposed to accommodate multiple applications [20]. However, at least two problems have since emerged: First, it is nearly impossible to identify all the potential queries that a user could make beforehand. For a global purpose, before building the actual database, designers will probably have very limited knowledge of who will query the database and what their interests will be. Second, even if all the potential queries were considered in advance, a new problem arises: designing a global schema that is able to support all these queries. For example, to design a large database system (even a large database system may not be a global system), a group of designers may need to work for many months [1]. As a result, *'the universal data model may be unattainable.'* [20]

The fundamental problem with existing data models is that they are schema-based [2]. In these models, data stored in a database is organized based on the schema of the database. However, the schema of a database is designed only for a particular application and it is fixed. It can merely represent data related to the application under the schema and answer only questions pertaining to this application.

1.3 Our Contributions

To overcome the problems of the traditional models, the instance-based data model (IBDM) was proposed [2]. The instance-based data model supports instances independent of classes. The model adopts a two-layer approach so that an instance may belong to more than one class or, alternatively, it may not belong to any classes at all in a database. By using a two-layered approach, the model makes it possible to construct multiple and flexible schemas for a dataset to support multiple, even unanticipated or changing purposes. Figure 1 illustrates multiple schemas (each related to an application) built on top of one set of data. As shown in Figure 1, each schema in the Class-level may only deal with part of data in the Instance-level. However, no global schema is needed in the model. So, with the model, it is easier to build a database for multiple applications.

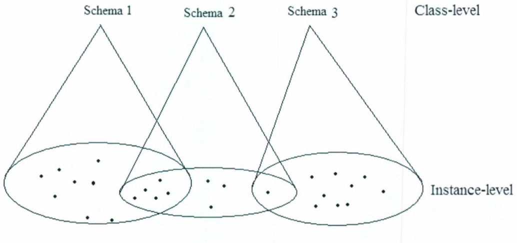


Figure 1: An Example of Multiple Schemas in the Instance-based Data Model

Previous research on the instance-based data model does not address a number of issues related to the strengths of separating instance and class layers in the IBDM in fulfilling the needs of particular applications, including supporting database administration issues such as providing more flexible security policies. In this thesis, we extend the semantics of the instance-based components and implementation methods for data expressed in the instance-based data model. We address several issues of the instance-based data model and their implementation methods according to Bunge's ontology. The main contributions of the thesis are follows:

1. Extending semantics of Bunge's ontology to the instance-based data model
    a. We clearly define the semantics of the instance identifier and propose a possible method to implement instance identifiers in the instance-based data model.
    b. We address how to express properties and their relationship in the instance-based data model in order to reduce the complexity of managing an instance-based database and increase the query capability of the model.
    c. We build a model to demonstrate how to represent the real world in different levels in the instance-based data model according to Bunge's ontology: instances related to each other form a higher level conceptual thing so that all instances combine together forming the real world in the highest level.
    d. We define several integrity rules of the instance-based data model to reduce possible inconsistencies in the model.

2. We demonstrate that, in theory and in practice, the instance-based data model is not only flexible for queries, but also faster than the relational model in processing a broad range of queries.
3. We propose a new security data model, the instance-based multilevel security model (IBMSM), based on the instance-based data model as an application of the theory. The new model solves several problems in the multilevel security control area. In the thesis:
    a. We formally define the instance-based multilevel security model to solve data polyinstantiation and data inference problems in class-based multilevel security model. This includes:
        i. Definition of data interpretation and integrity rules.
        ii. Definition of a two-layered control model for the instance-based multilevel security control.
    b. We extend operations of the traditional SQL statements and instance-based iQL[70] statements to the multilevel security model.
    c. We prove that the instance-based multilevel security model is a secure model.

1.4 Thesis Organization

In the next chapter, we first review the most commonly used data models. We point out the problems with the current data models in more detail. Next, we review the instance-based data model. The introduction of the instance-based data model includes:

the ontological principles used to build the instance-based data model, the basic concepts of the instance-based data model, and the possible structures and implementation methods in the instance-based databases. We also briefly discuss advantages of the instance-based data model.

In Chapter 3, we introduce the basic elements of Bunge's ontology: what is a thing, what are properties and attributes, what kinds of relationships exist between instances, what is a class and a kind, and the concept of systems.

In Chapter 4, we discuss some semantic extensions of the instance-based data model: what are the semantics of the instance identifier and how to implement them, what are properties and relationships between them, and how to represent the real world in the instance-based data model. In the final part of Chapter 4, we introduce integrity rules for the instance-based data model.

In Chapter 5, we compare querying in the instance-based data model with querying in the relational model. The comparison has two parts, the first part is a theoretical comparison and the second is an empirical evaluation on a test database.

Chapter 6 introduces the concepts of multilevel security control and the problems with current models. In Chapter 7 we propose the instance-based multilevel security model (IBMSM) and provide data interpretation, integrity rules, and operation methods of the model in this chapter. In this chapter, we also prove that the IBMSM is a secure data model and show that it addresses several unsolved problems under the traditional multilevel security models.

Chapter 8 compares the IBMSM model with other class-based multilevel relational models and indicates how the new model solves the problems of the current models.

Chapter 9 provides some conclusions and summaries the primary contribution of this research. It also suggests several research areas for future investigation.

Chapter 2

Data Models

A database is an organized collection of data. A data model is a collection of logical constructs used to represent the data structure and the data relationships within the database. [6]

2.1 Review of Data Models

Many data models have been proposed. Most of them are class-based models. For better understandings of the concepts of the instance-based data model, we will review some of the most popular models in the following sections.

2.1.1 Flat File Model

This is not considered as a data model by some scholars, since it merely shows tables of values. The relationships between records and between tables cannot be represented in such a model. Data are simply stored in the database. This model was mainly used in the early age of computer database, but it has no obvious advantage, compared to modern database systems. For example, users may have to search the entire database to find required results in the flat file model. Therefore, querying a large database is very slow.

2.1.2 Hierarchical Model

In the hierarchical data model [21], data are organized into a tree structure. The structure allows repeating different types of information using the parent/child (or hierarchical) relationships. However, the relationship between parent and child can only have one-to-many relationships. Figure 2 shows an example of the hierarchical model.

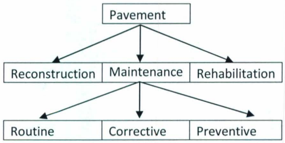


Figure 2: An Example of the Hierarchical Data Model

The hierarchical model is the first model which represents relationships between different data tables. Compared to the flat file model, hierarchically structured database systems are very fast for certain types of queries. Hierarchical structures were widely used in the first generation of mainframe database management systems.

Because of its one-to-many relationship, the hierarchical structure is simple to construct; however, the limitation of this model is also a consequence of its simplicity. Relationships in the real world are not just parent/child relationships, as many-to-many relationships are also very common. But it is costly to represent many-to-many relationships in the hierarchical data model. Thus, the hierarchical data model is often not able to adequately represent many structures that exist in the real world.

2.1.3 Network Model

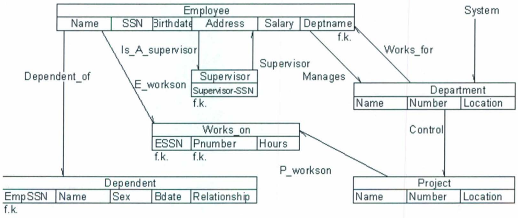


Figure 3: An Example of the Network Model [1]

The Network model was introduced in the same period as the hierarchical model by Codasyl data base task group [22]. Its structure is very similar to the hierarchical model. The only difference is that, instead of a tree of records such that each record has one parent but many children records, the network model allows records to have multiple parent and child records, forming a network structure. An example of the network model is illustrated in Figure 3. The advantage of the network model, in comparison to the hierarchical model, is that it allows a more natural way of modeling relationships between entities.

2.1.4 Relational Model

With the growth of data intensive applications and of computational capacity, a more flexible database model, the relational model [23], emerged to replace the hierarchical and network models. The fundamental assumption of the relational model is that all data can be represented as mathematical relations. The relational data model permits designers to create a consistent logical model of information and refine it through database normalization. An example of the relational model is shown in Figure 4.

Customer

Customer ID	Name	Address	City	State	Phone

Order

Order No	Customer ID	Invoice No	Date Placed	Date Promised	Status

Invoice

Invoice No	Customer ID	Order No	Date	Status

Product

Product Code	Product Description

Figure 4: An Example of the Relational Model

The foundation for the relational model is set theory. Set operations, such as union, intersection and Cartesian product, form the basis for querying data. The use of sets and set operations provides independence from physical data structures (in contrast to the hierarchical model and the network model), a pioneering concept at the time it was introduced.

The relational model is a successful commercial model; even today, most database systems still use this model. Its ultimate success also comes from continuing research efforts after the model was proposed [24].

2.1.5 Entity-Relationship (ER) Model

The entity-relationship (ER) model was proposed by Chen in 1976 [9]. The ER model is a conceptual data model (or semantic data model) that views the real world in terms of entities and relationships. A basic component of the model is the entity-relationship diagram, which visually represents data objects. The basic model has been extended [25] [26], and today it is frequently used to design databases.

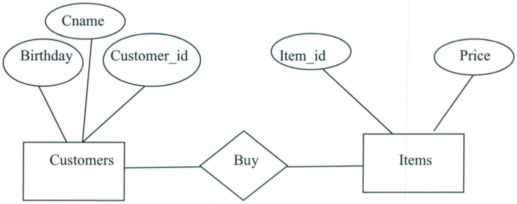


Figure 5: An Example of the ER Model

There are two primary advantages of the ER model. First, it maps well to the relational model. The constructs used in the ER model can be easily transformed into relational tables. Given this advantage, the model is mostly used as a design plan by the database developer to implement a data model using specific database management software. Second, it is easy to understand with minimum amount of training. Therefore, the model can be used by the database designer to communicate the design with end users.

2.1.6 Object-Oriented Model

The Object-oriented data model [27] integrates databases with object-oriented technologies. Several object-oriented models were proposed in the early 1990s [28]. However, these models suffered from two drawbacks, lack of standardization and lack of successful implementation approaches to ensure interoperability between products [29]. Even today, there are only a few applications using object-oriented data models, and they are usually specialized applications such as engineering databases or molecular biology databases rather than mainstream commercial data processing. However, object database ideas were picked up by the relational vendors and influenced extensions made to these products and indeed to the Structured Query Language (SQL). For example, both Oracle [30] and Microsoft [31] add extensions in their database platform to support some object-oriented features.

Based on the popular data models, Hammer and McLeod define database and database model as follows:

A database is more than just a collection of values. At every point in time, the contents of a database represent a snapshot of the state of an application system, and the changes to the database over time reflect the sequence of events occurring in the application environment. In other words, a database is a model of a real world system [32].

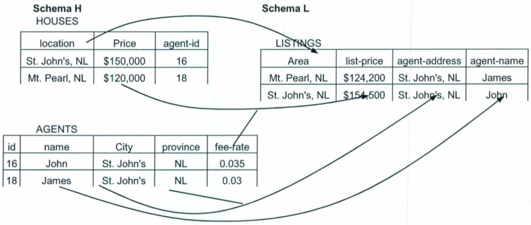
2.2 Problems with Current Models

All the above models are based on a common assumption that data instances (e.g. records, tuples, entities, objects) should always be classified into certain types (record types, relations, entity types, classes) and relationships exist between the types. Therefore, they represent each entry by a sequence of values and then assign the entry its type. The meaning of the entry is provided by the type to which it belongs. For example, the entry {Ford, 50, 80} that belongs to the type {name, age, weight}, which means a person name is Ford, his age is 50 and his weight is 80kg, will be completely different if it belongs to the type {Maker, Year, engine}, which means a car is made by Ford in 1950 and it has 80 horse power engine. We call this kind of model a 'class-based data model'. In a class-based data model, each type (record types, relation, entity types) can be expressed as a class. Class-based data models have been very successful for the past 50 years, but there are a number of problems with them.

2.2.1 Merging Problem

The merging problem is a problem of semantic integration in class-based data models. In the models, since each database is designed for special purposes, the schema of each database is different. Each database is a closed world under its schema [1] [33]. When it is necessary to query multiple databases for a new application, the databases first have to be merged. However, how to merge these databases is a crucial issue. Since models are based on classes, one has to merge the classes (schemas) first before merging the data itself.

There are lots of difficulties in merging schemas and data [34] [35] [36] [37] [38]. Figure 6 shows some of the basic problems when merging schemas of multiple databases in class-based data models. The schemas of two relational databases H and L on house-listing and the semantic correspondences between them are shown in Figure 6. Database


Schema H
HOUSES

location	Price	agent-id
St. John's, NL	$150,000	16
Mt. Pearl, NL	$120,000	18

AGENTS

id	name	City	province	fee-rate
16	John	St. John's	NL	0.035
18	James	St. John's	NL	0.03

Schema L
LISTINGS

Area	list-price	agent-address	agent-name
Mt. Pearl, NL	$124,200	St. John's, NL	James
St. John's, NL	$154,500	St. John's, NL	John

Figure 6: Merging Problems [34]

H consists of two tables: HOUSES and AGENTS; database L consists of the single table LISTINGS. Figure 6 also shows the duplicate record problem of merging data of multiple databases. The first record in schema H is the same as the second record in schema L. The list price is arrived by combining price and fee-rate.

Many challenges have been encountered in performing semantic integration [34] and researchers have proposed several approaches to do the matching, such as rule-based solutions [39] [40] [41] and learning-based solutions [42] [43] [44] [45]. Even with these methods, problems may not be eliminated. The basic idea of a rule-based solution is to design some rules for mapping one schema to another schema or a global schema. The rule-based methods do not require training so they are faster when trying to build a matching system; however, they cannot exploit data instances effectively since they need to calculate the matching (rules) at query time. Also, they cannot exploit previous matching efforts to assist in the current ones. The learning-based method is efficient after the calibration; however, the calibration process itself is much more complicated. Also, since it operates on records, it is quite slow when working on a large dataset. For example, most learning-based algorithms may take more than an hour to generate a higher level model on large datasets.

2.2.2 Multiple Application Problem

A database should support multiple classifications so that the user can choose a preferred classification as the basis of a query. A database should also be able to translate between different classifications. However, current database systems are not suited to this task [46] [47]. A database is designed for a specialized purpose in class-based data models; however, users may have several interests at the same time. For example, in a university, the head of a department may need academic information; library staff may need information about the books people borrowed; and an officer in the campus security department may need information about the list of faculty who are authorized to enter lecture halls during a period of time. To handle these purposes in traditional database models, it is often necessary to construct multiple databases. As the need for information increases, more databases (or views) need to be constructed. Building views may reduce some complexity to query databases, but views depend on base tables of databases. After base tables in a database are created, views of the database cannot provide any more information other than querying the base tables. It will be much more efficient if a master database system can be constructed to handle all the needs; however, predicting such

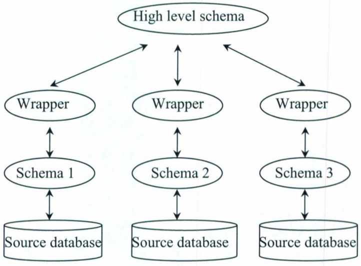


Figure 7: Constructing a Higher Level Schema for Multiple Databases

needs is challenging. Some researchers have tried to solve this kind of problem. The idea is to construct a single higher level schema to cover all the applications. Figure 7 demonstrates the basics behind this idea. However, such an approach will give rise to the merging problem from the previous discussion. The problem also will grow if no shared common schema of the multiple applications can be found.

2.2.3 Design Problem

In class-based data models, designers have to create a schema of a database in advance and it will be rarely changed after the database is built. This places a heavy burden on the designers and the process is time consuming. Designers have to discuss with the users and make sure they have understood what the users' demands may be for querying the database and what should be in the database. Therefore, it is difficult and time-consuming to build a schema for a large database [1]. Scholars have tried to simplify this step by building some standard schemas for similar applications and calling these schemas the universal data model [48]. The idea of the universal data model is to build standard schemas for people to reuse them. It may reduce some duplicated effects: "*Universal data models can substantially reduce the time to complete a corporate data model, logical data model or data warehouse design.*" [48] However, at least two problems with the universal data model emerge: First, nearly infinite numbers of potential applications exist for which people may need to query data. Using only a few schemas to cover all these applications is impossible. Second, another problem is that there is no method to evaluate whether the designed schema will be satisfactory or not before the database is built and used. So there is always the risk that such a database will not satisfy all the users' demands.

2.2.4 User Problem

In the traditional (relational) model, data are stored in tables (classes) and users have to follow the structure of the tables and their relationships (schema) to operate on the database. However, especially in large, complex databases, users may not have much knowledge about the database and/or may not be familiar with the schema of the database. Graphic interfaces could be helpful, but such interfaces also limit users' access based on the purposes of the interface [49].

2.2.5 Summary

The common problem of class-based data models is that they are specialized and application-based, in which the data stored in databases are based on its application. When a database is built, database designers only design a schema that structures the data to support the information needs for a proposed application. Since different information is required for each different application, there will be problems when users try to combine information coming from several different applications. Merging problems as we have described above could occur.

2.2.6 Current Solutions

There are two alternate solutions to the problem: one is that when one builds a database, one designs a data model that will be suited for multiple applications. However, although designers may have some idea about the proposed applications, they cannot predict if and what kind of new applications could be applied in the future. So, unless designers can build a universal data model which covers all the potential applications, this solution is not permanent to solve the multiple-application problem. However, there is no universal data model for class-based data models. Therefore, in a class-based data model, it is impossible to design such a database that covers all the potential needs of the users.

The second method is to build a new database whenever a different application is desired. Repeatedly constructing database systems will cause several problems, both technical and non-technical. For example, it will cause problems in maintenance of each of the databases in the future (technical), and it will waste time and money to build multiple databases (economical).

Since class-based data models have the problems above, researchers have proposed other models to overcome the problems by using instances as a basic unit of a database system. However, most proposed models are not efficient in class-related queries. For example, the functional data model [50] [51], the logical data model [52], the item-centric data model [53] and the attribute based data model [54] are inefficient when queries refer to classes since they do not store any information about classes.

2.3 The Instance-Based Bata Model (IBDM)

The instance-based data model is based on the ontology of Bunge [3] [4] and research on classification theory [2]. It does not rely on the concept of inherent classification, which is fundamental to class-based models such as the relational and object oriented models. This section will review some of the important concepts of the instance-based data model, based on Parsons and Wand [2].

2.3.1 Ontological Principles Used in the Instance-Based Data Model

The instance-based data model is based on research in the field of cognitive classification. It also depends on several ontology principles:

Principle 2.1: The world is made of things that possess properties.
An ontological principle connects the existence of a thing with its properties.

Principle 2.2: No two things can possess an identical set of properties.

Principle 2.3: Classes are abstractions created by humans in order to describe similarities among things.

Two conclusions can be derived from above principles [2]:

Implication 2.1: Recognizing the existence of things should precede classifying them.

Implication 2.2: 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 instance-based data model also assumes an "open world":

Postulate 2.1: Whether a property of a thing exists is not determined by human's recognition but by the thing itself.

When people refer to a property of things, there are two meanings: one is that it is possessed by things; the other is people realize and define a property (or a set of properties) of things. In Bunge's ontology [3] [4], we call this property an *attribute*. We will discuss this in more detail in Chapter 3.

2.3.2 Basic Concepts of the Instance-Based Data Model

The basic idea of the instance-based data model is to separate instances from classes. To achieve this goal, the instance-based data model suggests a two-layered approach using layers to store information separately: the Instance Layer, which stores all the information about instances, and the Class Layer, which only stores information about classes (structured as in Figure 8).

The instance layer consists of instances and their properties. Properties may be intrinsic (belonging to an instance) or mutual (shared by more than one instances, and often represented by relationships or associations in traditional data modeling terms). An instance in the Instance Layer can be represented as an instance-id followed by a set of

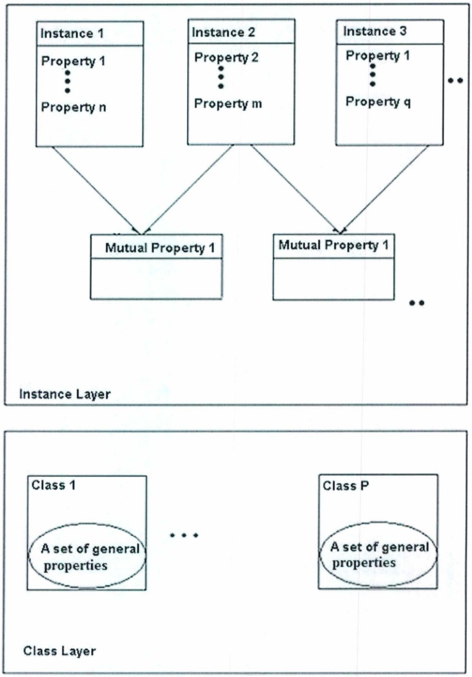


Figure 8: Two Layers in the Instance-Based Data Model

instance 1
property 1-ID
property 2-ID
property 3-ID
mutualproperty1-ID

instance 2
property 1-ID
property 3-ID
mutualproperty1-ID

instance 3
property 1-ID
property 2-ID
mutualproperty1-ID

mutual property 1		
instance1-ID	instance2-ID	value
instance3-ID	instance2-ID	value

property 1	
instance1-ID	value
instance2-ID	value
instance3-ID	value

property 2	
instance1-ID	value
instance3-ID	value

property 3	
instance1-ID	value
instance2-ID	value

Figure 9: Data Storage Methods for the Instance Layer

pairs of properties and values, i.e., Instance-id {(property1, value1), (property2, value2), …}. Figure 9 shows the structure of data in the Instance Layer. In Figure 9, there are two possible structures in the Instance Layer: one is only to use the shaded part to store the information; the other is to use both parts (shaded and unshaded) to do so. A framework for implementation and comparison of the two structures is discussed in [49]. From the implementation and comparison we know that the second data structure (using the shaded part) is suitable for small database systems, but the first data structure (both shaded and unshaded parts) is suitable for general purposes. Query operations on the Instance Layer consist of: (1) instances that exist and (2) properties of an instance (and their values). Update operations on the Instance Layer include: (1) addition and deletion of instances, (2) addition and deletion of properties of instances, and (3) update of values of properties.

The Class Layer consists of a collection of classes. A class is a set of instances that share common properties [3]. In the instance-based data model, a class is defined as a set of properties. Any instance that possesses these properties belongs to this class. The possible structures of the Class Layer are shown in Figure 10. Two possible structures exist. One is to use the shaded part only to store the information; the other is to use both shaded and unshaded parts for storage. A detailed comparison of the two structures is

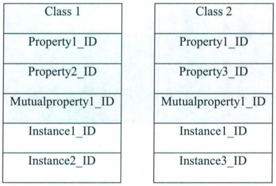


Figure 10: Data Storage Methods for the Class Layer

discussed in [49]. From that comparison we know that the first one (using the shaded part) is suitable for databases which have more update operations, but the second one (using both shaded and unshaded parts) is suitable for query intensive applications. Suggested query operations at the class level consist of: (1) classes that exist, (2) properties that make up a class, and (3) instances that belong to a class (this query associates the Class Layer with the Instance Layer). Update operations at the class level include insertion and deletion of classes (modification of class definitions can be done by a sequence of deletions and insertions).

The two-layered approach gives the instance-based data model two distinctive characteristics. First, instances can be added and stored to a database directly, not as an instance of some classes, even if they do not belong to a class, or alternatively, they can belong to many classes simultaneously. Second, changes can be made at the class level without operations on the instance level.

