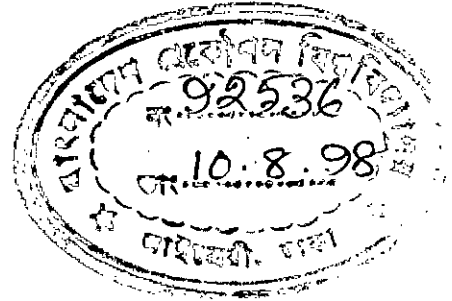


# Database Characterization using Induction

by



Md. Humayun Kabir

Submitted to the Department of Computer Science and Engineering  
in partial fulfillment of the degree  
of  
Master of Science in Engineering (CSE)



Department of Computer Science and Engineering  
Bangladesh University of Engineering and Technology  
July, 1998.

---

# Database Characterization using Induction

A Thesis

Submitted by

**Md. Humayun Kabir**


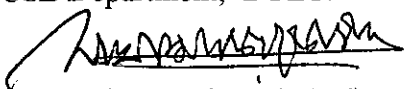


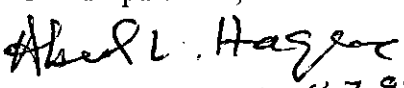
**Roll No: 9405034 P, Session: 1993-94-95**

For the partial fulfillment of the degree of M. Sc. Engg. in  
Computer Science and Engineering

Examination held on

July 05, 1998.

Approved as to style and contents by:

-  5.7.98
1. (Dr. Chowdhury Mofizur Rahman) (Supervisor) Chairman  
Assistant Professor  
CSE Department, BUET.
  2.  (Dr. Mohammad Kaykobad) (Head) Member  
Head and Professor  
CSE Department, BUET.
  3.  (Dr. M. Masroor Ali) 5.7.98 Member  
Assistant Professor  
CSE Department, BUET.
  4.  5.7.98 (Dr. Md. Abul Kashem Mia) Member  
Assistant Professor  
CSE Department, BUET.
  5.  (Dr. Abul L. Haque) 5.7.98 Member  
Associate Professor & Head (External)  
CS Department, NSU.
-


## ACKNOWLEDGEMENT

Profound knowledge and keen interest of Dr. Chowdhury Mofizur Rahman in the fields of Database and Machine Learning has influenced the author to carry out a research work in these fields. This research work was done under his kind supervision. His endless patience, scholarly guidance, constant and energetic supervision, valuable advice, suggestions and encouragement at all stages have made it possible to complete this thesis.

The author feels pleased to have the opportunity of expressing his heart-felt and most sincere gratitude to all who helped him in course of this research work.

Special gratitude are due to Dr. Mohammad Kaykobad, Head, CSE Department, BUET for his kind cooperation in the research work.

The all out support, cooperation and services rendered by the faculty members and the staff of the Department of Computer Science and Engineering, BUET are also acknowledged with sincere thanks.

 05/07/98

---

(Md. Humayun Kabir)

## ABSTRACT

Designing a good relational database for a particular system is still a challenging job for database designer. The designer must have expert knowledge on relational database theories, attribute dependencies (i.e., functional and multi valued dependencies) and normalization. Different design methodologies practiced at present do not use machine learning techniques. Seeing examples of data, machine learning techniques provide us the underlying attribute dependencies and the inherent structure that the database would possess. After having extensive investigation, it has become evident that machine learning techniques would certainly be a useful tool in designing good relational databases and the designer would not necessarily be a theoretical expert. And the design effort would reduce merely to collect attributes and the values on the attributes from a particular system. A new design framework has been proposed in which all the attributes from a particular system has been collected first. The initial database schema has been defined taking all the attributes in a single relation and instances have been collected on it. Machine Learning software has been developed for getting the underlying attribute dependencies (i.e., functional and multi-valued dependencies) in the initial database. New algorithms have been proposed and software has been developed to get relational database in 2NF using the set of functional and multi valued dependencies. Software has been developed for getting good relational databases by further characterizing and normalizing these 2NF relations into 3NF, BCNF and 4NF.