2.4 Advantages of the IBDM

The instance-based data model uses two layers to store classes and instances separately. Since it supports the existence of instances in a database without classes and class definitions only depend on the concepts (properties) in the database, the major advantages in this model are:

(1) Designation of a schema prior to populating a database is unnecessary in the instance-based data model. In the model, the schema can be defined according to the applications so that it enhances system efficiency in query operations. In this model, a fixed schema is unnecessary.

(2) In traditional database models, relationships are often described as links between classes (e.g., foreign keys in the relational model). Relationships are only between classes (and are part of the schema in the traditional database). Therefore, it is important to analyze carefully the relationships between classes during database design in traditional models. Therefore, it is arduous and laborious for the program designers to create all relationships beforehand. However, the instance-based data model supports relationships between individual instances. If an additional relationship is found or created between the instances, simply adding a mutual property shared by these instances will store this relationship and solve the problem.

(3) A database built using the IBDM can store properties about instances, regardless of whether there is a class to hold the data, in contrast to the traditional model in which the schema (e.g., relations) determines what properties can be kept about instances (via the attributes they possessed by virtue of their membership in classes).

(4) In the traditional database models, only one schema exists for one dataset (that is the schema of the database). Some views can be defined for query convenience. However, the views only support structure within the schema of a database; that is, a view only supports one specific purpose within the more general database purpose. As shown in Figure 11 (a), in the traditional database models, the schema of database abstracts the meaning of the database. A view can be and in most systems is a partial database schema. It cannot contain information that is not in the database schema. In contrast, in the instance-based data model a data set has dynamic schema, as shown in Figure 11 (b),

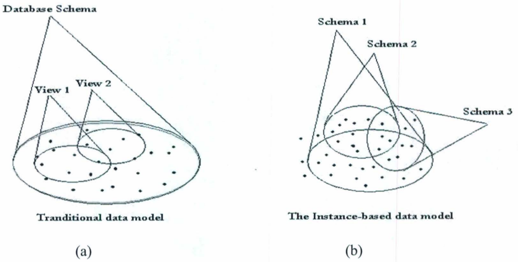


Figure 11: Differences between Views and Schemas

supporting different schema to meet different application needs. These schemas (views) are completely distinct from the views in the traditional models. In this model, there is no need to design a global master schema to include other schemas (views).

2.5 Summary

The instance-based data model provides a new approach to the field of databases. The model is guided by ontological principles recognizing that the world and instances are extendable and independent of classes. Those distinctive aspects of the instance-based data model make it an open-world system, compared to the traditional model that imposes a closed-world system. Several advantages show that the model is more appropriate for multiple applications in one single data set. In fact, some of the applications might not have been anticipated when the database was constructed.

In this chapter we reviewed several class-based data models. We described the common aspects of each model and pointed out the basic problem of the class-based model. We then introduced the instance-based data model. We introduced the principles of the instance-based data model and its possible data structures. Finally, we compared the instance-based data model with the class-based model and indicated the advantages of the instance-based data model. Since the instance-based data model is developed based on Bunge's ontology, to take advantage of this model, in the next chapter, we will discuss some related ontological issues to address these needs in the instance-based data model.

Chapter 3

Ontology Principles

The instance-based data model can resolve numerous problems that occur in class-based data models due to the restriction that instances must belong to classes. However, the model also faces technical challenges that are not addressed in the original IBDM proposal [2]. Prior to studying applications of the instance-based data model, some concepts and their operations under the model need to be described in detail. We do this in this chapter. In the next chapter, we extend the semantics of the instance-based components and implementation methods for data expressed in the instance-based data model.

3.1 Basic Ontological Principles

"Ontology is the study of being or existence and forms the basic subject matter of metaphysics, describing the basic categories and relationships of being or existence to

define entities and types of entities within its framework" [3]. The ultimate objective of ontology is to study the most general features of reality. In the field of information analysis, ontology is used by philosophers and scientists working in artificial intelligence, database theory, and natural language processing. In recent years, ontology has been introduced to a variety of applications ranging from system design and analysis, to Web services, to biomedical informatics, to the semantic Web. Ontology in these fields can be categorized into such tools as conceptual analysis or as approaches to solve technical problems [55] [10]. Our research focuses on the conceptual analysis category.

Wand and Weber first introduced Bunge's ontology [3] to the information system field [14] [15] [16] [17] [18] [19]. They define a mapping from the ontological concepts into modeling language constructs. They use ontology to analyze the meaning of common conceptual modeling constructs and provide a precise definition of several conceptual modeling constructs. Their research makes building a conceptual model closer to reality. Parsons and Wand developed several information models [2] [57] based on Bunge's ontology to solve problems in current models.

3.2 Things

In Bunge's ontology, the world consists of two types of things, concrete and conceptual. Concrete things are substantial individuals or entities. A concrete thing must exist in the real world. However, conceptual things are concepts that people use to describe the real world. They only exist in the human mind.

A thing can be simple or composite. An individual is composite if and only if it is composed of individuals other than itself and the null individual. Otherwise, the individual is simple. Simple things are basic objects that cannot be subdivided into other objects. A thing can be either a concrete or a conceptual thing. For example, a person is a concrete thing. A family, however, can be considered as a composite of persons.

Things can only be described by properties. If we define all unarized properties possessed by a single instance (we discuss this in more detail in the next section) as **P** and all substantial individuals as **S**, then the totality of unarized properties of a substantial individual $x \in \mathbf{S}$ is called

$$p(x)=\{\mathrm{P} \in \mathbf{P} \mid x \text{ possesses P}\}$$

Using the above notation, Bunge defines a thing as follows:

Definition 1 Let $x \in \mathbf{S}$ be a substantial individual. Then the thing X is the individual together with its unarized properties:

$$\mathrm{X} = \langle x, p(x) \rangle.$$

The definition of thing indicates that a thing should be described by two parts: an individual and a set of properties that the individual possesses. The first part indicates the thing itself and the second part describes the characteristics of the thing. In the absence of either part, the definition of things will be meaningless. This definition closely resembles the logic that people use to describe things in real life. In Figure 12, imagine several bowls on a shelf. If a person is referring to the bowl that is depicted with an arrow in Figure 12, how does this person let others know which bowl he/she is referring to? This person might designate the bowl as “that center large bowl”. In that phrase, ‘that … bowl’ indicates the bowl itself and ‘center large’ describes the characteristics of this bowl. However, neither “that bowl” nor “center large” is sufficient to designate the bowl.

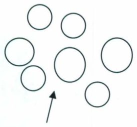


Figure 12: How People Refer to Things

Bunge posits that individuals have at least one unique property.

Postulate 3.1 No two substantial individuals have exactly the same properties. That is: for all $x, y \in \mathbf{S}$, if $x \neq y$ then $p(x) \neq p(y)$.

Postulate 3.1 points out that two different substantial individuals must have two different sets of properties. An immediate consequence of this is:

Corollary 3.1 For all $x, y \in \mathbf{S}$, if $p(x) = p(y)$ then $x = y$.

Here, Corollary 3.1 indicates if two substantial individuals have exactly the same properties, then they themselves are the same one. Following Postulate 3.1 and Corollary 3.1, Bunge concludes as follows:

Corollary 3.2 Everything is identical to itself.

3.3 Properties and Attributes

3.3.1 Properties

Properties can only be possessed by things. The existence of properties depends on things. There are two types of properties: intrinsic (unary) properties and mutual (n-nary) properties. An intrinsic property only describes the characteristics of a single thing, whereas a mutual property refers to the characteristics of more than one thing. For example, the hair color of a person is an intrinsic property, but marriage is a mutual property since it is referring to a relationship between more than one individual.

Bunge's ontology categorizes properties into general properties and specific properties. Informally, a general property represents a common aspect of a set of individuals, but a specific property only represents an aspect of an individual. An intrinsic property may be a general property or a specific property. Whether an intrinsic property is a general or a specific property is not decided by the concept of the property but by the instances that possess it. Bunge defines two kinds of properties:

Definition 3.2 Let $\mathbf{T} \subseteq \mathbf{S}$ be a nonempty set of substantial individuals and $\mathbf{P}$ the set of unarized substantial properties. Then

(i) the set of unarized properties of individual $x \in \mathbf{T}$ is called

$$\mathrm{p}(x) = \{\mathrm{P} \in \mathbf{P} \mid x \text{ possesses } \mathrm{P}\}$$

(ii) the set of unarized substantial properties of **T** is called

$$p(\mathbf{T}) = \{P \in \mathbf{P} \mid \text{For all } x \in \mathbf{T},\quad x \text{ possesses } P\}.$$

As described above, in most cases, an intrinsic property possessed by a set of individual instances should be a general property; an intrinsic property possessed only by a single instance should be a specific property. However, this is not always true. An intrinsic property possessed by a set of individual instances may be a specific property. For example, *people with black hair* is a set of individuals. However, the property they possessed, *with black hair*, can be considered a specific property. Bunge's ontology considers general and specific property as follows:

Postulate 3.2 Let **S** be the set of substantial individuals or some subset thereof, and let **T** to **Z** be arbitrary nonempty sets, equal to or different from **S**. Then:

1. Any substantial property in general is representable as a predicate of the form

   $$\mathbf{A}: \mathbf{S} \times \mathbf{T} \times \ldots \times \mathbf{Z};$$

2. Any individual substantial property, or property of a particular substantial individual $s \in \mathbf{S}$, is representable as the value of an attribute at s, i.e. as $\mathbf{A}(s, t, \ldots, z)$, where $t \in \mathbf{T}, \ldots, z \in \mathbf{Z}$.

That is, the postulate represents general properties as domains (each as a set of all possible values of an independent variable of a function) and specific properties as values of general properties, which resembles the representations in the class-based data models. Properties may be compatible or incompatible with each other.

Definition 3.3 Two properties P_1 and P_2 are incompatible over a set $\mathbf{T} \subseteq \mathbf{S}$ of substantial individuals if and only if possessing one of them precludes having the other. They are mutually compatible over **T** if and only if they are not incompatible over **T**.

A property may be preceded by another property.

Definition 3.4 Let P_1 and P_2 designate two properties. P_1 will be said to precede P_2 if and only if every thing possessing P_2 also possesses P_1.

3.3.2 Attributes

Everything has properties. However, not every property of a substantial individual is human-recognizable; some of them can be ignored. Human-recognized properties are called attributes. An attribute is a representation of a property or a set of properties. Properties are characteristics of things themselves. However, attributes are characteristics assigned to models of things according to human perceptions.

Humans can only recognize properties by attributes. By this definition, all properties possessed by a conceptual thing are attributes. Properties possessed by a concrete thing may not be attributes.

Bunge's ontology formalizes the representation of properties as follows:

Postulate 3.3 Let **P** be the set of all properties and **A** the set of all attributes. The representation of properties by attributes is via a function ρ: $\mathbf{P} \rightarrow 2^{\mathbf{A}}$ such that for each P $\in$ **P**, ρ(P) is a set of attributes $\mathbf{A} \in 2^{\mathbf{P}}$ such that for any a $\in$ **A**, a represents P.

Note that the postulate also indicates that different attributes may represent the same property.

3.4 Relations Between Instances

Several association theories have been proposed to distinguish relations between individuals. Bunge's ontology defines associations by adapting the semi-group theory and assembly.

The term semi-group was defined by Ljapin [58]. A semi-group (monoid) is a structure <**S**, ° >, where **S** is a nonempty arbitrary set and ° a binary operation in **S**. A finite semi-group, <**S**, ° >, means only a finite number of elements is in the set **S**. Bunge's ontology assumes that ° operation is commutative (which means: if $x, y \in \mathbf{S}$, then $x \circ y = y \circ x$) and idempotent (which means: for all $x \in \mathbf{S}$, $x \circ x = x$).

Bunge assumes that the set of individuals is a commutative monoid of idempotents and it is suitable to all real things.

Postulate 3.4 Let **S** be a non-empty set, □ a selected element of **S**, and ° a binary operation in S. Then the structure φ = <S, °, □> satisfies the following conditions:

(i) φ is a commutative monoid of idempotents;

(ii) S is the set of all substantial or null individuals;

(iii) The neutral element □ is the null individual;

(iv) $\circ$ represents the association of individuals;

(v) the string $a_1 \circ a_2 \circ \ldots \circ a_n$, where $a_i \in \mathbf{S}$ for $1 \leq i \leq n$, represents the individual composed of the individuals a_1 to a_n.

By Bunge's ontology, individuals can associate to form further individuals. That is: if $x, y \in \mathbf{S}$, then there is always a $z \in \mathbf{S}$ which makes $z = x \circ y$. So, an individual may be composite.

Definition 3.5 An individual is composite if and only if it is composed of individuals other than itself and the null individual. Otherwise, the individual is simple.

Definition 3.6 If x and y are substantial individuals, then x is part of y if and only if $x \circ y = y$. We use a symbol $x \sqsubset y$ to express this relation.

The symbol $\sqsubset$ indicates the relation, which is what we called the part-whole relation. The part-whole relation follows the rules:

(i) for all $x \in \mathbf{S}$, $\sqsubset$ is *reflexive*. That is $x \sqsubset x$;

(ii) for all $x, y \in \mathbf{S}$, $\sqsubset$ is asymmetric. That is if $x \neq y$ then $x \sqsubset y \Rightarrow \neg(y \sqsubset x)$;

(iii) for all $x, y, z \in \mathbf{S}$, $\sqsubset$ is *transitive*. That is if $x \sqsubset y$ & $y \sqsubset z$ then $x \sqsubset z$.

Following the part-whole relation we can formally define the simple things as:

Definition 3.7 For any $x \in \mathbf{S}$: x is simple if and only if for all $x \in \mathbf{S}$, $y \sqsubset x \Rightarrow y = x$ or $y = \square$.

A thing may be composed of many components. The composition of a composite thing is defined as:

Definition 3.8 The composition of a composite equals the set of its parts. That is if ζ: $\mathbf{S} \rightarrow 2^{\mathbf{S}}$ is a function from individuals into sets of individuals, and if for any $x \in \mathbf{S}$ there is $\zeta(x)=\{y\in \mathbf{S} \mid y \sqsubset x\}$, then $\zeta(x)$ is called the composition of x.

A composite thing may possess two types of properties: one is the properties possessed by its components. Bunge designates these properties as hereditary properties. And the other is the properties that describe the whole composite thing itself. Bunge designates the second type as emergent properties. Of course, some composite things may not have any hereditary property. However, any composite thing must have at lease one emergent property. In fact, the ontology also assumes that the set of properties of a composite thing is not equal to that of all the properties of its components. That is: for all $x, y, z \in \mathbf{S}$, if z is composed by x and y then $p(z) \neq p(x) \cup p(y)$.

3.5 Class and Kind

3.5.1 Class

To define a class, we need to first introduce the concept of scope.

Definition 3.9 The scope of a property, P, is the set of things possessing it. That is, the scope is a function $\varphi : \mathbf{P} \to 2^{\mathbf{S}}$ such that for $P \in \mathbf{P}$, $\varphi(P)$ is the set of all individuals having property P.

After introducing scope, Bunge defines a class as follows:

Definition 3.10 A non-empty subset X of the set of things, **S**, is called a class if and only if there is a property $P \in \mathbf{P}$ such that $X = \varphi(P)$.

Postulate 3.5 The intersection of any two classes of things, if non-empty, is a class.

Since a class is a scope function $\varphi(P)$, or a set of things that possess the same property (property P) in their property set, Postulate 5 indicates that for any two compatible properties $P, Q \in \mathbf{P}$ there is at least a third property $R \in \mathbf{P}$ such that $\varphi(R) = \varphi(P) \cap \varphi(Q)$.

3.5.2 Kind

A single property determines a class. A set of properties will determine a kind. The members of a kind are all the things that share all the properties in the given set. For example if three classes, C1, C2, and C3, have the intersection, K, then K represents a kind (this relation is illustrated in Figure 13).

Definition 11 Let $k: 2^{\mathbf{P}} \to 2^{\mathbf{S}}$ be the function assigning to each nonempty set $\mathbf{R} \in 2^{\mathbf{P}}$ of substantial properties the set, $k(\mathbf{R}) = \cap\ \varphi(P)$, of things sharing the properties in **R**. This value $k(\mathbf{R})$ is called the **R**-kind of things.

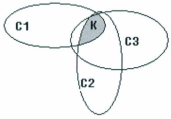


Figure 13: Class and Kind

Note that since **R** is finite, the corresponding **R**-kind is a class.

3.6 Summary

In this chapter, we introduced several ontological concepts related to the instance-based data model. These ontology concepts form the basis for the instance-based data model. In the next chapter we will discuss several semantic extensions of the instance-based data model based on the concepts outlined in this chapter.

Chapter 4

The Semantic Extension of the Instance-Based Data Model

In the last chapter we introduced several ontological concepts related to the instance based data model. In this chapter, we discuss underlying principles of the instance-based data model and their semantic extensions: the semantics of the instance identifier and the proposal of a possible implementation approach; detailed explanations of properties and their relationship; and the definition of classes. We also consider how to represent the real world using the instance-based data model. The research indicates types of relationships between instances in the instance-based data model. Finally, we present several integrity rules for the instance-based data model to assist future investigation.

4.1 Instance Identifier

In Bunge's ontology, an instance possesses properties and people recognize a thing by recognizing its properties. Based on this recognition, Bunge identifies an instance as follows:

If $x \in S$ is a substantial individual and $p(x) \subseteq \mathbf{P}$ the collection of its properties, the individual together with its properties is called the thing X: $X =_{df} <x, p(x)>$ [3].

Following Bunge's definition, in the previous chapter we have described an instance in an instance-based data model consisting of two parts, an instance identifier followed by a set of properties. However, an instance identifier was only referred to but neither the definition nor the implementation method was described in detail. Since the instance identifier is an important part of an instance, in the next section, we will examine instance identifiers to a greater extent.

4.1.1 Semantics of Instance Identifiers

People recognize a thing by its properties; and from the definition of instances, an instance cannot be equal to a set of properties.

An instance identifier has to be outside of the set of the properties of a thing and should not be any human-recognized property (or properties) of the instance, otherwise it would not be consistent with the definition of instances. In fact, an instance identifier is not a property of any thing that occurs naturally, which means instance identifiers do not belong to things themselves, but rather they are the notation that people use to distinguish one instance from another.

Bunge's ontology postulates that '*No two substantial individuals have exactly the same properties*'. That is: $\forall x, y \in \mathbf{S}$, if $x \neq y$ then $p(x) \neq p(y)$. On the other hand, $\forall x, y \in \mathbf{S}$, if $p(x) = p(y)$, then $x = y$. The ontology also postulates that the totality of things is an uncountable set.

Following the above postulates, the semantics of an instance identifier, which people use to distinguish a substantial individual from others, should represent all properties of an instance. One should note that the total number of properties of an instance may be very large [3], whereas the number of properties people recognize and represent via attributes is limited and always less than the total number of properties that the instance possesses.

The expression of an instance in the ontology can be considered a represent-describe model: that is, the instance identifier represents an instance while a set of properties describes the instance.

In the instance-based data model, an instance is an instance identifier followed by a set of properties. In this expression, the instance identifier and the properties describe two different concepts; on one hand, the instance identifier represents the summary of aspects of all properties of the thing and these aspects are only determined by the thing. But on the other hand, the properties are those that humans use to describe the represented particular thing in greater detail. Those properties are the properties of the thing recognized by people but not the actual total number of properties of the thing. That is, when we describe an instance, instance_id{P_1, P_2, …, P_n} in the instance-based data model, it does not mean that instance_id={P_1, P_2, …, P_n}. Therefore, when people begin to recognize more properties of the thing, the attributes increase, approaching the total number of properties of the thing, whereas the instance identifier, the representative of the thing itself, remains unchanged.

In summary, the instance identifier is a unique representation of a thing. Humans use the representation to indicate a thing. It represents the summary of all the properties of the thing. Since only the intrinsic properties are possessed by individual instances, an instance identifier should represent all intrinsic properties of the instance. Ideally, an instance identifier represents a globally recognized unique property of an instance, which can be used to distinguish the instance from others. For example, a human being has one unique brain; no one has the same brain as any other. This representation can be shown as following:

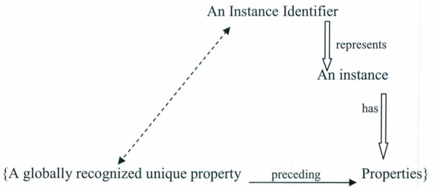


Figure 14: An Instance Identifier Represents a Globally Recognized Unique Property

4.1.2 Creating an Instance Identifier

In class-based data models, the most obvious way to create an unambiguous identifier for every distinct object is to count the objects and assign each object a value. For example, in the relational model, if there is no natural candidate attribute(s) for a key, the database system will alternatively create an identification field and use an ordered number for each record.

In the class-based approach, the number indicates there is a difference but such difference is ambiguous. Whereas the syntactic difference that each record is different from the others is achieved, the semantic difference (how each one is different from the others) is not achieved.

Currently, numerical values are widely used in software design and other industry fields to distinguish instances from each other. For example, manufacturers of automobiles assign a unique serial number on each car they produce. This number works as a real world identifier. Whenever a car is built, this serial number will be attached to the car and will not change. This number also is location insensitive; that is, the serial number remains unchanged wherever this automobile goes. This kind of system involves some aspects of a real instance identifier. It may be different from other approaches, like URLs [63], which are location sensitive. For example, the same website stored on different servers will have different URLs.

However, as noted in [64], to create a unique namespace to define instance identifiers has proven to be difficult, historically. The problem stems "from a confusion between the abstract idea of a global namespace, and the physical devices that have been created over the years to operate in this namespace."[64]

It is possible, in theory, to create a global namespace using a combination of local uniqueness and a tree of identities, like the long distance telephone system, so that any user who understands the path down through the tree could understand the identity of the information objects at the leaves of the tree. This approach is as shown in Figure 15. There are three steps to generate a system:

1. Each instance gets a locally unique identifier from a local system.
2. Several local systems combine together to form a higher level system (level 1).
3. When there are more than one intermediate level systems in the same level, they are combined together to form a higher level system until there is only one highest level system (the root level).

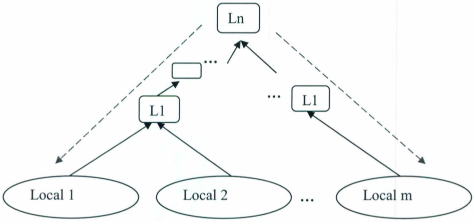


Figure 15: Implementation System of Instance Identifiers

An instance identifier system is generated from the lowest level to the top level. After the system is built, an instance can be identified from the top level down to the leaf level, as indicated by the dash arrows. Our approach is also called a hierarchical approach.

4.2 Properties and Their Relationships

People recognize a thing by recognizing its properties. Properties always belong to instances. However, as Bunge's ontology indicates, attributes of things are only the aspects of things that humans recognize. So, to describe real world things, we have to define what an attribute is and to find relationships between them.

Corollary 4.1 Attributes are concepts that humans use to describe the real world things.

In Bunge's ontology, human recognized properties are called attributes. However, to be consistent with the previous papers of the instance-based data model [2], we still use the name '*property*' not '*attribute*' for these type of properties. In the following, unless otherwise specified, the term property will imply attributes. In contrast, we use '*real property*' to indicate properties possessed by things.

Technically, a property is a concept possessed by instances and recognized by people. However, as suggested in the previous section, a real property of things may be represented by a set of properties and several properties may represent the same real property.

Previously, two types of properties, general property and specific property, were distinguished. A general property represents a domain which may be shared by a set of individuals, while a specific property represents one fundamental aspect of an individual, i.e. the aspect cannot be further divided. A property can be either general or specific. However, it is difficult to define general and specific properties in term of words instead of mathematical expressions. In class-based models, general properties are interpreted as attributes of classes, which describe entity types, and specific properties are interpreted as values of general properties. For example, *weight* may only be manifested as a numerical value in the unit of pounds. Consider the specific property (weight, 150lb), or 150 pounds in weight. Here, the property *weight* is a general property whereas 150lb is a specific property and it can be considered as a value of the property *weight.* Any further specialization of 150lb in such systems is impossible. Following the above logic, the property (weight, 150lb) is a specific property as well, as further specialization is not permissible in this property.

A class may not be solely based on general properties. In Bunge's ontology, a class is a set of individuals that share the same set of properties, which is in the class definition. The shared properties can be either general properties or specific properties. However, in class-based models we have to generate attributes to describe properties of the class and assign values to each individual in the class. In this case, a class definition can only be on general properties. Returning to the weight example, if we need to define a class that includes everything that has the weight of 150lb, we have to use the property (weight, 150lb) which is a specific property as discussed above. However, a general property is required for class definition, but any further specialization is not permissible for specific properties. The simplest way to resolve the situation would be to introduce another specific value for each individual instance, so that the property, (weight, 150lb), can be used to generate the specific values. For the above *weight* problem, the class definition could be ((weight, 150lb)) but any instance that possesses this property will be denoted as ((weight, 150lb), true). Of course, this approach is both redundant and increases the required space. Parsons and Wand introduced two-layered approach to solve the problem [2]. In their approach a class definition can use both general properties and specific properties.

Instead of general and specific relationships, the very simple relationships of properties in the class-based models, we define two types of relationships between properties in the instance-based data model. We call them compatible-related and non-compatible-related relationships between properties.

Compatible-related relationships depend on human recognition of properties. They describe how the relationships between the concepts of properties are related. Three types of compatible-related relationships exist between properties: belongs, joint, and discrete. If the set of concepts of one property is a subset of concepts of another property, then we say the first property *belongs* to the second property. The *belongs* relationship between properties may have hierarchical structures. For example, if there are two properties, *weight* and *weight is 150lb*, then these two properties have a *belongs* relationship. They also have the hierarchical structure of concepts. If the concepts of two properties have an *intersection*, the two properties are called *joint* properties.

Subsequently, if the concepts of two properties do not have any *intersection*, they are called *discrete* properties. In our research, we will largely deal with properties that either have the *belongs* relationship or have the *discrete* relationship. If there is a *joint* relationship between two properties in a system, we will translate them into *discrete* properties. The translation is based on Bunge's ontological assumption that for any two compatible properties P, Q $\in$ **P** there is at least a third property R $\in$ **P** such that $\varphi(R)=\varphi(P) \cap \varphi(Q)$. The steps of translation are shown in Figure 16. The concepts of property P_1 and P_2 have an *intersection.* A new property, P_3, manifests the joint part and we redefine the rest of P_1 and P_2 as P_1' and P_2'. After the translation, the three properties, P_1', P_2', and P_3, are *discrete* properties.

In traditional class-based models, the relationships of properties are expressed between classes. After the schema is defined, relationships between properties are fixed, which means only compatible-related relationships are between them.

The non-compatible relationship between properties was first introduced in the IBDM. To bring more clarity into explaining these kinds of relationships, the IBDM used Bunge's ontological concept of scope.

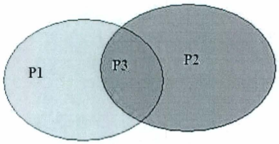


Figure 16: Transforming *Joint* Properties to *Discrete* Properties

In the IBDM, an instance may gain or lose properties. This enables dynamic modification of the scope of a property P, the set of instances which possess the property P.

Now, we introduce two non-compatible relationships between properties, *preceding* and *preceded* [57].

Let $\zeta(P)$ denote the power set of **P**. Then we have:

Definition 4.1: The *preceding* properties of P in **P** are defined by the function Preceding: $\mathbf{P} \rightarrow \zeta(\mathbf{P})$, such that Preceding(P)=$\{Q \in \mathbf{P} | \mathrm{Scope}(Q) \supseteq \mathrm{Scope}(P)\}$.

Definition 4.2: The *preceded* properties of a property P are all properties for which P is a *preceding* property. That is, the function Preceded: $\mathbf{P} \rightarrow \zeta(\mathbf{P})$ such that Preceded(P)=$\{Q \in \mathbf{P} | \mathrm{Scope}(P) \supseteq \mathrm{Scope}(Q)\}$.

Definition 4.1 indicates that a set of instances possessing property P may possess other properties, $Q \in \mathbf{P}$, such that $\{Q \in \mathbf{P} | \mathrm{Scope}(Q) \supseteq \mathrm{Scope}(P)\}$. Conversely, the preceded properties of a property P are those properties such that an instance possessing any of those properties must possess P. Note that whether one property precedes other properties or is preceded by other properties is based on which instances possess the properties. It does not matter whether these properties are in the same domain or not. So, we name them *non-compatible relationships*. Since an instance may gain or lose properties in the IBDM, the above two operations, Preceding(P) and Preceded(P), will generate outputs that will dynamically reflect the semantics, which instances possess P also possess some other properties (Preceding(P)) or instances possess other property also possess P.

Several technologies, such as data mining [65] and data warehousing [66] in particular, have been suggested recently in attempt to analyze a dynamic relationship between properties. Those technologies analyze attributes of instances to get related results. For example, a classic AI problem is market analysis: Who is most likely to buy a computer? A student, a young man with great credit, or a middle-aged white-collar man [67]? To tackle this kind of problem, numerous learning methods such as supervised learning [68] and non-supervised learning [69] have been suggested. By analyzing one set of values of an attribute (or a set of attributes) with values of other attributes, those approaches try to get useful information from data. However, in the class-based models, the statistical analysis is done on values of instances. The values are based on the schema of classes to which the instances belong. We believe that this severely limits the analysis capability in class-based models, and by using *preceding* and *preceded* relationships, the IBDM generates more useful information between properties.

4.3 Relationships Between Instances

In the relational data model, a relationship is defined between entity types. This is why sometimes designers find it difficult to define relationships in this model. It has always been a great confusion to designers whether a relationship is binary, ternary or even more complex [1]. However, expressing relationships in the instance-based data

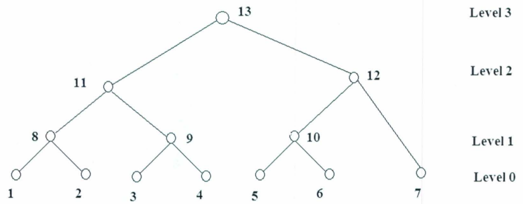


Figure 17: Representing the World at the Instance Level

model is relatively simple. We need only use the binary relationship to represent all relationships in a system.

Bunge's ontology assumes that "*the world is formed of things and relationships between them, and only binary relationship exists between things*" [3]. It also indicates that "*one thing associating with another thing will form a new thing (conceptual thing)*" [3]. Database systems also assume that things associated with each other do not erase any intrinsic properties of things themselves. Following the assumptions above, we express the world of things in two parts: things and associations between things. According to Bunge, an association between basic things forms a higher level conceptual thing. Each level of things stores its own information in its level. So, the whole world in the instance-based data model is modeled as a binary tree structure, as shown in Figure 17. All the basic things are on the bottom level (leaf nodes). All the internal nodes are concepts abstracted from the basic things. Finally, the root is the model of the real world. Our approach to abstract concepts of things is only to store binary relationships of the lower level things into higher levels. Higher level relationships can be formed between things in multiple levels. For example, between the neighboring levels in level 1, thing 10 is abstracted from two lower level things 5 and 6. However, in level 2, thing 12 is formed by thing 10 and thing 7. So, thing 12, in level 2, is actually formed by three things, 5, 6, and 7, in level 0; that is, a ternary relationship forms between level 0 and level 2.

A system may not store all internal nodes. For example, it might form the tree structure shown in Figure 18. A simple comparison with Figure 17 reveals that two internal things, things 8 and 10, are missing. A missing internal thing (e.g. thing 10) means the system does not contain any information relating only thing 5 and thing 6 (from which thing 10 is abstracted). However, for the implementation, when more than two things form a higher level thing, it is also safe to assume an abstract thing exists between two things in lower level so that we can always express the formed thing between two things as we showed in Figure 17.

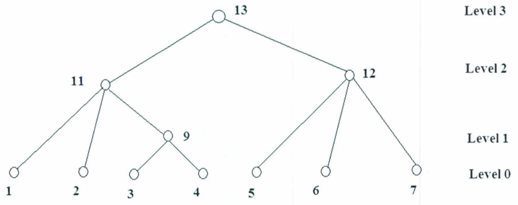


Figure 18: Data Representation at the Instance Level

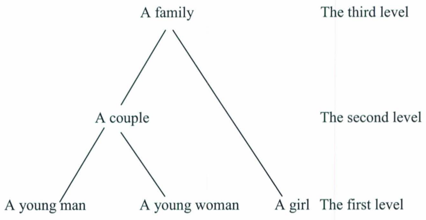


Figure 19: Three-level Structure of a Family

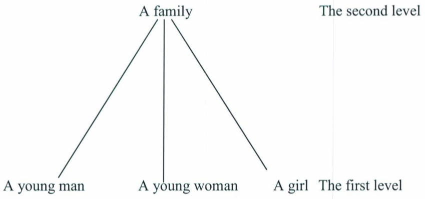


Figure 20: Two-level Structure of a Family

We have an example to illustrate the tree structure formed in the instance level as follows:

Assume there are three instances: a young man, a young woman, and a girl. The man and the woman are married and the girl is their child. We could express the relationship between these instances in three levels as Figure 19. That is, in the bottom level, we have three instances: a young man, a young woman, and a girl. In the second level we have an instance: the young man and the woman married forming a couple. In the top level we have an instance: the couple and the child form a family. Of course, sometimes the middle relationship is missed. In our case, three instances may have relationships in two levels as shown in Figure 20: at the bottom level there are three instances: a young man, a young woman, and a girl; at the top level they form a family.

After defining relationships between instances, the relationship between two individual instances is represented in the highest level thing of the path connecting the two instances. We refer to this thing as the *view-relationship* of the individuals. In a database, all the internal nodes represent view-relationships of individuals. The difference between the levels of the individuals and their view-relationship represents how closely the individuals are related. A lower number means they are closely related to each other. A higher number means their relationship is more distant.

4.4 Class Identifier

In the instance-based data model, we state that a class is a class identifier followed by a set of properties. The semantics of the class identifier is different from the instance identifier. An instance identifier is a representation of an abstraction of all the (possibly very large number of) properties of the instance itself whether or not they have been recognized by people. Thus, the semantics of an instance identifier does not equal to the properties that have been recognized by people. In contrast, by Bunge's ontology, a class is a set of instances which possess a common set of properties. A class identifier is a label that people use to indicate common finite properties recognized by people and shared by a set of instances. So, the semantics of a class identifier is equal to its definition, which is a set of common properties recognized by people.

Since a class identifier only expresses a finite set of properties recognized by humans, its implementation is much easier than the implementation of an instance identifier. To do this, we need first somehow to map the class identifier to its definition. This mapping should be onto, but is not necessarily one to one. Here, onto means a class identifier must map to a set of properties. However, in some cases, it is possible that several class identifiers map to the same set of properties.

4.5 Representing Data Semantics to Support Queries

In class-based models, the semantics of data is represented by the schema defining classes to which instances belong. One schema and its bounded data can only express one level of semantics. The models have no way to answer queries in different semantic levels. For example, if a system stores some information about persons and there is a field to express how tall this person is in centimeters, even a simple query '*identify the tall person*' cannot be answered automatically. The system does not have the ability to transform the semantics of *tall* into centimeters. In the instance-based data model, properties can be preceded by a set of other properties. Back to the above question, in the instance-based data model, we can define a property as tall, which can be preceded by any one whose height is more than 180 cm. So, for the query 'tall person', any one who is higher than 180 cm will be in the results set. A property also can precede other properties [57]. Using this method, a class can be defined as merely one property that precedes a set of properties, which are common properties of the instances in the class. Reducing properties in a class definition will speed up querying and updating the members of the class. For example, if a class can be defined using only one property in the instance-based data model, the speed to query the membership of the class in this model will be as fast as the relational data model [70].

4.6 Integrity Rules

As we have discussed, the instance-based data model separates instances from classes and stores information about instances and classes in two layers, the instance layer and the class layer. To guarantee the consistency of data in databases, we have to build rules for the model. There are four integrity rules: three for components of the instance-layer and one for classes in the class-layer.

Rule 4.1 (Instance integrity) An instance *i* can be inserted into an instance-based database if and only if no identical instance *j* exists in the database. This rule can be formally expressed as:

For all $P_k \in \mathbf{P}$, $k \in \mathbf{N}$ (**N** is a natural number), and $i, j \in \mathbf{I}$, if *i possesses* P_k and *j* possesses P_k are always true, then $i = j$.

Rule 4.1 guarantees that no duplicated object(s) exists in a database. It matches the semantics of the instance identifier. Since an instance identifier includes all the features of the instance and no instance should have completely the same features as other instances in the real world, rule 4.1 assumes uniqueness of the identification of each instance.

Rule 4.2 (Property Integrity): The instance $i\{P_p \mid P_p \in \mathbf{P}\}$satisfies property integrity if and only if for any $P_p \in i$, and $P_q \in i, p \neq q$, then $P_p \cap P_q = \phi$.

Rule 4.2 states that a property P_p of an instance i cannot be in a database if it already has property P_p, or the instance i possesses other properties which is a compatible property of P_p.

Rule 4.3 (Association integrity) An association (a mutual property), mp, of two instances, $i\{P_p \mid P_p \in \mathbf{P}\}$and $i'\{P_{p'} \mid P_{p'} \in \mathbf{P}\}$, exists if and only if:

(a) i and i' exist in database

(b) No another mutual property, mp', between i and i' exists such that mp $\cap$ mp' $\neq \phi$.

Rule 4.3(b) follows the property integrity rule. We assume that associations between things form conceptual things in a higher level. Properties of higher level conceptual things should have the same constraints compared to basic things. Rule 4.3(a) means associations can be formed between things only if the things exist in the database. The meaning of this rule is similar to the requirement of referential integrity in the relational model.

Rule 4.4 (Class Integrity Rule): A class $C\{P_p \mid P_p \in \mathbf{P}\}$satisfies class integrity if and only if the following expressions are always true:

a. For any property $P_p \in C$, $Scope(P_p) \neq \phi$

b. For any properties $P_p \in C$, and $P_q \in C$, $p \neq q$, then $P_p \cap P_q = \phi$.

c. If $\{P_k | k=1,\dots n\}$ is a subset of properties which define class C, then

$Scope(P_1) \cap Scope(P_2) \cap \dots \cap Scope(P_n) \neq \phi$.

In the instance-based data model, the basic information is stored in the instance layer. The class layer is generated from the instance layer. Rule 4.4(a) guarantees that only properties possessed by some instances can be used to define classes. Rule 4.4(b) is similar to Rule 4.2 (property integrity) in the instance layer. Rule 4.4(c) guarantees that any defined class must have some instances belonging to it.

4.7 Summary

In this chapter, we discussed several semantic extensions of the instance-based data model and the recognition concepts based on the instance-based data model. The semantic extension of the components of the instance-based data model clarifies the definition of the model and the implementation of the components simplifies the applications to real database systems. We also discussed relationships between instances in the instance-based data model and expressed these relationships in a balanced tree. The discussion provides a clear approach to build relationship between instances as well as a means to define how an instance is closely related to others by using the tree. Hopefully, this will provide some insights on how to query data efficiently in the instance-based data model. In the final section, we presented several integrity rules for the instance-based data model that reduce the redundant data and make the data more consistent.

Chapter 5

Query Performance in the IBDM

We have discussed in the previous chapters how different data models are suited for different applications and how, unfortunately, no data model is best suited for all applications. For example, a query using the hierarchical data model is generally faster than one using the relational model; however, the flexibility in the former model is considerably less than in the latter. The IBDM provides much additional flexibility to users relative to previous relational data model, but what is its efficiency compared with the class-based models? In this chapter, we investigate the query ability of the IBDM compared to the relational model. The comparison consists of two parts: the first is a theoretical comparison and the second is testing on sample data. The comparison will suggest some suitable application areas of the IBDM.

5.1 Introduction

Currently, SQL or SQL-like query languages are widely used to manage databases. When we query a database using SQL, we always need a *Select* clause, which is equivalent to a relational algebra project operation, to indicate what attributes we need to appear in the result set. The results of an SQL query will be a new temporary table that includes all attributes in the *Select* section. So, in an SQL query a user is looking for a set of attribute values that satisfy certain conditions or restrictions (specified in the *Where* clause). However, in relational database systems a record is stored together as a tuple. Whether a user queries only one attribute or all the attributes of a tuple, the system needs to access the whole tuple (that is, all the attributes of the table). This is a time-consuming process, especially when a table is so large that it cannot be fully loaded into the memory.

Using indexes may speed up some query processing. When attributes are indexed, queries on these attributes only need to search a part of the records to get results. However, indexes are not an original technology of the relational model; they are structures stored in a database to increase the performance of certain queries [71]. Indexes also have the drawback that they reduce the performance of updates, as the indexes themselves also need to be updated. So, when using indexes in a relational database there are always some questions. For example, how many indexes are needed? Which attributes should be indexed? Meanwhile, when a query predicate is not very selective, an index would not be useful and the entire relation would be scanned. In this case, the relational data model will have poor performance.

Recently, C-store [72] has provided a method called "column-stores" to increase performance on analytical workloads such as those found in data warehouses, decision support, and business intelligence applications in the relational model. The idea of "column-stores" is to store records in the column format instead of the row format of the relational data model so that the method will provide more efficient performance on some queries. However, as a method, "column-stores" lacks semantics of several operations. For example, what is a record ID? How is it generated? The method also has a drawback for mapping a record to its logical schema.

Compared to the relational model, the instance-based data model provides a new way to access a portion of the attribute values of a class (i.e., a table in the relational model) to answer a query. In the following section, we compare the differences between methods of relational queries and the instance-based queries for two basic SQL queries: select and project query, and join query.

5.2 Environment of the Comparison

We begin by defining an environment that is convenient for our comparison. In a real database system, data types of attributes are not the same. Currently, commercial databases support various data types which we may use for storing information. In most cases, the number of bits required to store a value of an attribute in a table is much larger than an instance identifier, which is a numeric type. For example, a company database as shown Figure 21 in Fundamentals of Database Systems [1], the attribute first name is varchar(15), which needs 2*15=30 bytes in Java, the minit needs 2 bytes, the lname needs 2*15=30 bytes, and so on. The total number of bytes needed for this table is 192 bytes per tuple. Since there are 10 attributes in the table, the average number of bytes needed to store a value of an attribute in the employee table is 19.2 bytes, which is nearly five times more in size compared to the instance identifier (normally it needs 4 bytes).

```
Create table employee
                    ( fname     varchar(15)   not null,
                    minit    char,
                    lname    varchar(15)   not null,
                    ssn      char(9)       not null,
                    bdate    date,
                    address  varchar(30),
                    sex      char,
                    salary   decimal(10,2),
                    superssn char(9),
                    dno      int           not null,
                    constraint empsuper_pk
                    primary key (ssn) );
```

Figure 21: An Employee Table

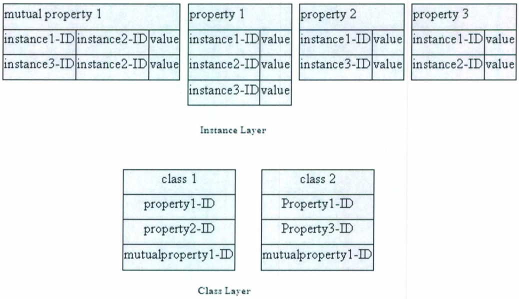


Figure 22: The Second Data Structure of IBDM [70]

However, to simplify the comparison, we assume that all the attributes have the same data type; they have the same data type as the instance identifier, and the length of that data type is q bits. Also, we assume each class (table) has m attributes and n records. Then, to store a table, in the relational database we need at least $q*m*n$ bits, but in the instance-based data model (we use the second data structure introduced in [70], shown in Figure 22, for the comparison) we need $2*q*m*n$ bits. It seems that the instance-based data model needs double the space to store the data. However, when we consider more than one related table, the relational model needs to repeatedly store some information in both tables, for example, the foreign key attributes; therefore, the instance-based data model needs somewhat less than twice the storage space than that of the relational model.

In this chapter we compare the original relational database model with the IBDM; the comparison only uses indexes of the primary attribute(s).

5.3 Select and Project Queries

In the relational data model, for any query purpose, the system needs to load all the records into the memory first, which requires time $(qmn)*v$ (v is the time needed for loading one bit of data to memory). After loading the data to memory, the system will compare the conditions in the *Where* clause; we assume this step takes time u for one comparison. There are n records to be compared, the time needed for this process is $n*u$. We will ignore the time spent to display the results. So the total time for the select/ project query in the relational data model is:

$$(qmn)*v + n*u \tag{1}$$

In the instance-based database, the data that the system needs to load into memory for the query are the values of the attributes that are in the *Select* clause and *Where* clause. If we assume there are p attributes to answer a query, the data size loaded is: $2*q*p*n$. Thus, the time required to load those data is: $(2qpn)* v$. After loading, we need to combine values of attributes to records. This process will consume time (we assume data is stored in a B^+ tree):

$n*(p-1)*\log_B(2qpn)*u$ (where B is the page size of the operating system).

Then the comparison takes time $2*n*u$ (one for value comparison, one for class check). So, the total time is:

$$(2qpn)* v+ n*(p-1)*\log_B(2qpn)*u + 2*n*u. \qquad (2)$$

In expressions (1) and (2), the only variable is n. By the theory of complexity [73] when n grows larger, the value of expression (2) will grow at a much faster rate than expression (1), so that the value of (2) is larger than (1). This is true when n is large enough. However, how large is large enough? With the increasing memory capability of computers, is there an n large enough to make the value of expression (2) greater than the value of expression (1)? We know that in a system the capacity of the memory is always less than the hard drive (typically less than 1 percent) and the access speed of the hard drive is less than that of the memory (typically less than one millionth). So, in a system we have to consider both factors. If we assume the memory size is M, then the time needed for relational system will be:

$$(qmn/\mathrm{M}-1)*v *\mathrm{M}*2+ n*u. \qquad (3)$$

For the instance-based data model, the time needed will be:

$$(2qpn/\mathrm{M}-1)*\ v*\mathrm{M}*2+\ n*(p-1)*\log_{\mathrm{B}}(2qpn)*u+2*n*u. \tag{4}$$

Then the question becomes: how large must *n* be to make (3) less than (4)? That is:

$$(qmn/\mathrm{M}-1)*v\ *\mathrm{M}*2+n*u < (2qpn/\mathrm{M}-1)*\ v*\mathrm{M}*2+n*(p-1)*\log_{\mathrm{B}}(2qpn)*u+2*n*u \tag{5}$$

Add v *M*2 on both sides:

$(qmn/\mathrm{M})*v\ *\mathrm{M}*2+n*u < (2qpn/\mathrm{M})*\ v*\mathrm{M}*2+n*(p-1)*\log_{\mathrm{B}}(2qpn)*u+2*n*u$

Divide by n on both sides:

$(qm/\mathrm{M})*v\ *\mathrm{M}*2+u < (2qp/\mathrm{M})*\ v*\mathrm{M}*2+(p-1)*\log_{\mathrm{B}}(2qpn)*u+2*u$

Delete M on both sides:

$(qm)*v\ *2+u < (2qp)*\ v*2+(p-1)*\log_{\mathrm{B}}(2qpn)*u+2*u$

Move the left to the right and switch the sides:

$(2qp)*v*2+(p-1)*\log_{\mathrm{B}}(2qpn)*u+2*u-(qm)*v\ *2-u > 0$

That is:

$(2qp)*v*2-(qm)*v\ *2+u+(p-1)*\log_{\mathrm{B}}(2qpn)*u > 0$

Finally, Expression (5) can be simplified into:

$$((2p-m)*\ v\ *2\ *q+u)+(p-1)*\log_{\mathrm{B}}(2qpn)*u > 0 \tag{6}$$

Since $p>=1$, we know that when $p=1$, expression (6) will become expression (7):

$$(m-2)*\ v\ *2\ *q > u \tag{7}$$

In the relational database we have $m>=2$ (the number of attributes in a table is at least 2), and in most cases $m>2$. We also know that $q>0$ (the number of bits required to store a value of an attribute is always positive) and $v > u$ (loading data from hard drive to memory is always slower than computing). So, in this case, expression (7) will be true whatever value n adopts. That is to say that the instance-based data model will always be faster than the relational model when we query only one attribute.