---

# Table of Contents

<i>List of Figures</i> .....	i
<i>List of Tables</i> .....	iii
<i>List of Algorithms</i> .....	iv
<i>List of Symbols</i> .....	v
<i>List of Notation</i> .....	vi
<b>1 Introduction</b> .....	<b>1</b>
<b>1.1 Data Analysis and Database Model</b> .....	<b>1</b>
<b>1.2 The Goal of Relational Database Design</b> .....	<b>1</b>
<b>1.3 Different Normal Forms</b> .....	<b>2</b>
1.3.1 First Normal Form (1NF).....	2
1.3.2 Second Normal Form (2NF).....	2
1.3.3 Third Normal Form (3NF).....	2
1.3.4 Boyce-Codd Normal Form (BCNF).....	2
1.3.5 Fourth Normal Form (4NF).....	2
<b>1.4 Functional and Multi-Valued dependencies</b> .....	<b>3</b>
1.4.1 Functional Dependency.....	3
1.4.2 Multi Valued Dependency.....	3
<b>1.5 Normalization based on Functional and Multi Valued Dependencies</b> .....	<b>3</b>
<b>1.6 Machine Learning Algorithms for determining Functional and Multi Valued Dependencies</b> .....	<b>4</b>
<b>1.7 Previous Work</b> .....	<b>4</b>
<b>1.8 Scope of the Work</b> .....	<b>5</b>
<b>1.9 Proposed Design Framework and new Algorithms</b> .....	<b>5</b>
<b>2 Data Models</b> .....	<b>6</b>
<b>2.1 Object-Based Logical Models</b> .....	<b>6</b>
2.1.1 The Entity-Relationship Model.....	6
2.1.2 The Object-Oriented Model.....	8
<b>2.2 Record-Based Logical Models</b> .....	<b>9</b>
2.2.1 Relational Model.....	9
2.2.2 Network Model.....	10
2.2.3 Hierarchical Model.....	11

---

2.3	Differences Between the Models .....	11
<b>3</b>	<b><i>Structure of Relational Databases</i></b> .....	<b>12</b>
3.1	Basic Structure.....	12
3.2	Database Scheme.....	14
3.3	Keys.....	16
3.4	Query Languages .....	19
<b>4</b>	<b><i>Functional Dependency</i></b> .....	<b>20</b>
4.1	Basic Concepts .....	20
4.2	Closure of a Set of Functional Dependencies .....	25
4.3	Closure of Attribute Sets .....	27
4.4	Canonical Cover.....	29
<b>5</b>	<b><i>Relational Database Design</i></b> .....	<b>31</b>
5.1	Pitfalls in Relational Database Design.....	31
5.1.1	Representation of Information .....	32
5.1.2	Loss of Information.....	35
5.2	Normalization Using Functional Dependencies .....	40
5.2.1	Desirable Properties of Decomposition .....	41
5.3	Boyce-Codd Normal Form.....	45
5.4	Third Normal Form.....	51
5.5	Comparison of BCNF and 3NF .....	53
5.6	Normalization Using Multi Valued Dependencies .....	55
5.6.1	Multi-valued Dependencies .....	56
5.6.2	Theory of Multi-valued Dependencies .....	58
5.6.3	Fourth Normal Form .....	61
<b>6</b>	<b><i>Characterizing a Database Relation</i></b> .....	<b>66</b>
6.1	Preliminaries.....	66
6.2	Characterization by functional dependencies .....	66
6.3	Characterization by multi-valued dependencies.....	72
6.4	Inductive learning of functional dependencies.....	77
6.4.1	The problem of incremental characterization: monotonicity.....	77