In the next step, we will consider the situation of $p>1$. In this case, (6) can be simplified as follows:

$$\log_B(n) > ((m-2p)^* v^* 2\ ^* q / u - 1)/(p-1) - \log_B(2qp) \quad (8)$$

On the right hand of expression (8), since on average, a query will not select more than ten attributes, we assume p<10. We also assume that the space needed to store an attribute is not more than 20 bits (q<20). Moreover, the page size of the operating system is assumed to be larger than 32K (B>32K). Therefore, we can reasonably assume that logB(2qp) is much less than 0.01. Let L = v *2 *q /u, then L>>100 (v will be millions of times larger than u). Then we consider three situations:

1) $m>2p$

This is the most general case when users query a database. In this case, the right hand of expression (8) will be larger than 10 (m-2p>1, L= v *2 *q /u >100, p-1<9). So, if the left hand side in expression (8) is greater than the right hand side, n must be larger than B^{10}. Even if B equals to 32K (which is the minimum value that B could take), B^{10} will be bigger than 10^{45}. In reality, there is no possibility of having a database that includes equal to/more than 10^{45} records in a table. So, there hardly exist any n to establish expression (8). This implies that there is no n in any real system that will result in a query in the relational data model being faster than the same query in the instance-based data model. That is, when query attributes are less than half of the attributes in a table, the instance-based data model will be faster than the relational model.

2) $m=2p$

In this case, expression (8) will become:

$$\log_B(n) > -1/(p-1) - \log_B(2qp)$$

Since $p<10$, a relatively small n will make the left hand side in expression (8) larger than the right. In other words, we do not need a large number of records in a table to make query in the relational data model faster than the instance-based data model. That is, when query attributes are equal to half of the attributes in a table, queries in the relational data model faster than the instance-based data model.

3) $m<2p$

In this case, the absolute value of the right hand of expression (8) will be larger than 10 but it is negative. So, n only needs to be larger than 1 or 2 for expression (8) to hold. This implies, when query attributes are more than half of the attributes in a table, queries in the relational data model will be faster than the instance-based data model.

As explained before, in a real database the average size of attributes is more than 3-4 times of the size of the instance identifier. So, the results of the above comparison should be:

when m>1.25p to1.33p, query on the instance-based data model will be faster than the relational model; when m<=1.25p to 1.33p, query on the relational model will be faster than the instance-based data model.

Although, the above comparison only shows one condition in which case the instance-based data model will be faster than the relational model, a broad range of queries fit this case. Thus in the most cases, queries on the instance-based data model are faster than queries on the relational model.

5.4 Join Query

There are two kinds of join operations in the relational data model: join on the key attribute and join on non-key attribute. However, in the instance-based data model they are the same.

5.4.1 Join on the Key Attribute

In the relational data model, the system needs to load at least two tables. So it needs time: $2*(qmn)*v$. Then the join operation needs time: $n* \log_B(qmn)*u$. So, the total time needed is:

$$2*(qmn)*v + n* \log_B(qmn)*u \qquad (10)$$

5.4.2 Join on non-Key Attribute

In the relational data model, the system still needs to load at least two tables, but the join operation itself needs more time. In this case, the join operation needs time $n*n*u$. In total, the time needed is:

$$2*(qmn)*v + n*n*u \qquad (11)$$

In the instance-based data model, the system only needs to load p attributes plus a mutual property. So, the loading time needed is: $(p+1)qn* v$. And there is no join needed in the instance-based data model. The only operation is a select operation (on key attribute), so, the needed time is: $u^* \log_B(n)$. Then the total time needed is:

$$(p+1)qn^* \, v + u^* \log_B(n) \qquad (12)$$

Since $p<m$, (12) will always be less than (10) or (11).

5.5 Test

A direct comparison of query performance of an instance-based DBMS with a relational DBMS is difficult, since commercial relational DBMSs incorporate proprietary query optimization techniques. In contrast, query optimization for the instance-based data model has not yet been studied. To provide a baseline for comparison, we implemented an instance-based data structure using a commercial relational DBMS based on the second data structure described earlier. We used a relational database to manage most parts of the instance-layer data, and binary and ternary relations to store properties. We implemented each intrinsic property as a binary relation with the property name as the name of the relation. In each binary relation, there is a key attribute Instance_ID and an attribute Value. The key attribute Instance_ID stores the identifiers of instances that possess the property, while the attribute Value stores the value of the property for each instance possessing the property. We implemented each mutual property as a ternary relation. Each ternary relation has three attributes. Instance_ID1 and Instance_ID2 form a key of the relation and indicate two instances related to each other by this mutual property. Also, another attribute Value in the relation is used to store a value of the mutual property between the instances (if there is one). In this way, we implemented all the architectural components below the Query or Update Algorithm in the instance layer of Figure 23 on a relational platform. However, the relational platform cannot implement

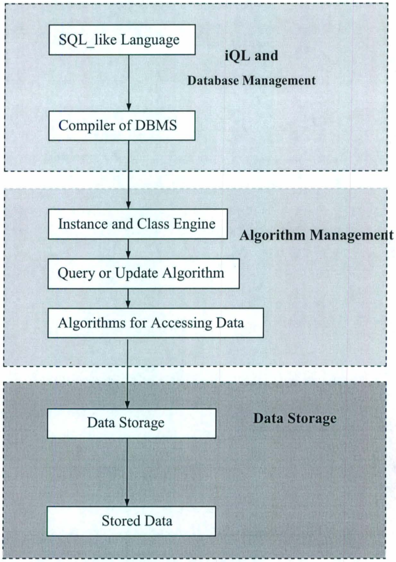


Figure 23: Architecture of the Instance-based Database System

all parts of the instance-layer. We implemented the Query or Update Algorithm and the instance engine of the instance-layer using a separate program that interacted with the relational database. The Query or Update Algorithm includes the methods and algorithms that relate to specific query or update operations and a translator that translates these methods and algorithms to relational operations.

5.5.1 Environment of the Test

We used MySQL [86] as the underlying platform to implement two databases, one based on the relational model (referred to as Rdb) and the other based on the instance-based model (referred to as idb). To compare query performance, both databases were based on the Scalable Wisconsin benchmark data set [56]. This data set includes three relations, ONEKTUP, TENKTUP1, and TENKTUP2. Each relation is composed of the thirteen integer attributes and three 52 byte string attributes. ONEKTUP has 100,000 records. TENKTUP1 and TENKTUP2 both have 1,000,000 records. The structure of each relation is shown in Figure 24. Further details are available in [56].

In the instance-based implementation, we used MySQL to store the instance-layer data, consisting of all the intrinsic properties and the mutual property. Each intrinsic property was implemented as a binary table, where the table name is a Wisconsin attribute name, the instance-id is an integer (not an existing identifier from one of the Wisconsin relations), and the property value is the value of the corresponding attribute. A mutual property was implemented as a ternary table to serve as a join attribute. However, unlike the relational model, we stored the class layer information in a class folder. Each class was stored as a file with the class definition under the class folder. Therefore, three relations in the relational database become three classes in the instance-based model: Tenk1 {unique2}, Tenk2 {unique22}, and Onek{unique23}. A class is defined as all common properties of a set of instances. However, for efficiency purposes, we essentially index class membership using a single property that, in turn, is possessed by all instances possessing all the properties that define the class.

Attribute Name	**Range of Values**	**Order**	**Comment**
unique1 order	0-(MAXTUPLES-1)	random	unique, random
unique2	0-(MAXTUPLES-1)	sequential	unique, sequential
two	0-1	random	(unique1 mod 2)
four	0-3	random	(unique1 mod 4)
ten	0-9	random	(unique1 mod 10)
twenty	0-19	random	(unique1 mod 20)
onePercent	0-99	random	(unique1 mod 100)
tenPercent	0-9	random	(unique1 mod 10)
twentyPercent	0-4	random	(unique1 mod 5)
fiftyPercent	0-1	random	(unique1 mod 2)
unique3	0-(MAXTUPLES-1)	random	unique1
evenOnePercent	0,2,4,...,198	random	(onePercent * 2)
oddOnePercent	1,3,5,...,199	random	(onePercent * 2)+1
stringu1	-	random	candidate key
stringu2	-	random	candidate key
string4	-	cyclic	

Figure 24: Attribute Specification of Scalable Wisconsin Benchmark Relations

We applied the modified Wisconsin benchmark queries to both databases. The Rdb versions of the queries used are listed in the Appendix. The idb equivalent versions were constructed as relational algebra operations (and consequently SQL queries) over the properties implemented as relations. The main activity in constructing these queries was to translate a property selected in iQL to a corresponding relational table storing that property. We tested all queries in the two databases running on a personal computer in the *Windows 7* environment. The processor of the computer is Pentium(R) Dual-Core CPU T4400 at 2.20GHz.

5.5.2 Results of the Test

We tested all queries in the two databases. Selected comparative results are shown in Figures 25-27. Complete results are provided in the Appendix.

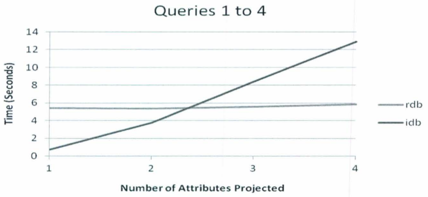


Figure 25: Non-indexed property queries[1]

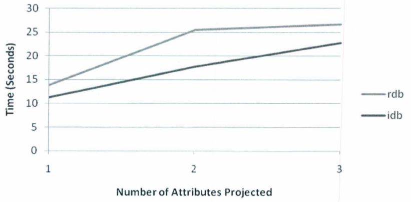


Figure 26: Non-Clustered-Index property queries

[1] Clustered-index properties provide similar results (see Appendix).

From Figure 25 (and the Appendix) we can see that for Non-indexed property queries and Clustered-Index property queries, idb is faster than Rdb when a small number of attributes is projected in the query. These results are somewhat less favorable to the IBDM than we would generally expect to observe because a higher proportion of the attributes in the Wisconsin benchmark (13 of 16) are of integer data type than would typically be the case in most business datasets. As analyzed in the previous section, if a table has more text attributes than integer attributes, the instance-based model will perform better than the relational model. We believe that if half or more of the attributes in the relations were text, idb will be faster than Rdb on projections of 40% or more of attributes.

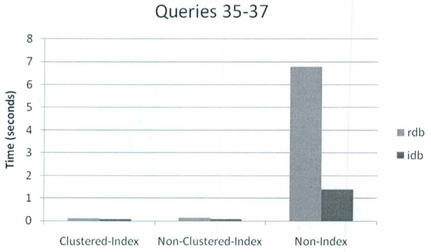


Figure 27: Update Operations

Figure 26 indicates that for Non-Clustered-Index property queries, the idb model has a greater performance advantage over the Rdb relative to clustered-index property queries. In this case, the idb is faster than the Rdb even on projection of more than 40% of the attributes of a class.

Figure 27 indicates that the idb and the Rdb have comparable query performance on clustered-index update operations (similar results hold for aggregation queries – see Appendix). However, when these operations deal with non-clustered index or non-indexed data, the idb is much faster than the Rdb.

Overall, these results provide ample query performance support for the viability of instance-based data structures as a mechanism for organizing data. By implementing these structures in a relational database environment and comparing performance to an equivalent benchmark based on a traditional relational (multi-attribute) design, we prove that the instance-based structures can perform better than the relational model in some queries. We believe that, with a native DBMS implementation and the development of appropriate query optimization techniques for the IBDM, further gains in performance over relational data structures are possible. Our objective was not to show that instance-based structures outperform class-based ones on all operations, but to show that reasonable performance could be achieved on a range of operations.

Notwithstanding this, the results of this comparison are quite interesting, and immediately suggest an approach to substantially improve database performance within a traditional relational database management system for queries that project a few attributes from a class. This requires reconceptualizing traditional thinking about database design. Unlike traditional methods in which a class-based conceptual model developed using the Entity-Relationship model is converted to a class-based relational design, our proposed alternative would implement an instance-and-property based design based on binary and ternary relations, with some class support defined outside the DBMS.

5.5.3 Summary of the Test

We show how an instance-based database can be supported using a relational database platform and demonstrate that this approach leads to faster query processing than an equivalent relational design, even though it does not provide "native" support of the IBDM. From this result, we conclude that database query performance can be improved by taking an instance-based approach to implementing a design using a relational database management system.

5.6 Summary

In this chapter, we compared the query performance in the IBDM with that in the relational model. A theoretical comparison and an empirical simulation show that the instance-based data model is faster than the relational model on some typical queries. Although we did not compare all possible operations, the results demonstrate the speed advantage of the instance-based data model. That is, the instance-based data model has much more flexibility than the relational model [2] and it can perform better than the relational model in certain applications.

Chapter 6

Multilevel Security Model

Multilevel database systems have been proposed to address the increased security needs of database systems. The word “multilevel” means that there are multiple clearance levels. A multilevel database is intended to provide the security needs for database systems that contain data at a variety of security classifications and serve a set of users having different clearances [74]. In multilevel databases, higher-level security users can access lower-level security data but not *vice versa.* If lower-level security users can use any means to access higher-level security data, directly or indirectly (e.g., by guessing), then the security system is termed *broken*. Such a means to allow a lower-level security user to access higher-level security data is referred to as a *covert channel* [1]. The basic motivation of multilevel database systems is to share data from different clearance levels but prevent any covert channels between levels. Many multilevel security database models have been proposed. Different models have advantages for different applications. For example, the Bell-LaPadula model [75] addresses two basic needs of multilevel security systems: (1) a lower-level user cannot read any higher level data; and (2) a higher-level user cannot update any lower-level data. The seaView security model provides an applied multilevel security database system by extending the standard relational model [74]. Several extensions of the seaView model have been proposed, for example the multilevel relational data (MLR) model [76] and the belief-consistent multilevel secure relational data (BCMLS) model [77]. However, the Bell-LaPadula model is the first one which clearly defines a situation where a multilevel security database may need to secure its data.

6.1 The Research Field Overview

The Bell-LaPadula model is expressed in terms of objects and subjects. An object is used to express a passive entity such as a record or a field within a record. A subject is used to express an active process that can request access to objects. A subject refers to a user in this thesis. Every object is assigned a classification and every user has a clearance. Classifications and clearances are expressed as labels, which signify the sensitivity of information. Labels are in hierarchical order of sensitivity, which means that higher hierarchical level labels are more sensitive than lower level ones. For example, a business information system might define levels *Top Secret*, *Secret*, *Confidential* and *Unclassified* with sensitivity labels L_1, L_2, L_3 and L_4 (here, the hierarchy is defined as $L_1>L_2>L_3>L_4$).

Given two labels, L_1 and L_2, the Bell-LaPadula model proposes that the following two restrictions should be applied on all data accesses:

1) "No Read Up": A user assigned at level L_1 is authorized to read an object assigned at level L_2 if and only if the user's label L_1 is higher than or equal to the level of the object's label L_2.

2) "No Write Down": A user assigned at level L_1 is authorized to modify an object assigned at level L_2 if and only if the object's label L_2 is higher than or equal to the level of the user's label L_1.

An easy implementation of the Bell-LaPadula model in the relational database is the tuple-level labeling model [78]. In the tuple-level labeling model, each record is considered as an object of the Bell-LaPadula model and assigned to a security level. Users are also assigned to different security levels. They access records according to the above two Bell-LaPadula's restrictions. A simple example of the tuple-level labeling model is illustrated in Figure 28.

Label	Name	Age	Home Phone
L_1	John	21	(709)737-1234
L_2	John	21	(709)737-1234
L_2	Alice	25	(709)781-4321

Figure 28: An Example of the Tuple-level Labeling Model

In the tuple-level labeling model, a tuple is assigned a single label, so the system can either allow or deny a users access to a tuple. For instance, in the above example, if $L_1 > L_2$ (the security level L_1 is higher than the security level L_2) and if a L_2 level user attempts to query the table, it can only access the second and third records. It cannot read or write any portion of the first record.

Compared to the tuple-level labeling model, in the element-level labeling model [78] each attribute of a record can be assigned a security level. Different attributes of a record may be assigned different security levels. Figure 29 illustrates an example table using the element-level labeling model.

Name	Label	Age	Label	Home Phone	Label
John	L_1	21	L_2	(709)737-1234	L_1
Alice	L_2	25	L_2	(709)781-4321	L_1

Figure 29: An Example of the Element-level Labeling Model

In Figure 29, the first record's *name* and *home phone number* are assigned to L_1 level. However, the age is assigned to L_2 level. The same strategy can be applied to the second record, as well; the two attributes *name* and *age* are assigned to L_2 level but *home phone attribute* is assigned to L_1 level.

In this model users are also assigned to different security levels; the accessibility of records is based on the accessibility of each attribute in the record to users, governed by the above two Bell-LaPadula's restrictions. In contrast to the tuple-level labeling model, in this model a user may access a portion of a record (tuple). For example, a user at the L_2 security level querying the table in Figure 29 will get the results in Figure 30.

Name	Age	Home Phone
Null	21	Null
Alice	25	Null

Figure 30: A Query Result for an L2 User in the Element-level Labeling Model

Compared to the tuple-level labeling model, the element-level labeling is more flexible. For example, properties in one record may be assigned to different security levels as shown in Figure 29. However, this flexibility also creates a problem for this model. Since the elements of a record can be assigned to different levels, a lower level user may get a lot of null values (we call it the null value problem) in this model as shown in Figure 30.

Smith and Winslett proposed another multilevel security data model, the Smith-Winslett model [80]. It combines the aspects from both the tuple-level labeling and the element-level labeling models. In this model, each record is assigned to a security level, identical to the tuple-level labeling security model; the difference with the tuple-level labeling model lies in the fact that in a record the elements may belong to a security level lower than the tuple-level. A sample table based on the Smith-Winslett model is shown in Figure 31.

TC	Name	Label	Age	Home Phone
L_1	John	L_2	21	(709)737-1234
L_2	John	L_2	21	(709)737-1234
L_2	Alice	L_2	25	(709)781-4321

Figure 31: An Example Data of Smith and Winslett Model

TC expresses the tuple-level label. Label expresses the element label. In Figure 31, two records, the first and second, have the same values on attributes. The only difference is the tuple-level labeling of the two records. The security level TC of the first record, L_1, is higher than its element label, L_2. The major difference between the Smith-Winslett model and the tuple-level labeling model is that the Smith-Winslett model is belief-based, in which a higher level tuple TC may "believe" a lower level record and borrow the lower level data as its record. In Figure 31, the first record believes the second record (lower level record) and borrows the data from it. The greatest advantage of the Smith and Winslett model is that it achieves semantic integrity at the tuple level. For example, in Figure 31, if a L_2 level user updates the attribute *Age* value of the second record from 21 to 22, the first record, which is a L_1 level record, will automatically update its record value since the L_1 level users believe the L_2 level record. However, if the same update happened in the tuple level labeling model, users have to update the data in each level. In the model, different levels' records cannot co-operate with each other. For example, in the tuple-level labeling model as shown in Figure 28, when an L_2 level user updates the attribute *Age* value of the second record from 21 to 22, the first record, the L_1 level record, will not update its value since the L_1 level's data are separated from the L_2 level's data even though they represent the same object.

Although the Smith-Winslett model is a belief-based model, it is based on the tuple-level. In the model, a higher-level security record either believes an entire lower-level record or builds its own record. It cannot believe partial lower-level record. This limitation reduces its flexibility.

The SeaView security model was proposed by Denning and Lunt in 1987 [74]. It was also the first multilevel model used in practice. The model labels each record at the tuple level and also at the attribute level. The typical data format of SeaView security model is shown in Figure 32.

TC	<u>Name</u>	Label$_1$	Age	Label$_2$	Home Phone	Label$_3$
L$_1$	<u>John</u>	L$_1$	21	L$_2$	(709)737-1234	L$_1$
L$_1$	<u>John</u>	L$_2$	25	L$_2$	(709)737-1234	L$_1$
L$_2$	<u>John</u>	L$_2$	21	L$_2$	(709)737-1234	L$_2$
L$_2$	<u>Alice</u>	L$_2$	25	L$_2$	(709)781-4321	L$_2$

Figure 32: Typical Data of the SeaView Security Model

The SeaView security model integrates the entity and the referential integrity rules of the relational model with the security model. It also proposes a new concept, *polyinstantiated* data. In the SeaView model, a multilevel relation may have multiple tuples with the same primary key with different security levels. These tuples are referred to as polyinstantiated tuples. Another type of polyinstantiated data is the polyinstantiated element. Polyinstantiated elements are elements identified by a primary key, the security level of the primary key, and an element security level, so that multiple elements for an attribute may have different security levels but are associated with the same (primary key, security level of primary key) pair. By introducing the polyinstantiation integrity rule, the SeaView security model is able to solve most security problems in relational databases. However, some problems still remain. For example, null values are not allowed in a database based on the SeaView model. But it is common to have null values in some attributes of a record in the application of relational databases. So, the restriction limits its applications.

The Sandhu-Jajodia model [79] is a derivation of the SeaView security model. The basic difference between the SeaView security model and the Sandhu-Jajodia model is that there are more restrictions in the insertion and update operations in the latter. In the Sandhu-Jajodia model, a given entity can be assigned to only one tuple on each security level. For example, in the Sandhu-Jajodia model, the first and the second records of Figure 32 cannot co-exist in the database since they express the same entity (the same key value) on the same security level. This restriction is based on the assumption that the same level security users should have the same view on one object. It reduces the ambiguity when multiple records represent the same object at certain security level but have different information. The Sandhu-Jajodia model achieves the semantic integration in the tuple level through this restriction. Typical data in the Sandhu-Jajodia model will be similar to data shown in Figure 33.

TC	Name	$Label_1$	Age	$Label_2$	Home Phone	$Label_3$
L_1	John	L_1	21	L_2	(709)737-1234	L_1
L_2	John	L_2	21	L_2	(709)737-1234	L_2
L_2	Alice	L_2	25	L_2	(709)781-4321	L_2

Figure 33: An Example Table for the Sandhu-Jajodia Model

The Sandhu-Jajodia model provides security control by using tuple-level labeling combined with the flexibility of the element-level labeling. By playing with the requirements, an object must have the same view in the same security level; the model achieves data view integrity in each security level.

The multilevel relational model (MLR model) [76] is substantially based on the Sandhu-Jajodia model and it is an improvement of the SeaView security model. It also combines the belief-based semantics used by the Smith-Winslett model. The model introduces several new concepts. For example, higher level security data can be borrowed from lower security levels. This approach reduces some data redundancies in databases. The model introduces the uplevel statement, so that lending a lower level security data to a higher level becomes easier. The model also introduces data-borrow integrity rules that ensure the consistency of data in the lower-level with the data borrowed in higher levels. This rule eliminates ambiguity and retains upward information flow which exists in the SeaView security model and its extensions. For example, according to the data-borrow integrity rules, in Figure 34, if the second record is deleted from the table then the age of the first record should set to null since no L_2 level value of this attribute can be borrowed. However, since the SeaView security model does not allow null values, the system may not allow this deletion or it may allow the lower level user to delete data at its level (the second record) but the value remains unchanged at the higher level (the first record). In the first case, a covert signal channel will result. In the second case, ambiguous information will appear since there is no lower level view of data in the database. In addition, Sandhu-Chen proved the soundness, completeness, and security of the model.

	Name	Label$_1$	Age	Label$_2$	Home Phone	Label$_3$
L_1	John	L_1	21	L_2	(709)737-1238	L_1
L_2	John	L_2	21	L_2	(709)737-1234	L_2
L_1	Alice	L_2	25	L_2	(709)781-4321	L_2
L_2	Alice	L_2	25	L_2	(709)781-4321	L_2

Figure 34: An Example Data of the MLR Model

The Belief-Consistent Multilevel Secure Relational Data (BCMLS) model is a belief-based multilevel security model [77]. It extends the SeaView security model by allowing higher-level users to interpret the information on lower-level data (Figure 35).

TC	Name	Label$_1$	Age	Label$_2$	Home Phone	Label$_3$
L_1	John	L_1	21	L_1	(709)737-1238	L_1
L_2- L_1	John	L_1L_2	21	L_1L_2	(709)737-1234	L_2- L_1
L_1 L_2	Alice	L_1L_2	25	L_1L_2	(709)781-4321	L_1L_2

Figure 35: An Example Data of the BCMLS Model

In the BCMLS model, a tuple or an attribute may be assigned more than one label. In Figure 35, the second record at tuple level (TC) has two labels and a hyphen sign between them. In the model, any label before hyphen means that users of this level believe the labeled object (a tuple or an attribute) is true. Any label after the hyphen means that this level of users do not trust the labeled object. For example, the tuple level labels of the second record in Figure 35 means that the L_2 level users believe this record but the L_1 level users do not believe the record.

By introducing more than one level of security label to a tuple or an attribute, the model provides an unambiguous interpretation to users. For example, when an L_1 level user queries the table in Figure 35, it will obtain the results shown in Figure 36 (a), whereas L_2 level users will obtain different results as shown in Figure 36 (b). Users at each level will obtain exactly one record for each entity. So, the BCMLS model addresses the interpretation problem.

Name	Age	Home Phone
John	21	(709)737-1238
Alice	25	(709)781-4321

(a)

Name	Age	Home Phone
John	21	(709)737-1234
Alice	25	(709)781-4321

(b)

Figure 36: Unambiguous Results to the Users in the BCMLS Model

From the above introduction, one can see that after constructing the basic multilevel security model, either the tuple-level labeling model or the element-level labeling model, researchers have attempted to add constraints to reduce ambiguity and improve data sharing in databases. For example, to reduce ambiguity, the Smith-Winslett model introduces the belief-based constraint to achieve semantic integrity at the tuple level; the Sandhu-Jajodia model adds the entity rule to achieve a constant view of an entity in each security level; the MLR model introduces the borrow concept to achieve semantic integrity at either the tuple or the attribute level. Increasing labels, of course, will increase the complexity of accessing of data. On the other hand, the belief-based constraint increases the data sharing between higher-level and the lower-levels and finally the BCMLS model combines some higher-level data and lower-level data together.

We have reviewed several main data models that provide multilevel database security. Although some other successful proposed models exist, most of them are related to the introduced models. However, although many models have been proposed, some problems remain unsolved. In the following sections, we will point out some problems of the current models. In the next chapter, we introduce a new model, the instance-based multilevel security model.

6.2 Problems with the Traditional Models

Multilevel security database systems face many challenges. A multilevel security database system is governed by the two restrictions that Bell-LaPadula proposed above. The purpose of the two restrictions is to avoid any covert channel between security levels, including direct and indirect covert channels. However, when applying the two restrictions to a data model, in all the above models, there is a tradeoff between sharing data and getting more security. Some of the problems are listed below.

6.2.1 Data Redundancy

The SeaView security model and its derivatives define the polyinstantiation integrity to achieve the goal of protecting the higher security data. However, the polyinstantiation integrity rule is not a data-sharing rule. The basic principle of the polyinstantiation integrity rule is to split data into security levels. For example, the relation in Figure 37 is a polyinstantiation example in a multilevel security database. *Name* is the primary key of the relation and the security classifications are assigned at the granularity of individual data elements. *TC* (namely L_1, L_2, L_3, and $L_1 > L_2 > L_3$) is the tuple class of each record.

Name		Weight		Age		TC
John	L_3	180	L_3	28	L_3	L_3
John	L_3	180	L_3	28	L_3	L_2
John	L_3	180	L_3	28	L_3	L_1

Figure 37: A Customer Relation

As we can see in the relation, the three records express the exact same real world thing, a customer *John*. However, the polyinstantiation integrity rule splits data to different security levels. This approach, of course, increases the data stored in database, resulting in a data redundancy problem. The MLR introduced the borrow concept and stores pointers, but not real data, in the higher-level to deal with this problem. However, even in that model, we still need to store three records. The BCMLS model can solve the problem only if the redundancy of all attributes in an entity belongs to the same security level. Otherwise, the redundancy cannot be reduced. For example, data in Figure 37 can be stored as one record in BCMLS model as shown in Figure 38.

The BCMLS model can reduce redundant data in Figure 37 because all attributes of each record have the same security level. However, returning to the data shown in Figure 35, although the first two records differ in the values of only one attribute, *home phone*, the system still has to store both records with all other information as well. So, the BCMLS model still cannot address the problem.

Name	Weight	Age	TC
John $L_1L_2L_3$	180 $L_1L_2L_3$	28 $L_1L_2L_3$	$L_1L_2L_3$

Figure 38: A Customer Relation in the BCMLS Model

6.2.2 Null Value Inference Problem

The second problem is inference when dealing with sensitive data in multilevel security models. A covert channel in a database is a means by which one can infer data classified at a high level from data classified at a low level. The inference problem is the problem of detecting and removing covert channels. An inference of sensitive data from nonsensitive data can only be represented within a database if the nonsensitive data itself is stored in the database. For example, if we have the set of data shown in Figure 39(a), a query from L_3 level user may result in null values, as shown in Figure 39(b). The null values generated may result in security risks of inference. The lower level user may infer that there is a value for *John* in *Weight data* attribute that is not accessible.

Name		Weight		Age	
James	L_3	180	L_3	32	L_3
John	L_3	225	L_2	28	L_3

(a)

Name		Weight		Age	
James	L_3	180	L_3	32	L_3
John	L_3	Null		28	L_3

(b)

Figure 39: The Null Value Inference Problem

Name		Weight		Age		TC
James	L_3	180	L_3	32	L_3	L_3
John	L_3	225	L_2	28	L_3	L_2
John	L_3	220	L_3	28	L_3	L_3

Figure 40: Data Redundancy Problem

And in some cases, L_3 users can even breach the security system and obtain the sensitive information by using some statistics queries [1].

The null value inference problem can be reduced if tuple-level labels are added and each tuple-level label is set to at least the highest security level of its components. However, since lower-level users cannot access tuples with higher tuple-level labels, additional tuples have to be created for the lower-level users to access. In Figure 40, a tuple needs to be created for L_3 level users. In general, for each lower level, we may need to add a tuple. However, this will result in the data redundancy problem as previously described.

6.2.3 Sensitive Key Value Problem

In the traditional multilevel security model, the polyinstantiation integrity rule, intended to protect sensitive data from lower level users, allows only non-key attributes to have different values at different security levels. However, these models leave a problem unsolved: if the sensitive data is in the key attribute(s), how should the model deal with it? Since the relational model uses key attribute(s) to indicate records (instances), when sensitive data is included in the key attribute(s), the polyinstantiation integrity rule can be relaxed and therefore cannot protect the sensitive data. Figure 41 demonstrates this problem. In the *Person* table, the attribute *Name* is a key attribute. Three records are assigned to different security levels. In the first record, the value *James* in *Name* attribute is assigned to L_3 level. In the second record, the value is still *James* for the L_2 level; however, in the third record, the value for *Name* attribute has changed to *John* and is assigned to the L_1 level. This will not be a problem if the records contain data of two real world people. However, in this case, the three records express one real world thing, possibly a government agent, whose real name is *John*. In this case, the highest level's users (L_1 level users) will access all three records as in Figure 41. However, without further information they would not recognize that the first two records are only

Person

Name		Weight		Age		TC
James	L_3	160	L_3	32	L_3	L_3
James	L_2	170	L_2	30	L_3	L_2
John	L_1	170	L_2	31	L_1	L_1

Figure 41: The Sensitive Key Value Problem

masks, which protect the last record from lower level users.

The difficulties with multilevel security models arise from the basic concepts of class-based models as indicated in [2], rather than multilevel security models themselves. In a class-based database, a schema is the global view of the data in the database. The schema presents a closed world for any data in the database. Because of the existence of such schema, any data (and/or users) have to be assigned to a certain (global) security level; this is where problems arise. Since the designer has to decide security levels in advance to match the schema, a number of questions also arise: How many security levels will be sufficient for the application? Should users with the same security level but different needs access the same portion of the dataset? For example, does the sensitive information in a security department share the same security level as an academic department? Users also encounter very similar problems: Do all users in the sensitive level share identical authorization to access the data? How can administrators modify the authorizations of users after the system is built? If a user is initially assigned with medium security level, how can we allow this person to access sensitive data beyond his/her security level?

Based on its theoretical principles, the multilevel security model should be a reliable and convenient method for protection of sensitive information. However, the basic concepts of class-based models have limited the applications of this security control methodology, hence its success.

In the next chapter we propose a new multilevel security model, the instance-based multilevel security model (IBMSM), based on principles of the instance-based data model, ontology, and basic security theory, to solve current problems associated with class-based models.

Chapter 7

The Instance-based Multilevel Security Model (IBMSM)

In this chapter we propose a new security model, the instance-based multilevel security model (IBMSM), based on the instance-based data model. The research will formally define the IBMSM. We will also prove that the model is secure.

In the following sections, the problems raised by multilevel security control models will first be addressed. Several concepts based on the instance-based data model will be introduced. Finally, the instance-based multilevel security model will be formally defined. The work also includes an extension of SQL operations to the model.

7.1 IBMSM

People recognize a thing by recognizing its properties. However, since the knowledge that different people have about the real-world thing is different, the level of understanding, or level of recognition, of users is different as well. For example, a young child may only describe the sun as a bright sphere. However, an astronomer will characterize the sun in much more detail, including properties such as its mass, temperature and photospheric composition. In the multilevel security model, the abilities for users to recognize an object in terms of its properties are directly related to their security levels. In the context of the above example, the astronomer can be considered to be in a higher security level than the young child. To recognize this hypothesis, we propose the following about an instance's views:

Proposition 7.1 (Property Views) Users in different security levels recognize an object by recognizing its properties. Different levels of users have different capabilities to identify a property; hence there are different views of the same real property. We call these views of the real property.

In Proposition 7.1, we assume that all users in the same security level have the same capability to identify a property. However, they may not have the same interests in the objects. For example, in a company a technical manager may be interested in objects related to the technical area; however, a business manager may not have interests in technical issues but in promoting and merchandising company products. So, we offer another proposition to deal with this situation.

Proposition 7.2 (Class View) Different users may be interested in different sets of instances; each set of instances could be recognized as a class, which expresses all the common aspects of the instances.

Following the above propositions, the instance-based multilevel security model consists of three parts, *the instance*, *the class* and *the control models*. The definitions of the three parts are:

Definition 7.1*:* A view of an instance at a security level L_j is denoted by $\boldsymbol{i}$ $\{(P_i, L_j) \mid P_i \in \mathbf{P}$ and $L_j \in \mathbf{L}\}$, where $\boldsymbol{i}$ is an instance identifier, P_i is a view of a property over the set of all properties (which is **P)**, and L_j is a security level over the domain **L**. A pair (P_i, L_j) indicates that a property's view, P_i, belongs to the security level L_j.

An instance may have a different view in different security levels. For example, an instance *instance1{(Name James,* L_3*), (Weight 160,* L_2*), (Weight 160,* L_3*), (Age 30,* L_1*), (Age 32,* L_2*)}*, where $L_1 > L_2 > L_3$ (in this chapter and the following chapter we will always assume that $L_1 > L_2 > L_3$), has a view at the L_2 level as following:

instance1{(Name James, L_3*), (Weight 160,* L_2*), (Weight 160,* L_3*), (Age 32,* L_2*)}*

However, in the L_3 level, the instance will have view as following:

instance1{(Name James, L_3*), (Weight 160,* L_3*)}*

As we can see, since the lower level user cannot access the higher level data, the instance view at the L_3 level includes less information than at the L_2 level.

Note that a higher level user can see lower level data in the Definition 6.1, which is the same as traditional multilevel security models (for example, the MLR model). In the IBMSM, an instance may not have one view of a property at high security levels but several views of the same property at lower levels. However, in traditional multilevel security models, if a higher level record needs to access values of a lower level record, it has to borrow a value from a certain level record before higher level records can be inserted into a database (we call it the belief-based assumption in traditional multilevel security models). We will discuss the problem of the belief-based assumption in the next section.

Definition 7.2: A class is denoted by Class_ID ($\{P_i\}$, $\{U_i\}$), where Class_ID is a class identifier, $\{P_i\}$ is a subset of properties of all properties (which is **P**), and $\{U_i\}$ is a subset of user identifiers over all the system.

A class contains two pieces of information. First, it includes information about which instances should be included in the class. Second, it includes information about which users can access this class. For example, if we define a class *Class1({Name, Age}, {user1, user3})* then an instance *Instance1{(Name James), (Weight 170), (Age 30)}* belongs to the class. However, an instance *Instance2{(Name John), (Weight 170)}* does not belong to the class. Meanwhile, user1 and user3 can access this class. However, other users cannot access this class.

Definition 7.3: A view of an instance at a certain security level L_j, which is $\boldsymbol{i}$ $\{(P_i, L_q) \mid P_i \in \mathbf{P}, L_q \leq L_j, \text{ and } L_q, L_j \in \mathbf{L}\}$, belongs to a class $C(\{P_k\}, \{U_i\})$ if and only if $\{P_k\}$ is a subset of $\{P_i\}$. A user U can access a class $C(\{P_k\}, \{U_i\})$ if and only if $U \in \{U_i\}$.

As we already indicated, an instance may have different views at different security levels. Thus, an instance may belong to different classes at different security levels. For example, if we define two classes, *Class1({Name, Age}, {user1, user3})* and *Class2({Name, Weight}, {user2, user3})*, then an instance, *Instance1{(Name James,* L_3*), (Weight 160,* L_2*), (Weight 160,* L_3*), (Age 30,* L_1*), (Age 32,* L_2*)}*, belongs to Class1 and Class2 at the L_1 and L_2 levels. However, it only belongs to Class2 at the L_3 level. At the

L_3 level, the instance only has a view *Instance1{(Name James,* L_3*), (Weight 160,* L_3*)}*. Of course, at this level it does not have the set of properties *{Name, Age},* which is in Class1's definition.

To enhance system security, we propose a rule for the IBMSM model originating from the two Bell-LaPadula restrictions for the instance-layer.

Rule 7.1: A user U at a certain security level L (designated as U_L) can read a property (which is a view of the property) of an instance at a security level L_j (we express this property as P_{L_j}) if and only if $L \geq L_j$. However, U_L can update P_{L_j} if and only if $L=L_j$.

Rule 7.1 indicates that the data that users at a certain level security can read consists of two portions: One is the data in the same security level as that of users, and the other is the data in the security levels lower than that of users. The latter can be updated by lower-level users who have the same security level as the records. In other words, a user can update the data in the same security level as itself (the user); it cannot update data in any lower security levels even though it can read them.

7.2 Data Interpretation in IBMSM

For all instances i $\{P_i \mid P_i \in \mathbf{P}\}$, in the IBMSM, data are interpreted in two parts, the instance part and the property part. We describe each of them as follows:

7.2.1 Property P_i and its Security Level L_j

An instance possessing a property view, P_i, at a security level, L_j, is denoted as a pair, (P_i, L_j), as in *Definition 7.1*. However, since an instance may possess views of the same property in more than one security levels, (P_i, {L_j}) is used to denote more than one pair, (P_i, L_j), for instance (P_i, L_1), (P_i, L_2) … (P_i, L_{10}). Conversely, if an instance possesses more than one property view of different real properties in the same security level L, they are denoted as ({P_i}, L).

7.2.2 An Instance's View and its Security Levels

An instance identifier *i* identifies an instance in the database. *i*(L) denotes that an instance *i* possesses some properties at security level L. To represent that an instance possesses properties that belong to more than one security level (i.e. an instance *i* which possesses some properties in security levels $\{L_j\} \in \mathbf{L}$) the notation *i*({L_j}) is used.

If an instance possesses a property at a security level L, the notation (P_i, L) represents that an L level user has created a property, P_i, of the instance.

Instances *p* and *q* are identical at a security level L if and only if they have the same view at the security level. That is, if for all P_i, (P_i, L) is a view of a property possessed by instance *p* if and only if (P_i, L) also is a view of a property possessed by instance *q*.

For example, two instances, *instance 1{(Name James,* L_3*), (Weight 160,* L_2*), (Weight 160,* L_3*), (Age 30,* L_1*), (Age 32,* L_2*)}* and *instance 2{(Name James,* L_3*), (Name John,* L_2*),* *(Weight 160,* L_3*), (Age 32,* L_1*), (Age 30,* L_2*)}*, are identical at the L_3 level since they have the same property view, *{(Name James,* L_3*), (Weight 160,* L_3*)}*, at that security level.

Notice that the semantics of data in the IBMSM model is different from the class-based model. In particular:

1) In the IBMSM, the requirement that an instance should belong to any class (schema) is eliminated. So, the greatest lower bound to define a view of an instance at any security level, common in all class-based multilevel security models, is unnecessary. Eliminating this assumption in the IBMSM will enhance the security of the model (we discuss this further in the next section). For example, the null value problem does not exist in the IBMSM.

2) An instance's views at different security levels may belong to different non-hierarchical classes. In the IBMSM model, users recognize an instance by recognizing its properties. Users at different security levels have different abilities to recognize properties of an instance. Since a higher-level user can access any lower-level data, the higher-level users may recognize that an instance belongs to a class that the lower-level user may not recognize. For example, we define an *overweight* class: a person (which is an instance) is overweight if his weight is more than 300lb. Then an instance *{(Name James)* L_3*, (Weight 280lb)* L_3*, (Weight 305lb)* L_2*, (Age 21)* L_3*}* belongs to the *overweight* class at the L_2 level since its *Weight* is *305lb* at the L_2 level. However, it does not belong to the *overweight* class at the L_3 level since the L_3 level users only recognize its *Weight* as *280lb*.

3) The absence of a property of an instance for users at a security level means that this property is not present at the security level. However, the absence does not reflect the rejection of this property. Users in the security level may define the property later. For example, in the relational model any instance must belong to a table. We have to use *null* to indicate a value of an attribute of the instance if the value of the attribute is missing or the instance does not have this attribute. However, the *null* value could be confusing (this is the closed-world problem). On one hand, the *null* value itself tells that we do not know whether the instance has this attribute. On the other hand, the schema indicates that any record in a table must have the same set of attributes. In the instance-based data model, this problem is solved. The instance-based data model uses the open-world model to indicate an instance. If users recognize a property of the instance, they just add it to the instance.

4) There is no *belief-based* assumption in the IBMSM. In class-based security models, most recent models use a *belief* model to share data between higher levels and lower levels. For example, the MLR model utilizes the concept called *borrow* to allow sharing between high and low level records [76]. The BCMLS model has multiple labels to indicate whether or not higher level records can trust a lower level record [77]. However, whether a higher level record trusts lower level data has to be specified in advance of a query, not spontaneously with a query. This could cause the belief problem. Figure 42 illustrates this problem. Assume the three records refer to one object, a person James, in each relation. The actual age of James is *32*. In the beginning L_3 level users recognize that the age of James is 32, and the L_2 level users think the age of James is 30.

The L_1 level users *believe* the L_3 level users' view of the property, thus they inherit from it. Figure 42(a) illustrates these kinds of recognitions. Afterwards, the L_3 level users think that the real age of James was 30, whereas the L_2 level users realize that they were wrong in the beginning and update the age of James to 32, thus their initial views have switched; if the L_1 level users still believe the L_3 level view, thus continue to inherit from it, the proper view of the property *age* to the L_1 level user is lost. Figure 42 (b) shows the final views of the different security levels.

Name		Weight		Age		TC
James	L_3	160	L_3	32	L_3	L_3
James	L_3	170	L_2	30	L_2	L_2
James	L_3	170	L_2	32	L_3	L_1

(a)

Name		Weight		Age		TC
James	L_3	160	L_3	30	L_3	L_3
James	L_3	170	L_2	32	L_2	L_2
James	L_3	170	L_2	30	L_3	L_1

(b)

Figure 42: The Belief Problem

5) In class-based security models, an object could have several views in different security levels. For example, most class-based security models combine several key attributes of a table and a security level as the real key to identify records in the table. Since the key attributes identify objects in the relational model, it is possible that several

records (as many as there are security levels) could refer to one object. However, in IBMSM, any object is described by its instance identifier. An object only has one identifier however many security levels it might belong to. This is the biggest advantage of the IBMSM model compared to class-based security models. The model solves several problems caused by class-based models. We will discuss this issue in more detail in the next chapter.

7.3 Data Access and Integrity Rules

A database is a collection of related data. A database state is a collection of all instances of a database at a particular time. A secure database is a database in which the state of the database can only change from one secure state to another secure state. In this section we define data access and integrity rules to guarantee that data in an IBMSM database is secure and consistent.

Rule 7.2 (*Instance View Integrity*) An L level view of an instance *i*, which is *i*(L), can exist in an instance-based multilevel security database if and only if no identical view of another instance *j*, *j*(L), exists in the same level in the database.

Rule 7.2 guarantees that no duplicate objects exist in any level of a database. It guarantees the semantics of the instance identifier. Since an instance identifier includes all the features of the instance, and no instance should have completely the same features as any other instances in the real world, if a user discovers that any objects have completely duplicated properties, the objects are identical. The original idea of Rule 7.2

comes from the semantics of the instance identifier in the instance-based data model; however, in the security model, we extend the rule to each security level.

Rule 7.3 (*Property Integrity*) Instance $i\{(P_p, L_j) | P_p \in \mathbf{P}$ and $L_j \in \mathbf{L}\} \in \boldsymbol{i}$ satisfies property integrity if and only if for any pair of $(P_p, L) \in i$, $(P_q, L) \in i$, and $P_p \neq P_q$ the expression $P_p \cap P_q = \phi$ is always true.

Rule 7.3 states that a user at security level L can create a property P_p, which is (P_p, L), of an instance *i*, if and only if the instance does not have property P_p at the security level L and there is no other property of the instance *i* at the level L that is a compatible property of P_p. Rule 7.3 is also a rule extended from the instance-based data model (Rule 4.2 defined for the instance-based data model in Chapter 4) to the IBMSM.

Note that the property integrity rule only applies to properties in the same security level. If two properties of an instance belong to different security levels, then it does not matter whether they have an intersection.

Rule 7.4: A user at security level L can read a property P_i of an instance at security level L_j, which is the pair (P_i, L_j), if and only if $L \geq L_j$.

Rule 7.5: A user at the security level L can only create (or update) a property P_i of an instance at the security level L (not higher, not lower).

Rule 7.4 and Rule 7.5 extend the basic Bell-LaPadula rules, which are *No Read Up* and *No Write Down*, to the instance-based setting.