---

6.4.2	Inductive learning of possible multi-valued dependencies .....	80
6.4.3	From possible multi-valued dependencies to satisfied multi-valued dependencies: a querying approach .....	85
<b>7</b>	<b><i>Proposed Design Frame Work and Algorithms</i></b> .....	<b>91</b>
7.1	Design Framework .....	91
7.2	Proposed New Algorithms .....	92
<b>8</b>	<b><i>Experimental Results</i></b> .....	<b>95</b>
8.1	Target System .....	95
8.2	Results .....	95
<b>9</b>	<b><i>Conclusions</i></b> .....	<b>111</b>
<b>10</b>	<b><i>References</i></b> .....	<b>112</b>

---

## List of Figures

Figure 2.1 A Sample E-R diagram	07
Figure 2.2 A Sample relational database	10
Figure 2.3 A Sample of network database	10
Figure 2.4 A Sample of hierarchical database	11
Figure 3.1 The <i>deposit</i> relation	13
Figure 3.2 The <i>customer</i> relation	15
Figure 3.3 The <i>customer</i> relation	17
Figure 3.4 The <i>borrow</i> relation	18
Figure 3.5 The <i>branch</i> relation	18
Figure 4.1 Sample relation <i>r</i>	21
Figure 4.2 The <i>customer</i> relation	22
Figure 4.3 The <i>borrow</i> relation	23
Figure 4.4 The <i>branch</i> relation	24
Figure 5.1 Simple <i>branch</i> relation	32
Figure 5.2 The <i>borrow</i> relation	33
Figure 5.3 The branch X borrow relation	34
Figure 5.4 The relations <i>amt</i> and <i>loan</i>	37
Figure 5.5 The relation <i>amt</i> X <i>loan</i>	37
Figure 5.6 An instance of <i>Banker-Scheme</i>	54
Figure 5.7 Tabular representation of $\alpha \rightarrow \beta$	56
Figure 5.8 Relation <i>bc</i> , an example of redundancy in a BCNF relation	57
Figure 5.9 An illegal <i>bc</i> relation	57
Figure 5.10 Projection of relation <i>r</i> onto a 4NF decomposition of <i>R</i>	64
Figure 6.1 Refutation of the satisfaction of an fd with two tuples	69
Figure 6.2 Refutation of the satisfaction of an mvd with three tuples	74
Figure 6.3 Non-trivial pmvds on $R = \{ A, B, C \}$ , ordered by generality	82
Figure 8.1 The <i>university</i> relation	97
Figure 8.2 The <i>student</i> relation	98
Figure 8.3 The <i>borrow</i> relation	98
Figure 8.4 The <i>student_info</i> relation	99
Figure 8.5 The <i>student_course</i> relation	99



Figure 8.6 The <i>course</i> relation	100
Figure 8.7 The <i>borrow_1</i> relation	107
Figure 8.8 The <i>student_regular</i> relation	108
Figure 8.9 The <i>advisor</i> relation	109
Figure 8.10 The <i>student_course</i> relation	109
Figure 8.11 The <i>course</i> relation	110
Figure 8.12 The <i>student_borrow</i> relation	110
Figure 8.13 The <i>student_special</i> relation	110
Figure 8.14 The <i>student_research</i> relation	110

## List of Tables

Table 6.1 A relation satisfying some functional dependencies _____	67
Table 6.2 A relation satisfying some multi-valued dependencies _____	72
Table 6.3 MVD is not monotonic: $r$ satisfies $A \twoheadrightarrow B$ , $r \cup \{t_1, t_2\}$ does not, and $r \cup \{t_1, t_2, t_3, t_4\}$ again does _____	80
Table 6.4 A relation contradicting the mvds $A \twoheadrightarrow C$ and $A \twoheadrightarrow BD$ _____	87
Table 6.5 $\langle p, n \rangle$ is complete for the relation in Table 6.4 _____	87