Rule 7.6: A user at security level L can read the instance identifier of an instance $i(\{L_j\})$ if and only if $L \geq L'$, where L' is the lowest level of $\{L_j\}$.

Note that an instance identifier is a symbol that humans use to indicate an instance. By Bunge's ontology, properties can only be possessed by things. The existence of properties depends on things. If a user can recognize a property of an instance, it should know which instance this property belongs to.

Rule 7.7: (*Association Integrity*) An association of two instances, $i\{(P_i, L_j) | P_i \in \mathbf{P}$ and $L_j \in \mathbf{L}\}$ and $i'\{(P_{i'}, L_{j'}) | P_{i'} \in \mathbf{P}$ and $L_{j'} \in \mathbf{L}\}$, at a certain security level L_{ij} exists only if

(a) *i* and *i'* exist in the database. That is $i \in \boldsymbol{i}$ and $i' \in \boldsymbol{i}$.

(b) The security level of the association, L_{ij}, should belong to both $\{L_j\} \in i$ and $\{L_{j'}\} \in i'$. That is, $L_{ij} \in \{L_j\} \cap \{L_{j'}\}$.

Rule 7.7 (a) follows the referential integrity of the instance-based data model introduced in the previous chapter, but (b) is new to the security model. Rule 7.7 (b) indicates that instances can be associated in a security level if and only if they both can be updated in that level. For example, assume there are two instances, instance *1 {(Name John,* L_2*), (Age 21,* L_1*), (Weight 120,* L_2*), (Sex M,* L_2*)}* and instance *2 {(Name Alice,* L_3*), (Age 20,* L_2*), (Sex F,* L_2*)},* in an IBMSM database. If the two instances associate together to form a higher level thing (e.g. they are married), such an association can only be formed at the L_2 level. Although instance 1 belongs to the L_1 level and users at the L_1 level can read instance 2, instance 2 does not belong to the L_1 level. Following Rule 7.4, users at the L_1 level cannot update any information about instance 2. Adding an association between instance 1 and instance 2 at the L_1 level means to update information of both instances at the L_1 level, so the operation is not allowed.

Rule 7.8: A user u, can access data through a class, $C(\{P_i\}, \{U_j\})$, if and only if $u \in \{U_j\}$.

Rule 7.8 represents the basic idea of the two-layered access control in the IBMSM that plays an important role in the security control. We discuss it in more detail in the next section.

In this chapter, we introduced eight data access and update rules for the IBMSM. Some rules are inherited from previous multilevel security models. For example, Rule 7.4 and Rule 7.5 come from the Bell-LaPadula model. However, some of the rules are unique to the IBMSM. For example, Rule 7.8 is used for the two-layered security control. In combination, these rules guarantee the consistency and security of the model (a proof of this claim is provided in Section 7.6). We will show how the rules can be applied in different operations in the next sections.

The rules proposed in this section are extended directly from the instance-based data model. The original IBDM did not have these rules [2]. Instead, earlier work focused on how to present the query ability and the flexibility of the model rather than on whether rules are needed to ensure that data remain consistent. Even when a prototype system was implemented based on the IBDM [70], the rules were not all included. After proposing the instance view integrity rule and the property integrity rule for the IBMSM (the first two rules of IBMSM), we found that the instance-based data model itself needed these rules to make the model consistent, even though the rules in the instance-based data model may not be as strict as in the IBMSM. So, we added a section in Chapter 4

(Section 4.6) to discuss integration rules in the instance-based data model. We have proposed rules in Section 4.6 and in this section in conjunction. Finally, we have rules for the instance-based data model to improve the consistency and reduce possible redundancy of data in the model. We also have rules for the IBMSM to guarantee the consistency and security of the model. Since the IBMSM is based on the instance-based data model, when we describe the rules in IBMSM, we indicate that some rules are extended directly from the instance-based data model. But most rules in IBMSM were developed in order to ensure a secure model and were not part of the original IBDM specification.

7.4 Two-Layered Access Control

The instance-based multilevel security model uses two layers to control access to data: the instance layer and the class layer. In the model, access is first controlled by the class layer. The class layer governs the range of objects accessible to a particular user. The second access is controlled by the instance layer, which controls the accessibility of the sensitive data in a class to a user.

Figure 43 in the next page illustrates such control in the two-layered approach. The dataset constitutes the instance layer; the classes represent the class layer. The original dataset is divided into different sections, and each section contains both sensitive and non-sensitive data. To access the data in each section, either non-sensitive or sensitive, users have to be able to access different classes first. As shown in the Figure,

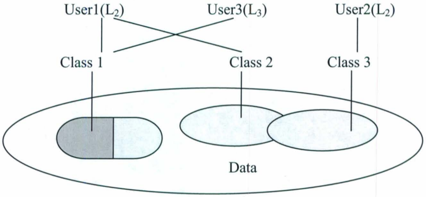


Figure 43: Two-layered Controls (Darker color indicates L_2 level data, grey color indicates L_3 level data)

in order to access the dark section (sensitive data), users have to be authorized to access classes first. Both User 1 and User 3 may access data in the dark data section since they both have ability to access Class 1; however, User 2 cannot do so since it does not have authorization to access this class. Users' abilities to access the same class does not necessarily mean that they share the same ability to access the data in the data-section through this class. For instance, a L_3 level user, User 3, can only access the L_3 level data but not the L_2 portion, even if it is authorized to access the same classes as the L_2 level user, User 1.

The SeaView and its derivatives use views to control access, as well. However, the two-layered security approach in the instance-based multilevel security model is different from the original multilevel security models in class-based models. For example, in the MLR model, even if a real object has several records in different security levels, they belong to the same class. However, in the IBMSM, an instance may belong to different classes on different security levels. For example, an instance *1{(Name John,* L_3*),* *(StudentID 2000001,* L_2*), (Birthday 93/05/06,* L_1*)}* is a student (defined by *student{StudentID}*) on L_2 level, but it does not belong to the class student on L_3 level. This approach increases the security of the IBMSM (we will discuss this in greater detail in the next chapter).

7.5 Operations

7.5.1 Insertion

Insert an instance

The syntax to insert an instance, issued by a L level user, is the same as the instance-based data model as follows:

Insert Instance ins_ID $(p_1[, p_2, ..., p_n])$ (1)

In (1), Insert and Instance are key words which indicate the insert instance operation. The ins_ID is an instance identifier. $p_1, p_2, \ldots, p_n$ are properties. [ ] is used to indicate optional elements and '…' indicates repetition.

Each Insert Instance command can insert at most one instance into a database. The inserted instance is constructed as follows:

(a) If ins_ID is not in the database, the insertion will insert the instance, *ins_ID*$\{(P_i, L) | P_i \in \{p_1[, p_2, ..., p_n]\}$ *and* $L \in \mathbf{L}\}$, into the database.

(b) If *ins_ID* is in the database, the insertion will add the property pairs, $\{(P_i, L) | P_i \in \{p_1[, p_2, ..., p_n]\}$ *and* $L \in \mathbf{L}\}$, into the instance *ins_ID*.

There are two constraints to restrict the insertion of an instance. The instance, *i,* can be inserted into the database if and only if after insertion

(a) No properties P_i and P_j exist such that $(P_i, L) \in i$ and $(P_j, L) \in i$, and $P_i \cap P_j \neq \phi$

(b) No instance j exists for which $(\{P_i\}, L) \in i(L)$ equals $(\{P_j\}, L) \in j(L)$ for any $L \in \mathbf{L}$

For example, if there are some instances in a database including an instance, *instance1 {(Height 200,* L_3*), (Weight 180,* L_3*), (Color red,* L_1*)}*, then a user at L_3 level can insert an instance, *instance2 {(Height 200,* L_3*), (Weight 180,* L_3*), (Color red,* L_3*)}*, into the database. However, the user cannot insert an instance, *instance3 {(Height 200,* L_3*), (Weight 180,* L_3*)}*, into the database, since instance1 and instance3 have the same property set in the security level L_3.

The first constraint indicates that the inserted instance must satisfy the property view integrity rule, while the second ensures that the resulting database satisfies the instance view integrity rule.

Insert a mutual property

The syntax of the insertion of a mutual property issued by L level user, U_L, is the same as the instance-based data model, as follows:

Insert Mutualproperty mp_ID shared by insID1, insID2[, ...] (2)

In (2), *Insert* and *Mutualproperty* are key words used to indicate the insertion of a mutual property. The mp_ID is a mutual property identifier. The sequence insID1, insID2[, …] are instance identifiers.

Similar to the *Insert Instance* operation, each *Insert Mutualproperty* command inserts at most one mutual property into a database. The inserted mutual property is constructed as follows:

(a) If the sequence *insID1, insID2[, ...],* which indicates the concept of combined instance, is not in the database, the insertion will insert the concept of combined instance identifier, which is the sequence *insID1, insID2[, ...],* followed by the pairs $((mp\_ID, L) \mid mp\_ID \in \boldsymbol{MP} \text{ and } L \in \boldsymbol{L})$..

(b) If the combined instance is in the database, the insertion will add the pairs, $((mp\_ID, L) \mid mp\_ID \in \boldsymbol{MP} \text{ and } L \in \boldsymbol{L})$, into property set of the combined instance.

From the above method, construction of a mutual property is like construction of an instance. However, more constraints are applied to it:

(a) The combined instance can be inserted into the database if and only if for any $i \in \{insID1, insID2[, ...]\}$, $i(L) \in \boldsymbol{i}$

(b) After the insertion, for any two property pairs (MP_i, L) and (MP_j, L) of the combined instance, $MP_i \cap MP_j = \phi$

The above two insertion operations are in the instance level. They operate on the basic data of the database. An insertion operation on the class level is also proposed, designated by the command *insert class* and is intended for system administrators.

Insert a class

The syntax of the *insert class* issued by a system administrator in the instance-based security model as follows:

Insert Class Class_ID $(\{P_1[, P_2, \ldots, P_n]\}, \{U_1[, U_2, \ldots]\})$ (3)

In (3), *Insert* and *Class* are key words used to indicate the insertion of a class. *Class_ID* is a class identifier. $P_1, P_2, \ldots, P_n$ are properties, and $U_1, U_2, \ldots$ are user identifiers.

Each *Insert Class* command will insert one class into a database. The inserted class is constructed as follows:

(a) If *Class_ID* is not in the database, the operation will insert the class, *Class_ID*$\{(\{P_i\}, \{U_j\}) \mid P_i \in \{P_1[,P_2, \ldots, P_n]\}$ *and* $U_j \in \{U_1[,U_2, \ldots]\}$.

(b) If *Class_ID* is in the database, the operation will update the class with the new definition: *Class_ID*$\{(\{P_i\}, \{U_j\}) \mid P_i \in \{P_1[,P_2, \ldots, P_n]\}$ *and* $U_j \in \{U_1[,U_2, \ldots]\}$.

Several restrictions have been placed to restrict the operation of *Insert Class*; a class *c* can be inserted into the database if and only if:

(a) For any property, P_i, if P_i is in the definition of class *c* then $P_i \in \mathbf{P}$.

(b) For any two properties P_i and P_j, $P_i \in c$ and $P_j \in c$, there is no precedence between P_i and P_j such that $P_i \longrightarrow P_j$ or $P_j \longrightarrow P_i$. (Here the arrow $\longrightarrow$ indicates preceding relation between properties [57].)

(c) For any $U \in \{U_j\}$, $U \in \mathbf{U}$.

Restriction (a) indicates that any property in the inserted class must be in the instance layer. By Bunge's ontology, a class is a set of instances that share some properties. If no instance possesses a property, then, of course, no class will exist. Restriction (b) indicates that any inserted class should have as few properties as possible in its definition. One property preceding another means they are possessed by the same set of instances. Restriction (b) eliminates this possibility which reduces the number of properties in the class definition. The restriction (c) indicates that only existing users can access a class.

7.5.2 Deletion

Delete an instance

The syntax of the deletion of an instance issued by an S level user is very similar to the operation of insertion. The basics of the operation on the instance-based data model are as follows:

Delete Instance
From c
[Where P_p *[%]]* (4)

Similar to the insertion operation, *Delete* and *Instance* in (4) are key words to indicate deletion of an instance. The *%* sign indicates any lower (lower than L) security levels of the property, P_p, that need to be considered as a condition of the command. The *From* and *Where* clauses are conditions added for the deletion, which have the same semantics as their definitions in the relational model.

The semantics of command (4) is implemented as follows:

For any instance *i*, a user at security level *l* issuing a *Delete Instance* command will result in the following:

(a) if the instance, *i*, does not have any property view at *l* level or the instance does not exist in the database, then no instance or instance view will be deleted.

(b) If *i* is only in the security level *l*, that is the operation will delete all properties of *i* and the instance identifier, *i*, itself.

(c) If *i* belongs to more than one security levels, that is, then the *Delete Instance* command issued by *l* level user only deletes the properties of the instance, *i*, in *l* level. Other security levels' properties and the instance identifier of the instance are preserved.

An *l*-level user issuing a *Delete Instance* operation must meet two requirements before the command can be executed. For the deletion of the instance *i*, selected by clause *From* and *Where* conditions:

(a) The user must be able to access the class c.

(b) The L level instance does not participate in a mutual property.

Delete a mutual property

The syntax of the deletion of a mutual property issued by an *l* level user is defined as follows:

Delete Mutualproperty mp_ID [shared by instance insID$_1$, insID$_2$, ...] (5)

In (5), *Delete* and *Mutualproperty* are key words used to indicate the deletion. *mp_ID* is a mutual property identifier. The sequence *insID1, insID2, ...* are instance identifiers.

The semantics of the command (5) are similar to the *Delete Instance* command. The implementation of (5) is as follows:

for the mutual property, *mp_ID*,

(a) If *mp_ID* is only in the security level *l*, then the operation will delete the association *(mp_ID)* between the sequence $insID_1$, $insID_2$, ... from the database.

(b) If *mp_ID* belongs to more than one security level, then the operation issued by *l* level user only deletes the *l* level association between each instance of the sequence. The associations on other security levels will still be preserved.

The requirements of an L level user to issue a *Delete Mutualproperty* command follow the rules of the security control:

(1) The mutual property, *mp_ID*, must possess a security level view equal to the security level L.

(2) The user must have the ability to access each instance member in the sequence $insID_1$, $insID_2$, ... through some classes.

Delete a class

In the instance-based security model, instances are stored separately from classes. Classes serve as security control functions most of the time. So, class deletion, with syntax as *delete class className*, issued only by system administrators in the model, is very easy to implement. The system simply deletes the class and no condition needs to be met for the operation.

7.5.3 Select

The syntax of the select operation issued by an *l* level user, u, is as follows:

Select P_1*[[%],* P_2*...]*
From c_1 *[,* c_2*, ...]*
[Where P_a *[%]]*
[Sharing mp [%]] (6)

In (6) P_1, P_2... and P_a are properties, c_1*[,* c_2*, ...]* are classes, *mp* is a mutual property, and the *%* sign indicates to include all security levels lower than user u's security level *l*.

The semantics of the *Select* statement is that only the instances accessible from the class c_1 to the user u and possessing properties P_1*[,* P_2*...]* in *l* level (or lower if there is a *%* sign) will be present in the results. If there are more than one class in the *From* clause, whenever the user u accesses instances from these classes, the query results will be presented. The *Where* and the *Sharing* clauses place restrictions only on the final results, which is identical to the IBDM model.

There may be a *%* sign followed by a property in both *Select* and *Where* clauses in the IBMSM model when a query is issued as shown in (6). The sign indicates the biggest difference between class-based security models and the IBMSM model. As we have discussed, most of the multilevel security models are belief-based models. For example, the MLR model uses *borrow* to allow a higher level record to share some information from a lower level record. However, their belief model is based on *pre-belief*, which means that whether a higher level record believes some information of lower level records is decided before the higher level record is added to a database. We already discussed the problems of *pre-belief* in section 6.2. The IBMSM does not use the *pre-belief* model. It uses a *post-belief* model, which means that whether a higher level user believes a lower level information is determined at the time he/she issues a query. In the model, we use the *%* sign to indicate which level of information the higher level user wants to query or use the *%* sign to add some conditions which are in lower levels (in the *Where* clause or *Sharing* clause). For example, an *l* level user may issue a query:

Select Name L_3 *From student* (7)

This query will give the *l* level user all the student names at the L_3 level.

The *Select* statement is very flexible in the instance-based security model. Users can query almost anything and the statement will always return results (even if the result is empty).

A user may declare a query as:

Select P_1 *From c* (8)

The user may issue query (8) even if P_1 is not in the definition of class c. However, in general, we assume that P_1 is, at least, a property preceded [57] by properties of c to make more sense of the queries. This restriction is useful in the security model. For example, if a class student is defined as enrolling a university, then a user may issue a query:

Select Name, StudentID From student (9)

Query (9) will be fine in the IBMSM model since any student should have a name and student number. So, the properties *Name* and StudentID precede enrolling a university. However, if a user issues a query:

Select Name, Salary From student (10)

Query (10) makes no sense if students do not have a salary (as students).

The restrictions on Query (8) are also important in the IBMSM. As we already discussed, the IBMSM is based on the IBDM. In the IBDM, an instance may belong to more than one class. For example, an instance *Instance1{Name James, Age 30, Weight 380, Salary 4800}* might belong to two classes: *Overweight{Name, Weight>300}* and *Employee{Name, Salary}*. However, in the IBMSM, the ability for a user to access an instance is controlled by two-layers: the class layer and the instance layer. Assume a user is a doctor and can access patient information through the *Overweight* class. However, he cannot access any employee information. If we do not have the restriction, he may issue a query:

Select Name, Salary From Overweight (11)

to get some information about his patient's employment, which we suppose he should not have the need or capability to access.

After we introduce the restrictions above, Query (11) is not allowed since *Salary* is not preceded by *Overweight* class.

7.6 Security

Since the IBMSM is based on the instance-based theory, it is fundamentally different from the traditional class-based models. In this section, we will prove that the IBMSM is a secure model, which means there is no covert channel between any security levels.

As we have discussed, IBMSM utilizes a two-layer approach to control access, the class and the instance layers. In the class layer, the accessibility of a class to a user is not determined by a user's security level. Users at different security levels may access the same class (the definition of the class). Also, sensitive information is not accessible by users in the class layer. Any data in the class layer is maintained by system administrators. A user may query a class definition if he/she can access the class. Therefore, the formation of covert channels is essentially impossible at the class layer. The following proof will focus on the instance level.

The following notations will be used:

U: the set of all users with varying security levels.

I: the set of all instances with varying views in all security levels.

P: a property with varying security level views in an instance.

For a certain security level *l*,

UL(*l*): the set of users with security levels lower than or equal to *l*.

UH(*l*): the set of users with security levels higher than *l*.

IL(*l*): a set of views of instances with the security level *l* or lower.

IH(*l*): a set of views of instances with the security level higher than *l*.

PL(*l*): a set of views of properties of instances with the security level equal to or lower than *l*.

PH(*l*): a set of views of properties of instances with the security level higher than *l*.

From the above notations, six equations can be obtained:

$\mathrm{UL}(l) \cup \mathrm{UH}(l)=\mathrm{U}$ (1)

$UL(l) \cap UH(l) = \phi$ (2)

Equations 1 and 2 mean that: all users are in a security level either higher than *l* level or lower (or equal to) than *l* level, and no user is in a security level both higher than *l* level and lower (or equal to) than *l* level at the same time.

$IL(l) \cup IH(l) = I$ (3)

$IL(l) \cap IH(l) = \phi$ (4)

The equations 3 and 4 mean that: all views of instances have their security level either higher than *l* level or lower (or equal to) than *l* level, and no view of any instance has its security level both higher than *l* level and lower (or equal to) than *l* level.

$PL(l) \cup PH(l) = P$ (5)

$PL(l) \cap PH(l) = \phi$ (6)

The equations 5 and 6 mean that all views of properties are in a security level either higher than *l* level or lower (or equal to) than *l* level, and no view of any property is in a security level both higher than *l* level and lower (or equal to) than *l* level.

Note that in IBMSM, if an instance *i* is in *l* security level, *i(l)*, this means that it has at least one view of at least one property of the instance *i* at the security level *l*.

A database is collection of values of data. A database state is the collection of data in database at a particular time. A secure state of a database is a state in which data can only be accessed by following Bell-LaPadula's rules (*No Read Up* and *No Write Down*). A secure database is a database in which the state of the database can only change from one secure state to another secure state.

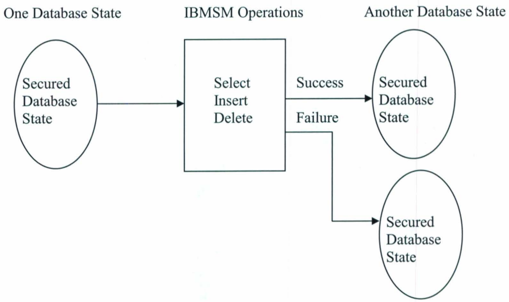


Figure 44: IBMSM Database State Transformation

A secure data model is a database model that takes a database from one secure state, through a number of operations, to another secure state. Figure 44 illustrates this idea. Goguen and Meseguer suggest a noninterfering security data model [81]. In this model, they define several concepts for security data models. We will also use their definition for a secure database. For any security level *l*, a command to delete any data issued by users at a higher security level does not affect the view of data to any user at the lower security levels.

Theorem 7.1: The IBMSM is a secure data model.

As shown in Figure 44, a database is modified from the initial to the final state by a series of user operations. To prove that IBMSM can only go from one secure state to another secure state through a sequence of operations, we only need to prove that IBMSM operations are secure, since those are the only operations allowed.

In IBMSM, all database operations are issued by users on a certain security level. First, we will prove any security level database operation will not affect (here affect means increase or reduce information) at another security level. For example, a higher security level user operating on data should not affect any lower security level users. Also, a lower security level user operating on data should not affect any higher security level users. Thus, a direct path from neither left to right nor right to left is possible, in Figure 45.

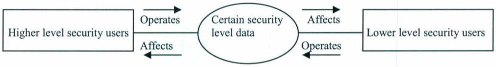

Higher level security users
Operates
Affects
Certain security level data
Affects
Operates
Lower level security users

Figure 45: Different Security Levels of Users Affect Each Other

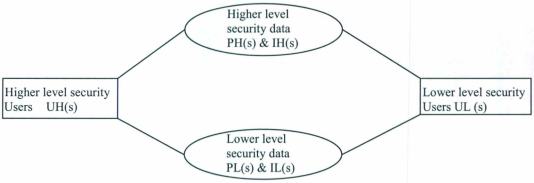

Higher level security data PH(s) & IH(s)
Higher level security Users UH(s)
Lower level security Users UL (s)
Lower level security data PL(s) & IL(s)

Figure 46: Users Affect Each Other on Different Level Data

In fact, a user at a certain security level, l, may operate on two types of data. One is the data at security levels equal to or lower than l. The other is the data at the security levels higher than l. So, whether users affect each others in Figure 45 can be expressed as Figure 46.

We will prove *Theorem 7.1* by following two steps:

To prove that any higher level data change will not affect any lower level user in IBMSM, we must first prove Lemma 7.1.

Lemma 7.1: For any security level l, changing data at higher level security views of instances, IH(l), will not affect any users $u \in UL(l)$.

Proof: A user u at security level l' can use several operations to access database, namely *Select*, *Insert*, and *Delete*. The user, u, at security level l', $l' \leq l$, means $u \in \mathrm{UL}(l)$.

For the *select* operation, the user u can access any view of an instance, $i(l') \in \mathrm{IL}(l')$, whose security level is equal or less than the security level l'. $\mathrm{IL}(l')$ is a set of views of instances that are in l' level. Only $\mathrm{p}(l')$ or lower security level properties of instances, or $PL(l')$, can be accessed by the user u. Views of instances at levels higher than l' level are not accessible by user u. Since $l' \leq l$, so $\mathrm{PH}(l') \supseteq \mathrm{PH}(l)$ and $\mathrm{IH}(l') \supseteq \mathrm{IH}(l)$. We already know that $\phi = \mathrm{IL}(l') \cap \mathrm{IH}(l')$ and $\phi = \mathrm{PL}(l') \cap \mathrm{PH}(l')$. So, $\phi = \mathrm{IL}(l') \cap \mathrm{IH}(l)$ and $\phi = \mathrm{PL}(l') \cap \mathrm{PH}(l)$. That is, any changes in $\mathrm{PH}(l)$ and $\mathrm{IH}(l)$ will not affect $\mathrm{PL}(l')$ and $\mathrm{IL}(l')$. Therefore, changes of $\mathrm{PH}(l)$ and $\mathrm{IH}(l)$ do not affect l' level users $u \in \mathrm{UL}(l)$ with $l' \leq l$.

For *Insert* and *Delete* operations, any operation can be either a success or a failure. However, since IBMSM only allows u to operate on the data with these operations at its own security level, successful operations only modify the data at the security level l'. Changes at other security levels of data other than l' will not affect the users at those levels at all. That is, changing PH(l) and IH(l) will not affect any l' level users $u \in UL(l)$ with $l' \leq l$.

Several factors can result in failures of operations:

(1) An *Insert Instance* command issued by a l' level user, u, to insert an instance i, could fail if and only if

(a) There are two properties P_i and P_j, which $(P_i, l') \in i$ and $(P_j, l') \in i$, and $P_i \cap P_j \neq \phi$

(b) There is an instance, instance j, which $i(l')=j(l')$ for any $l' \in \boldsymbol{L}$

(2) An *Insert Mutualproperty* command issued by a l' level user, u, to insert a mutual property of two instances, i and j, could fail if and only if

(a) $i(l') \notin \boldsymbol{i}$ or $j(l') \notin \boldsymbol{i}$

(b) There are two property pairs (MP_i, l') and (MP_j, l') of the combined instance, $MP_i \cap MP_j \neq \phi$

(3) A *Delete Instance* command issued by a l' level user, u, to delete an instance view at the l' level, $i(l')$, could fail if and only if a combined instance (mutual property) is formed by the instance at the l' level.

(4) A *Delete Mutualproperty* command issued by a l' level user, u, to delete a mutual property view at the l' level, $mp(l')$, could fail if and only if a combined instance (mutual property) is formed by the mutual property at the l' level.

All above situations are dealing with the l' level data (instance view, property view, etc). However, any data at the l' level, either $p(l')$, $mp(l')$, or $i(l')$, belongs to $PL(l')$ or

IL(*l*'). Since *l*'≤ *l*, PH(*l*') ⊇ PH(*l*) and IH(*l*') ⊇ IH(*l*). So, p(*l*'), mp(*l*') ∉ PH(*l*') ⊇ PH(*l*), and i(*l*') ∉ IH(*l*') ⊇ IH(*l*). That is, changes in PH(*l*) and IH(*l*) do not affect any *l*' level users u∈UL(*l*) with *l*'≤ *l*.

The proof of Lemma 7.1 is complete. □

Second, we are going to prove that any higher level user will not affect any lower level data in IBMSM. Using the same strategy, we obtain *Lemma 7.2*.

Lemma 7.2: For any security level l, higher level security user u, u∈UH(l), changing data does not affect any data PL(l) or IL(l).

Proof: A user may change data in two ways: insert data to or delete data from an IBMSM database.

(1) An *Insert Instance* command issued by the *l'* level (*l*'>*l*) user *u*, u∈UH(*l*), to insert an instance view *i*(*l*'), can only add a set of property views of the instance *i*, {p(*l*')}∈PH(*l*), to the database. Since *l*'>*l*, the added data {p(*l*')}∉ PL(*l*).

(2) An *Insert Mutualproperty* command issued by the *l'* level (*l*'>*l*) user *u*, u∈UH(*l*), to insert a *mutual property* view of two instances, *i* and *j*, can only add a mutual property view of the instances *i* and *j*, mp(*l*')∈PH(*l*), to the database. Since *l*'>*l*, the added data mp(*l*')∉ PL(*l*).

(3) A *Delete Instance* command issued by the *l'* level (*l*'>*l*) user u, *u∈UH(l),* to delete an instance view at the *l*' level, *i*(*l*'), can only delete a set of property views of the instance *i*, *{p(l')}∈PH(l),* from a database. Since *l*'>*l*, the deleted data *{p(l')}∉ PL(l).*

(4) A *Delete mutualproperty* command issued by the *l'* level user, *u*, to delete a mutual property view at the *l'* level, *mp(l')*, can only delete a mutual property view $mp(l') \in PH(l)$ of instances formed to a database. Since $l'>l$, the deleted mutual property $mp(l') \notin PL(l)$.

From the above operations, it becomes clear that if the *l'* level user, $l'>l$, changes any data it does not affect any data in *PL(l)* or *IL(l)*.

The proof of Lemma 7.2 is complete. □

From the above two lemmas, neither path from higher level users to lower level data nor higher level data to lower level users is applicable. Figure 47 shows the results of the two lemmas.

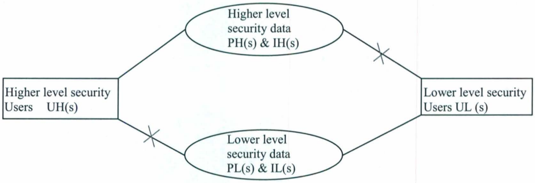

Higher level security data PH(s) & IH(s)
Higher level security Users UH(s)
Lower level security Users UL (s)
Lower level security data PL(s) & IL(s)

Figure 47: Affecting on Different Levels

Finally, *Theorem 7.1* can be proven.

Proof: As shown in Figure 47, since $U = UL(l) \cup UH(l)$ and $\phi = UL(l) \cap UH(l)$, no user is in between UL(*l*) and UH(*l*). Because $I = IL(l) \cup IH(l)$, $\phi = IL(l) \cap IH(l)$, $P = PL(l) \cup PH(l)$, *and* $\phi = PL(l) \cap PH(l)$, the intersection between two sets of data, *{PH(l) & IH(l)}* and *{PL(l) & IL(l)},* is empty, as well . Thus, no connection can be made between either different security levels of users or different security levels of data, which in turn proves that the IBMSM is a secure model.

The proof of Theorem 7.1 is complete. □

7.7 Structure and Implementation Methods of IBMSM

We have proposed the instance-based multilevel security model and proven that the model is a secure model. In this section, we will discuss a possible way to implement the model.

In chapter 5, we have shown that the instance-based database management system consists of three parts. However, the IBMSM system needs additional components as shown in Figure 48. These four categories are (from the bottom to the top of Figure 48): data storage, algorithm management, security management, SiQL language, and database management. The components of each category are as the following:

Stored Data: This component includes two parts. The first is the instance layer data, which includes instances, intrinsic properties and mutual properties, and security information for each intrinsic and mutual property, as well as the values of intrinsic and mutual properties for all instances. The data structure may vary according to the features of the instance-based data model. For example, in the instance layer the stored data of an IBMSM database may have a structure as shown in Figure 49, according to the second data structure in the instance-based data model. In this data structure, we maintain each intrinsic property as a list, consisting of the property identifier, followed by a set of three

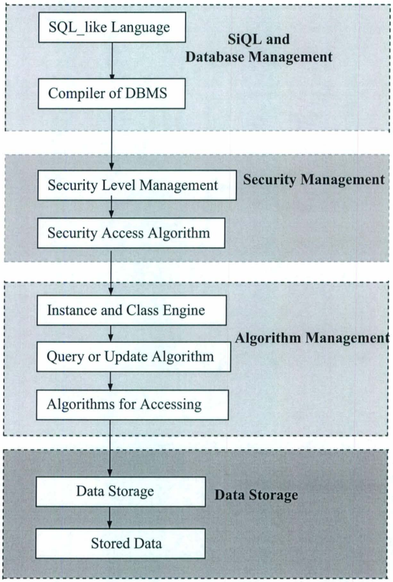


Figure 48: Architecture of the Instance-based Multilevel Security System

values: an instance identifier, the value of the property for this instance, and the security level. In the same way, a mutual property is maintained as a list of mutual property identifier followed by a set of four values[1]: first two are instance identifiers which jointly possess the mutual property, the third one is the value of the mutual property, and the last one is the security level of the mutual property. The second part is the class layer data. It includes class definitions (in term of properties), followed by a set of users who can access the class (Figure 49).

Mutual Property 1				**Property 1**			**Property 2**		
ID 1	ID 2	value	L_3	ID 1	value	L_2	ID 1	value	L_3
ID 3	ID 2	value	L_3	ID 2	value	L_3	ID 3	value	L_2
				ID 3	value	L_3			

The Instance Layer

Class 1
Property 1
Property 2
Mutual Property 1
Set of Users

The Class Layer

Figure 49: An Example Data Structure of IBMSM

Data Storage Methods: This area also includes two types of methods. The first methods are the methods used to store the instance layer information, (i.e., methods to store instances themselves, intrinsic properties, mutual properties and the security

[1] 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.

information of properties). The second are the methods 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, the algorithms can be categorized into two major types: methods to access instance layer information and to access the class layer. The first type includes an algorithm that specifies how to insert, delete, and retrieve instance information; an algorithm that specifies how to insert, delete, and retrieve intrinsic property information; and an algorithm that specifies how to insert, delete, and retrieve mutual property information. The second type includes algorithms to access class layer information. It includes inserting, deleting and retrieving the definition of classes. Each algorithm above must adhere to the rules we have defined for the instance-based data model in Chapter 4.

Query or Update Algorithm: This component includes the methods and algorithms to query or update operations. The algorithms need to follow the constraints that we have discussed in Section 7.5. For example, an insert of an instance operation should be denied if an instance has the same set of properties as the insert instance in a certain security level.

Instance and Class Engine: The instance and class engines are the same as the instance-based data model. The instance engine manages the instance layer. It creates a unique identifier for each instance, intrinsic property, and mutual property in the database. The class engine manages the class layer. It creates a unique class identifier for each class.

Security Access Algorithm: This component implements algorithms for the multilevel security control, for example the *No Read Up* and *No Write Down* algorithm. All rules we have discussed in Section 7.3 and Section 7.4 must also be implemented in this component.

Security Level Management: This component manages security levels in an IBMSM system, for example the number of levels in a database and/or the kinds of relations between two different levels will be managed by this component.

Compiler: It is used to compile SiQL commands.

SiQL language: This component uses the standard language of the IBMSM database system. It is a security extension of the iQL (instance-based Query Language) [70]. It supports a SQL-like query language as described in Section 7.5.

7.8 Summary

In this chapter, we proposed a new multilevel security model, IBMSM, according to instance-based concepts. The model uses two layers to ensure complete security control compared to the traditional multilevel security control methods, and we built several rules and operation methods for the model. After providing these rules, we proved that the model is a secure model. At the final section we discussed a possible way to implement the IBMSM system. The described method shows that the IBMSM system is implementable. In the next chapter, we will give some examples to illustrate that the IBMSM model solves several problems raised in traditional MLR models.

Chapter 8

Power and Security

In the last chapter we proposed the IBMSM. The IBMSM is more powerful and secure than traditional multilevel security models. In this chapter, we discuss how the instance-based multilevel security model addresses several unsolved problems under the traditional multilevel security models.

8.1 Duplicate Records Problem

First, when traditional models, for example the seaView model and the MLR model, deal with entity polyinstantiation or element polyinstantiation [82], we have to store information about the same real world thing in different security levels. The number of security levels a system has determines how many records are needed, even if only one real world thing is being represented. Entity polyinstantiation implies a relation that contains multiple tuples with the same key attribute values but with different security levels. Figure 50(a) illustrates entity polyinstantiation. In Figure 50(a), the first and the second records of the relation *Person* have the same key attribute value, *John*. However, they belong to different security levels. In the first record the value of name belongs to the L_2 level, but in the second record the value of name belongs to the L_3 level. Figure 50(b) illustrates element polyinstantiation. In Figure 50(b), the first record and the second record of the relation *Person* have the same key attribute value in the same security level, where name is *John* and the security level is L_3. However, the record itself belongs to the different security levels. The first record belongs to the L_2 level but the second record to the L_3 level.

Person

TC	<u>Name</u>	$Label_1$	Age	$Label_2$	Home Phone	$Label_3$
L_2	<u>John</u>	L_2	21	L_3	(709)737-1236	L_2
L_3	<u>John</u>	L_3	21	L_3	(709)737-1234	L_3
L_3	<u>Alice</u>	L_3	25	L_3	(709)781-4321	L_3

(a)

Person

TC	<u>Name</u>	$Label_1$	Age	$Label_2$	Home Phone	$Label_3$
L_2	<u>John</u>	L_3	21	L_3	709-737-1236	L_2
L_3	<u>John</u>	L_3	21	L_3	709-737-1234	L_3
L_3	<u>Alice</u>	L_3	25	L_3	709-781-4321	L_3

(b)

Figure 50: Entity Polyinstantiation and Element Polyinstantiation in MLR Model

As we have already seen, in the semantics of class-based models, records having the same key value represent the same real world thing. However, in class-based models, as shown in Figure 50, whether we adopt entity polyinstantiation or element polyinstantiation, we have to store the same real world thing to different security levels even if they only have one attribute that belongs to different security levels. This kind of approach reduces the possibility of data sharing and induces data storage duplication. In general, there is always a balance between sharing data and securing data in class-based models. For example, if we build a separate database for each security level, there will not be any possibility of any covert channel occurring between any security levels; thus, the data is the most secure. However, data sharing between these databases is nearly impossible. Users have to maintain these databases separately. To conclude, building a separate database for each security level will cause at least two problems: the first is that it will significantly increase the cost to maintain several databases compared to maintaining only one database with the same size. The second problem is that it will be difficult to maintain consistency of data between these databases. If we build a database for each individual security level, we have to maintain the non-secured data consistently.

In the IBMSM, any real world thing only has one identifier. An instance will belong to a security level if the instance possesses a property that belongs to the security level. In this model, the first and the second records in Figure 50(b) can be combined to store as one instance, *Instance1{(Name John, L_3),(Age 21, L_3), (Home Phone 709-737-1234, L_3), (Home Phone 709-737-1236, L_2)}*. In this case, no redundant data will be stored. So, the IBMSM is more efficient in storing multilevel data.

8.2 Null Values

As we have discussed before, in traditional class-based models all records of a class *(relation, table)* have the same number of attributes. These attributes express the lowest boundary of the class. Any record has to be assigned to a class before it can be stored to a database. However, there are lots of possibilities where some values of attributes cannot be decided or obtained when users insert a record into a database. The relational model introduces a null value to deal with this kind of situation [85]. But the semantics of the null value is incomplete. For example, the most commonly asked question about a null value is whether it means "we do not know" or "it is not accessible". In the multilevel security model a null value may also produce a covert channel as we have discussed before. So, the most traditional class-based MLR models do not support a null value. For example, SeaView and the MLR do not allow null values to avoid this kind of confusion and the problems that result. However, the null value problem comes from the theory of class-based models. Traditional relational models are based on the class-based model. They simply cannot avoid null values. Null values widely exist in traditional relational databases. By not supporting null values, class-based security models cannot be applied to this kind of dataset. Of course, this limits the applications of models.

The IBMSM is based on the instance-based data model, which is an open world model. In contrast to class-based models, which only support fixed attributes in a class, in the instance-based multilevel security model, an instance may be independent of any class. In the model, before an instance is stored into a database, one does not need to assume the instance belongs to certain classes. An instance belongs to a class because the set of properties in the class definition is a subset of the properties of the instance. An instance does not need to store any information about a property if the property does not belong to it. So, in the IBMSM the null value problem does not exist.

8.3 Sensitive Data in the Key Attributes

When dealing with sensitive data in the key attributes, traditional multilevel security models generate several problems.

8.3.1 Multilevel Key Attributes Problem

When the key consists of multiple attributes, traditional MLR models need to set all the key attributes in the same security level. In this case, the values of the key attributes cannot be borrowed from the lower level as indicated in MLR. Let us assume that we have the same relation as Figure 50(a); however, the key of the relation is defined as two attributes: *Name* and *Age*. Since the key attributes should be in the same security level, the value of the age attribute in the first record of Figure 50(a) cannot be borrowed from the second record (the lower level record). We have to set a separate value of this attribute at L_2 level, which is the same security level as the name attribute level. Figure 51 shows the resulting relation. The value of the *Age* attribute in the first record and the value of the *Age* attribute in the second record belong to different security levels. They are not related to each other even though they have the same value and represent the same aspect of the same thing. Of course, reducing the *borrow* capability in key attributes reduces the flexibility of databases. On the contrary, an IBMSM database does not have key attributes. An instance will automatically belong to a security level if it possesses a property that is in the security level. In the IBMSM model, the instance depicted in Figure 50(a) can be stored as *Instance1{(Name John, L_2), (Name John, L_3), (Age 21, L_3), (Home Phone 709-737-1234, L_3), (Home Phone 709-737-1236, L_2) }*. In this case, the L_2 level users will get all stored information of the instance, whereas the L_3 level users will get a view of the instance as: *Instance1{(name John, L_3), (Age 21, L_3), (Home Phone 709-737-1234, L_3)}*.

The IBMSM is clearly flexible to represent things that belong to multilevel security levels.

Person

TC	Name	$Label_1$	Age	$Label_2$	Home Phone	$Label_3$
L_2	John	L_2	21	L_2	(709)737-1236	L_2
L_3	John	L_3	21	L_3	(709)737-1234	L_3
L_3	Alice	L_3	25	L_3	(709)781-4321	L_3

Figure 51: Multiple Key Attributes Problem in MLR

8.3.2 Key Loophole Problem

Another problem when traditional MLR models deal with sensitive data in the key attributes is the key loophole problem [83]. A key loophole occurs when key attribute values are different in different security levels but represent the same entity in the real world. To illustrate the problem, consider the following example:

A fictional airline company, let us call it West Airline, keeps track of its passengers in an MLS database. The airline classifies its employees into two clearance categories, L_3 and L_2, by the sensitivity of information they are allowed to see. Every passenger must be accounted for, on every clearance level. However, the actual passenger's age and occupation may be hidden from some security levels. The sample relation shown in Figure 52 contains information about two passengers. All the information on passenger *Alice* is available for all two clearance levels. However, the information on passenger *John*, an air marshal, is more sensitive. The subjects on the L_2 level view *John*'s age, the correct occupation and seat number. The subjects on the L_3 level can only see the seat number; both *John*'s age and occupation are masked.

Passenger

Name		Age		Occupation		Seat#		TC
Alice	L_3	25	L_3	Student	L_3	123	L_3	L_3
Alice	L_3	25	L_3	Student	L_3	123	L_3	L_2
John	L_3	28	L_3	Teacher	L_3	125	L_3	L_3
John	L_3	30	L_2	Air Marshal	L_2	125	L_3	L_2

Figure 52: Passenger Relation

There will not be any problem to use MLR in this case. For example, the L_3 level and L_2 level users will have their views as in Figure 53.

However, if we need to mask *John*'s name on L_3 level as well, then we only have one way to do that in the current class-based MLR models, as shown in Figure 54.

Passenger L_2 Level View

Name		Age		Occupation		Site#		TC
Alice	L_3	25	L_3	Student	L_3	123	L_3	L_3
Alice	L_3	25	L_3	Student	L_3	123	L_3	L_2
John	L_3	28	L_3	Teacher	L_3	125	L_3	L_3
John	L_3	30	L_2	Air Marshal	L_2	125	L_3	L_2

Passenger L_3 Level View

Name		Age		Occupation		Site#		TC
Alice	L_3	25	L_3	Student	L_3	123	L_3	L_3
John	L_3	28	L_3	Teacher	L_3	125	L_3	L_3

Figure 53: Different Level Views on Passenger Table

Passenger

Name		Age		Occupation		Seat#		TC
Alice	L_3	25	L_3	Student	L_3	123	L_3	L_3
Alice	L_3	25	L_3	Student	L_3	123	L_3	L_2
David	L_3	28	L_3	Teacher	L_3	125	L_3	L_3
John	L_2	30	L_2	Air Marshal	L_2	125	L_3	L_2

Figure 54: Mask John's Name in Passenger Relation

In Figure 54, if we simply change mask *John* under a fictional name *David* on the L_3 level, it seems very simple. However, we now have a problem. In this case, the user on the L_2 level should know that *David* is simply a mask for *John*, and the passenger named *David* is fictional. The L_2 level user should treat all records related to *David* as non-existing. This can cause problems in situations when an L_2 level user has to communicate with lower level users. For example, the L_2 level user would not know that L_3 level users are aware of the passenger David in case he has a medical emergency.

Another way for the MLR models to deal with the key loophole problem is to hide the higher level data from lower level users. This approach is illustrated in Figure 55.

In this case, the information about the passenger John is completely hidden from the lower level users. However, unless we physically separate the seats of a higher security level passengers from the lower security passengers on an aircraft, which is impossible and against the purpose of having air marshals on board, this will cause confusion and possibly leak information to the lower level users.

Passenger

Name		Age		Occupation		Seat#		TC
Alice	L_3	25	L_3	Student	L_3	123	L_3	L_3
Alice	L_3	25	L_3	Student	L_3	123	L_3	L_2
John	L_2	30	L_2	Air Marshal	L_2	125	L_2	L_2

Figure 55: Hidden Passenger John from Lower Level Users

For example, a stewardess (assuming a L_3 level user) will notice that a passenger is in the seat number 125. However, according to the database, this seat should be empty. This apparently conflicting information can cause confusion and, more importantly, the L_3 level users to become aware of the fact that that information was kept hidden from them. Therefore, this approach has a potential to open covert channels [84].

In the instance-based multilevel security model, the instance identifier represents the real world thing. It is not assigned to any security level. And there is only one instance identifier for one real world instance. For example, to set a cover story for passenger *John,* in IBMSM model what we need to do is set a mask value for his name, age, and occupation. An IBMSM database stores John's information like the following *Instance1{(Name John, L_2), (Name David, L_3), (Age 30, L_2), (Age 28, L_3), (Occupation Air Marshal, L_2), (Occupation Teacher, L_3), (Seat# 125, L_3)}*. When a L_3 level user queries John's information, he/she will get a view as the following *Instance1{(Name David), (Age 28), (Occupation Teacher), (Seat# 125)}*. John's information is completely covered. The potential to produce covert channels in relation to traditional class-based MLR models is greatly limited because, in the instance-based multilevel security model, no key attribute needs to be dealt with.

8.4 Summary

In this chapter we have given several examples to demonstrate how the IBMSM model solves the problems which traditional MLR models cannot. The most significant difference between IBMSM and the class-based traditional multilevel security models is how the data are stored in databases. In the IBMSM, there is only one instance stored in databases, regardless of how many security levels to which the instance belongs. This gives IBMSM an advantage in achieving the purpose of building a multilevel security database that supports sharing data under different security clearances without any covert channel. Meanwhile, the IBMSM model also solves several problems that occur when traditional MLR models deal with sensitive data in the key attributes. The model also does not have the null value problem which widely exists in traditional MLR models. Therefore, the IBMSM model is very suitable for management of multilevel security data.

Chapter 9

Conclusions and Future Work

This chapter has three major sections. The first section describes the primary conclusions that have been arrived at through this research. The second section summarizes the research contributions. The third section outlines a number of directions for future research.

9.1 Conclusions

Traditional databases are designed for special purposes. Designing a database for multiple applications is a difficult task using traditional database models. The instance-based data model adopts a two-layered approach to manage data in a database. It provides the possibility to use one data set for multiple applications by storing data on the bottom level (the instance level) and building multiple applications on the top level (the class level). This thesis addresses issues related to the instance-based data model in three parts:

First, we assigned ontological semantics to the instance-based data model to address several semantic problems. We provided an ontological interpretation of the instance identifier and proposed a method to implement it. The research also provided a mean to express properties and their relationships in the instance-based data model, which reduced the complexity in the management of the instance-based database and improved the query efficiency of the model. Also, by assigning the semantics of Bunge's ontology to the instance-based data model, we developed a method to represent things in the real world as instances in the instance level. Related instances form a higher level conceptual thing; thus, in principal, when one combines all instances, the real world could be represented in the highest level. This method makes the instance-based data model closer to an ontological view of the nature of the real world, which in turn benefits the future development of the model. The research also defines several integrity rules of the instance-based data model to prevent possible inconsistencies in an instance-based database.

The instance-based data model provides not only flexibility and multiple-application potential [2], but also comparable or better performance than the traditional model. In the second part of the research, we provided evidence that the instance-based data model can perform queries even faster than relational model in some cases. The theoretical analysis and the empirical evaluation showed some interesting results and improved our confidence to develop the instance-based data model for future applications.

In the final part of the research, as an application to the instance-based theory, we proposed a new security model, the instance-based multilevel security model (IBMSM). We formally defined the instance-based multilevel security model. By defining data interpretation, integrity rules, and two-layered control model for the instance-based multilevel security control, we showed that the new model solves several problems in the multilevel security control field. The research extended operations of the traditional SQL and instance-based iQL statements to the IBMSM. We also proved that the instance-based multilevel security model is indeed a secure data model. There are two key features that make the IBMSM more secure than traditional multilevel security models: first, the model uses the '*post-belief*' method so that the higher level users can share lower level data without pre-definitions. This method improves flexibility in data sharing and reduces possible data redundancy in different security levels. Second, the model uses a two-layered control method to restrict database access. Since higher level security users can recognize all the properties which have been recognized by lower level users but not *vice versa*, this method guarantees that higher level security users may access more instances even through the same class as lower level users. By using two-layered control, the IBMSM acquires more powerful security than traditional multilevel security models.

9.2 Contributions

Our research provides several contributions for the instance-based database development.

First, following Bunge's ontology and [2], we have mapped the concepts in the database area to the real world and provided several implementation approaches for such mapping; thus, it provides an opportunity for further instance-based application development without the restriction implied by the traditional class-based model. Second, the performance of the instance-based data model was not addressed in previous research. It might seem natural that by providing more flexibility to applications the instance-based data model may have to trade off with lower efficiency. However, in this research, we proved that the instance-based data model can be even more efficient than the class-based model for certain query operations. For example, we proved that if the number of attributes in a table is more than 1.25-1.33 times the number of the queried attributes, the instance-based model (by using the second data structure) is much faster than the relational model. By developing a more suitable data structure that incorporates query optimization techniques, we believe the instance-based data model may perform even better than class-based models.

Third, previous work on the IBDM focused on building the model itself [2] [49] [57]. Our research proposes a new application model, the IBMSM model, to the security control area to solve several problems that exist in class-based security models. The success of the IBMSM demonstrates the instance-based data model is not only an abstract model but can be a platform to generate specific security models for different applications. Our research provides an example to implement the instance-based data model to an applied field. We believe that the instance-based data model can be explored in many other fields in the future.

9.3 Future Work

Future research on the instance-based data model can be of two categories. The first is related to the model itself. Although property precedence has an excellent capability for managing semantics in the instance-based data model, to manage the precedence system efficiently is a challenge. By definition, one property precedes another property because any instance that possesses the latter property must possess the former, as well. However, in the instance-based data model, an instance may gain or lose a property at any time. So, the preceding relation between properties may change over time. This presents a unique challenge to database developers. Also, since the structure of the instance-based data model is different from the class-based model, the access method (structure) should not be the same, as well. [70] provides a basic structure (B^+ tree) to implement an instance-based database system, since B^+ tree is designed to store fixed length of data, we believe B^+ tree is not the best structure for the instance-based data model. The next challenge would be to suggest a better structure other than a B+ tree for the instance-based data model.

The second area of needed research is related to the application of the model. We already know that the instance-based data model is suitable for building multiple applications on one data set. For example, in this research we have built the IBMSM which had several security level applications on one data set. Developing suitable models for application areas will be the next direction. Another challenge of the research is on merging the instance-based data model with the current models. As of now, data are still stored in different systems for different purposes. For example, sales systems are used for retail applications, and bank systems are used for financial organizations. To build an instance-based system that can efficiently utilize those data directly with the minimal cost would be the next research area. To build a two (or more) layered model is a possibility, as suggested in [2]. However, better approaches to integrate data from class based model to the instance-based database are still desired.

The instance-based data model provides many potential opportunities for database applications. We believe that as more research is done on the topic, new application areas will be found and more advantages of the model would be discovered.

Bibliography

[1] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison Wesley Longman Inc. 2000. Print.

[2] Jeffrey Parsons and Yair Wand. "Emancipating Instances from the Tyranny of Classes in Information Modeling." *ACM Transactions on Database Systems* 25.2 (2000): 228-268. Print.

[3] Mario Bunge. *Ontology I: The Future of the World, Treatise on Basic Philosophy*. New York: D. Reidel Publishing Co. Inc. 1977. Print.

[4] Mario Bunge. *Ontology II: A World of Systems, Treatise on Basic Philosophy*. New York: D. Reidel Publishing Co. Inc. 1979. Print.

[5] Jeffrey Parsons and Jianmin, Su. "The Instance-based Multilevel Security Model (IBMSM)." *Global Perspectives on Design Science Research. Lecture Notes in Computer Science* 2010.6105 (2010):365-380. Print.

[6] Peter Rob and Carlos Coronel. *Database Systems*. Boyd & Fraser Pub Co. 1995. Print.

[7] Avi Silberschatz, et al. *Database System Concepts*. McGraw-Hill. 1996. Print.

[8] Scott Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*, John Wiley & Sons, Inc. 2003. Print.

[9] Peter Chen. "The Entity-Relationship Model--Toward a Unified View of Data." *ACM Transactions on Database Systems* 1.1 (1976): 9-36. Print.

[10] Michael Ashburner, et al. "Gene Ontology: Tool for the Unification of Biology." *Nature Genetics* 25.1 (2000): 25-29. Print.

[11] Nicola Guarino. "Formal Ontology, Conceptual Analysis and Knowledge Representation." *Spec. issue of International Journal of Human and Computer Studies* 43.5/6 (1995): 625-640. Print.

[12] Jeffrey Parsons and Yair Wand. "Using Objects for Systems Analysis." *Communications of the ACM* 40.12 (1997):104-110. Print.

[13] Yair Wand and Ron Weber. "A Model of Control and Audit Procedure Change in Evolving Data Processing Systems." *The Accounting Review* 64.1 (1989): 87-107. Print.

[14] Yair Wand and Ron Weber. "An Ontological Evaluation of Systems Analysis and Design Methods." *Information System Concepts: An In-depth Analysis*. Elsevier Science (1989): 79-107. Print.

[15] Yair Wand and Ron Weber. "An Ontological Model of an Information System." *IEEE Transactions on Software Engineering* 16.11 (1990): 1282-1292. Print.

[16] Yair Wand and Ron Weber. "Mario Bunge's Ontology as a Formal Foundation for Information Systems Concepts." *Studies on Mario Bunge's Treatise*, Ed. Paul, Weingartner, and Georg, Dorn. Atlanta: Rodopi. 1990. Print.

[17] Yair Wand and Ron Weber. "A Unified Model of Software and Data Decomposition." *ICIS '91 Proceedings of the twelfth international conference on Information systems.* Ed. Janice, DeGross, et al. University of Minnesota Minneapolis. 1991. Print.

[18] Yair Wand and Ron Weber. "On the Ontological Expressiveness of Information Systems Analysis and Design Grammars." *Journal of Information System* 3.4 (1993): 217-237. Print.

[19] Yair Wand and Ron Weber. "On the Deep Structure of Information Systems." *Journal of Information System* 5,3 (1995): 203-223. Print.

[20] Joshu, Greenbaum. "The Quest for the Universal Data Model." Intelligent Enterprise. Web. May 31, 2003.
<http://www.intelligententerprise.com/030531/609enterprise1_1.jhtml>

[21] Dennis Tsichritzis and Frederick Lochovsky. "Hierarchical Database Management: A Survey." *ACM Computing Surveys* 8.1 (1976): 105-123. Print.

[22] Robert Taylor and Randall Frank. "CODASYL Data-Base Management Systems." *ACM Computing Surveys* 8.1 (1976): 67 – 103. Print.

[23] Edgar Codd. "A Relational Model of Data Large Shared Data Banks." *Spec. issue of Communications of the ACM* 26.1 (1983): 64-69. Print.

[24] Christopher Date. *Database Relational Model: A Retrospective Review and Analysis*. Addison-Wesley. 2000. Print.

[25] Peter Chen. "Entity-Relationship Modelling: Historical Events, Future Trends, and Lessons Learned." *Software Pioneers: Contributions to Software Engineering*. New York: Springer, 2002. Print.

[26] Peter Chen. "A Preliminary Framework for Entity-Relationship Models" *Proceedings of the Second International Conference on the Entity-Relationship Approach to Information Modeling and Analysis*. Amsterdam: North-Holland. 1981. Print.

[27] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley. 1994. Print.

[28] Pedro Sampaio and Norman Paton. "Deductive Object-oriented Database Systems: A Survey." *Lecture Notes in Computer Science*. Berlin / Heidelberg: Springer. 1997. Print.

[29] Douglas Barry and Joshua Duhl. *Object Storage Fact Books: Object DBMSs and Object-Relational Mapping*. Barry & Associates Inc. 2001. Print.

[30] Burleson Consulting. Oracle Object Oriented Relational Features. Oracle Tips. Web. 2001. <http://www.dba-oracle.com/art_oracle_obj.htm>

[31] Steven Wort, et al. *Professional SQL Server 2005 Performance Tuning*. Wrox Press. 2008. Print.

[32] Michael Hammer and Dennis McLeod. "Database Description with SDM: A Semantic Database Model." *ACM Transactions on Database System* 6.3 (1981): 351-386. Print.

[33] Ray Reiter. "On Closed World Data Bases." *Readings in nonmonotonic reasoning*. San Francisco: Morgan Kaufmann Publishers Inc. 300-310. 1987. Print.

[34] AnHai Doan and Alon Halevy. "Semantic Integration Research in the Database Community: A Brief Survey." *AI Magazine* 26 (2005): 83-94. Print.

[35] Carlo Batini, et al. "A Comparative Analysis of Methodologies for Database Schema Integration". *ACM Computing Surveys* 18.4 (1986): 323-364. Print.

[36] Ahmed Elmagarmid and Calton Pu. "Guest Editors' Introduction to the Special Issue On Heterogeneous Databases." *ACM Computing Surveys* 22.3 (1990): 175-178. Print.

[37] Amit Sheth and James Larson. "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases." *ACM Computing Surveys* 22.3 (1990): 183-236. Print.

[38] Christine Parent and Stefano Spaccapietra. "Issues and Approaches of Database Integration." *Communications of the ACM* 41.5 (1998): 166-178. Print.

[39] Tova Milo and Sagit Zohar. "Using Schema Matching to Simplify Heterogeneous Data Translation." *Proceedings of the 24rd International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc. 1998. Print.

[40] Jayant Madhavan, et al. "Corpus-based Schema Matching." *Proceedings of the 21st International Conference on Data Engineering*. IEEE Computer Society Press. 2005. Print.

[41] Jayant Madhavan, et al. "Representing and Reasoning about Mappings between Domain Models." *Eighteenth National Conference on Artificial intelligence*. Menlo Park: American Association for Artificial Intelligence. 2002. Print.

[42] Wen-Syan Li, et al. "Database Integration Using Neural Network: Implementation and Experience." *Knowledge and Information Systems* 2.1 (2000): 73-96. Print.

[43] Chris Clifton, et al. "Experience With a Combined Approach to Attribute-Matching Across Heterogeneous Databases." *Proceedings of the IFIP Conference on Data Semantics*. Chapman & Hall, 1997. Print.

[44] Jacob Berlin and Amihai Motro. "Database Schema Matching Using Machine Learning With Feature Selection." *Proceedings of the 14th International Conference on Advanced Information Systems Engineering*. Springer-Verlag, 2002. Print.

[45] Mauricio Hernández and Salvatore Stolfo. "The Merge/Purge Problem For Large Databases." *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, New York: ACM. 1995. Print.

[46] Jason Wang, et al. *Data Mining in Bioinformatics*. London: Springer-Verlag London Limited, 2005. Print.

[47] Cedric Raguenaud and Jessie Kennedy. "Multiple Overlapping Classifications: Issues and Solutions." *14th International Conference on Scientific and Statistical Database Management*. IEEE Computer Society Press. 2002. Print.

[48] Josh Howard. Universal Data Models and Patterns: Developing Higher Quality Data Models in Less Time. Embarcadero Technologies. Web. June 2009.
<http://datamodel.wordpress.com/2009/06/11/universal-data-models/>

[49] Jianmin Su and Jeffrey Parsons. "Analysis of Data Structures to Support the Instance-based Database Model." *First International Conference on Design Science Research in Information Systems and Technology*. Claremont, California. 2006

[50] Edgar Sibley and Larry Kerschberg. "Data Architecture and Data Model Considerations." *AFIPS '77 Proceedings of the June 13-16, 1977, national computer conference*. New York: ACM. 1977. Print.

[51] David Shipman. "The Functional Data Model and The Data Language Daplex." *ACM Transactions on Database Systems* 6,1 (1981): 140-173. Print.

[52] Gabriel Kuper and Moshe Vardi. "The Logical Data Model." *ACM Transactions on Database Systems* 18,3 (1993): 379-413. Print.

[53] Mikko Rönkkö, et al. "Benefits of an Item-Centric Enterprise-Data Model in Logistics Services: A Case Study." *Computers in Industry* 58.8/9 (2007): 814-822. Print.

[54] Tsau Lin. "Attribute Based Data Model and Polyinstantiation." *Proceedings of the IFIP 12th World Computer Congress on Education and Society - Information Processing '92*. 2. 1992. Amsterdam: North Holland. Print.

[55] Ian Niles and Adam Pease. "Towards a Standard Upper Ontology." *In Proceedings of the 2nd International Conference on Formal Ontology in Information Systems*. 2001. New York: ACM Press. Print.

[56] David Dewitt. "The Wisconsin Benchmark: Past, Present, and Future." *The Benchmark Handbook for Database and Transaction Systems*. 1993. Ed. Jim, Gray. Morgan Kaufmann. Print.

[57] Jeffrey Parsons and Yair Wand. "Property-Based Semantic Reconciliation of Heterogeneous Information Sources." *Proceedings of the 21st International Conference on Conceptual Modeling*. 2002. London: Springer-Verlag.

[58] John Howie. *Fundamentals of Semigroup Theory*. Springer New York. 1995. Print.

[59] Von Bertalanffy. "General System Theory - A New Approach to Unity of Science (Symposium)." *Human Biology* 23.4 (1951): 303-361. Print.

[60] Von Bertalanffy. "An Outline of General System Theory." *British Journal for the Philosophy of Science* 1.2 (1950): 139-164. Print.

[61] Ross Ashby. *An Introduction to Cybernetics*. London: Chapman & Hall. 1956. Print.

[62] Kenneth Boulding. "General System Theory: The Skeleton of Science." *Management of Science* 2.3 (1956): 197-208. Print.

[63] URL. W3C Web set. Web. <http://www.w3.org/Addressing>

[64] Peter Lucas, et al. "Distributed Knowledge Representation using Universal Identity and Replication." Technical Report MAYA-05007. MAYA Design Inc. 2004

[65] Mehmed Kantardzic. *Data Mining: Concepts, Models, Methods, and Algorithms*. John Wiley & Sons. 2003. Print.

[66] Joachim Hammer, et al. "The Stanford Data Warehousing Project." *IEEE Data Engineering Bulletin* 18 (1995): 41-48. Print.

[67] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd ed.)*. Upper Saddle River: Prentice Hall. 2003. Print.

[68] Sotiris Kotsiantis. "Supervised Machine Learning: A Review of Classification Techniques." *Informatica Journal* 31 (2007): 249-268. Print.

[69] Richard Duda, et al. *Unsupervised Learning and Clustering (2nd edition).* New York: Wiley. 2001. Print.

[70] Jianmin Su. *A Database Management System to Support the Instance-based Data Model: Design, Implementation, and Evaluation*. Master's Thesis, Memorial University of Newfoundland. 2003. Print.

[71] Daren Bieniek, et al. *Implementing and Maintaining Microsoft SQL Server 2005*. Microsoft Press. 2006. Print.

[72] Mike Stonebraker, et al. "C-Store: A Column-Oriented DBMS." *Proceedings of the 31st International Conference on Very Large Data Bases*, Trondheim, Norway, 2005. ACM. Print.

[73] Thomas Cormen, et al. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 2001. Print.

[74] Teresa Lunt, et al. "The SeaView Security Model." *IEEE Transactions on Software Engineering* 16.6 (1990): 593-607. Print.

[75] Elliott Bell and Leonard Padula. "Secure Computer Systems: Unified Exposition and Multics Interpretation." Technical Report MTR-2997, The Mitre Corporation, Burlington Road, Bedford, MA, USA, 1973. Print.

[76] Ravi Sandhu and Fang Chen. "The Multilevel Relational Data Model." *ACM Transaction on Information and System Security* 1.1 (1998): 93–132. Print.

[77] Nenad Jukica, et al. "A Belief-Consistent Multilevel Secure Relational Data Model." *Information Systems* 24.5 (1999): 377-400. Print.

[78] Xiaolei Qian and Teresa Lunt. "Tuple-level vs. Element-level classification." *Results of the Sixth Working Conference of IFIP Working Group on Database Security on Database security, VI : Status and Prospects*. New York: Elsevier Science Inc. 1993. Print.

[79] Sushil Jajodia and Ravi Sandhu. "Toward a Multilevel Secure Relational Data Model." *ACM SIGMOD Record* 20.2 (1991): 50-59. Print.

[80] Kenneth Smith and Marianne Winslett. "Entity Modeling in the MLS Relational Model." *Proceedings of the 18th International Conference on Very Large Data Bases*. San Francisco: Morgan Kaufmann Publishers Inc. 1992. Print.

[81] J. Goguen, and J. Meseguer. "Security Policies and Security Models." *IEEE Symposium on Security and Privacy*. Oakland: IEEE Computer Society Press. 1982. Print.

[82] Teresa Lunt. "Polyinstantiation: An Inevitable Part of a Multilevel World." *Proceedings of Fourth IEEE Workshop Computer Security Foundations*. Menlo Park: IEEE Computer Society Press. 1991. Print.

[83] Nenad Jukic, et al. "Closing the Key Loophole in MLS Databases." *ACM SIGMOD Record* 32,2 (2003): 15-20. Print.

[84] Ira Moskowitz, et al. "Covert Channels and Anonymizing Networks." *Proceedings of the 2003 ACM Workshop on Privacy in the Electronic Society* New York: ACM. 2003. Print.

[85] Edgar Codd. "Extending the Database Relational Model to Capture More Meaning." *ACM Transactions on Database Systems* 4.4 (1979): 397-434. Print.

[86] MySQL work bench 5.2. Web. <http://www.mysql.com/>

Appendix: Wisconsin Benchmark Queries and Results

Query	Non-indexed property queries	Rdb	idb	Comment
1	Select unique1 from TENKTUP1 where unique1 between 0 and 10099;	5.4	0.73	1% selection
2	Select unique1 , two from TENKTUP1 where unique1 between 0 and 10099;	5.37	3.76	
3	Select unique1 , two, four from TENKTUP1 where unique1 between 0 and 10099;	5.6	8.43	
4	Select unique1 , two, four, unique3 from TENKTUP1 where unique1 between 0 and 10099;	5.87	12.9	
5	insert into tmp Select unique1 from TENKTUP1 where unique1 between 792 and 100791;	7.78	2.71	10% selection
6	insert into tmp Select unique1, two from TENKTUP1 where unique1 between 792 and 100791;	7.73	7.8	
7	insert into tmp Select unique1 , two, four from TENKTUP1 where unique1 between 792 and 100791;	7.9	12.61	
8	insert into tmp Select unique1 from TENKTUP1;	21.22	12.57	100% selection
9	insert into tmp Select unique1, two from TENKTUP1;	21.06	18.25	
10	insert into tmp Select unique1 , two, four from TENKTUP1;	21.56	24.1	
	Clustered-Index property queries			
11	Select unique2 from TENKTUP1 where unique2 between 0 and 10099;	0.14	0.06	1% selection
12	Select unique2 , two from TENKTUP1 where unique2 between 0 and 10099;	0.13	0.14	
13	Select unique2 , two, four from TENKTUP1 where unique2 between 0 and 10099;	0.14	0.19	
14	Select unique2 from TENKTUP1 where unique2 between 792 and 100791;	0.64	0.2	10% selection
15	Select unique2 , two from TENKTUP1 where unique2 between 792 and 100791;	0.62	0.66	
16	Select unique2 , two, four from TENKTUP1 where unique2 between 792 and 100791;	0.72	0.67	
17	insert into tmp Select unique2 from TENKTUP1;	21.15	11.28	100% selection
18	insert into tmp Select unique2, two from TENKTUP1;	21.01	17.77	
19	insert into tmp Select unique2 , two, four from TENKTUP1;	21.67	22.77	
	Non-Clustered-Index property queries			
20	Select unique3 from TENKTUP2 where unique3 between 0 and 10099;	0.11	0.06	1% selection
21	Select unique3, two from TENKTUP2 where unique3 between 0 and 10099;	0.33	0.14	

22	Select unique3, two, four from TENKTUP2 where unique3 between 0 and 10099;	0.33	0.19	
23	Select unique3 from TENKTUP2 where unique3 between 792 and 100791;	0.2	0.2	10% selection
24	Select unique3, two from TENKTUP2 where unique3 between 792 and 100791;	1.45	0.66	
25	Select unique3, two, four from TENKTUP2 where unique3 between 792 and 100791;	1.4	0.67	
26	insert into tmp Select unique3 from TENKTUP2;	13.88	11.28	100% selection
27	insert into tmp Select unique3, two from TENKTUP2;	25.44	17.77	
28	insert into tmp Select unique3, two, four from TENKTUP2;	26.72	22.77	
	Aggregation			
29	select min(unique1) from tenktup1;	5.34	0.84	
30	select min(unique2) from tenktup1;	0.05	0.05	
31	select min(unique3) from tenktup1 group by onepercent;	6.8	4.52	
32	select min(unique2) from tenktup1 group by onepercent;	5.65	4.48	
33	select sum(unique3) from tenktup1 group by onepercent;	5.38	5.6	
34	select sum(unique2) from tenktup1 group by onepercent;	5.65	4.91	
	Update			
35	update tenktup1 set unique2=1000001 where unique2=19000;	0.11	0.09	Clustered-Index
36	update tenktup2 set unique3=1000002 where unique3=19000;	0.13	0.09	Non-Clustered-Index
37	update tenktup1 set unique2=1000002 where unique1=19000;	6.79	1.4	Non-Index
	Join			
38	select tenktup1.unique1, tenktup2.unique3 from tenktup1, tenktup2 where tenktup1.unique2=tenktup2.unique2 and tenktup1.unique1<10000	6.49	4.15	Join on clustered-index property
39	select tenktup1.unique1, tenktup2.unique3 from tenktup1, tenktup2 where tenktup1.unique2=tenktup2.unique2 and tenktup2.unique3<10000	0.27	0.36	