## List of Algorithms

Algorithm 4.1 An algorithm to compute $\alpha^+$ ; the closure of $\alpha$ under $F$	27
Algorithm 5.1 An algorithm for testing dependency preservation	43
Algorithm 5.2 BCNF decomposition algorithm	47
Algorithm 5.3 Dependency preserving loss less join decomposition algorithm into 3NF	52
Algorithm 5.4 4NF decomposition algorithm	62
Algorithm 6.1 Downward characterization by functional dependencies	70
Algorithm 6.2 Specialization of an fd contradicted by two tuples	71
Algorithm 6.3 Downward characterization by multi-valued dependencies	75
Algorithm 6.4 Specialization of a dependency basis contradicted by two tuples	76
Algorithm 6.5 Incremental downward characterization by functional dependencies	78
Algorithm 6.6 Non-incremental downward characterization by functional dependencies	79
Algorithm 6.7 Incremental downward characterization by possible multi-valued dependencies	83
Algorithm 6.8 Specialization of a possible multi-valued dependency contradicted by three witnesses	84
Algorithm 7.1 The algorithm for merging functional dependencies	92
Algorithm 7.2 The algorithm for constructing relations in 2NF	93

## List of Symbols

- $D$  : The set of functional and multi valued dependencies.
- $F$  : The set of functional dependencies.
- $F^+$  : The closure of  $F$ .
- $F_c$  : The canonical cover of  $F$ .
- $\cup$  : The set union operation.
- $\cap$  : The set intersection operation.
- $\times$  : The set Cartesian product operation.
- $\sigma$  : The SQL select operation.
- $\Pi$  : The SQL project operation.
- $=$  : The equivalence symbol.
- $\subseteq$  : The subset symbol;  $A \subseteq B$  means  $A$  is a subset of  $B$ .
- $\supseteq$  : The superset symbol;  $A \supseteq B$  means  $A$  is superset of  $B$ .
- $\in$  : The membership symbol;  $a \in A$  means  $a$  is a member or element of  $A$ .
- $\rightarrow$  : The functional dependency symbol;  $A \rightarrow B$  means  $B$  is functionally dependent on  $A$ .
- $\rightarrow\rightarrow$  : The multi valued dependency symbol;  $A \rightarrow\rightarrow B$  means  $A$  multi determines  $B$ .
- $\emptyset$  : The empty set.
- $|=$  : The hold symbol;  $p \models A \rightarrow\rightarrow B$  means set of relations  $p$  holds the dependency  $A \rightarrow\rightarrow B$ .
- $\Rightarrow$  : The possible multi valued dependency symbol.

## List of Notations

$F_1, F_2, \dots, F_n$  represents the sets of resultant restrictions driven from  $F$  the restriction of  $R$  after having the decomposition of  $R$  relation scheme into  $n$  smaller relation schemes.

$F$  is the union of  $F_1, F_2, \dots, F_n$ .

Greek letters  $\alpha, \beta, \gamma$  represents any attribute/set of attributes of a relation.

Italic capital letters  $A, B, C, \dots, K, \dots, X, Y, Z$  represents any attribute/set of attributes of a relation.

Italic small letters  $a, b, c, \dots, k, \dots, x, y, z$  represents value/values on corresponding attribute/set of attributes  $A, B, C, \dots, K, \dots, X, Y, Z$ .

$R = (attr_1, attr_2, \dots, attr_n)$  is a definition of a relation scheme  $R$  of  $n$  attributes.

$R_1, R_2, \dots, R_n$  represents decomposition of a relation scheme  $R$  into  $n$  smaller relation schemes.

$r(attr_1, attr_2, \dots, attr_n)$  is a definition of a relation  $r$  on the relation scheme  $R$ .

$r(R)$  is a set of relations  $r$  on the relation scheme  $R$ .

$t_i$  is  $i$ th tuple in the set of relations  $r$ .

$t[X]$  is the  $X$ -value of tuple  $t$  in  $r$

# Chapter 1

## Introduction

92536  
10.8.98

### 1.1 Data Analysis and Database Model

Data analysis concentrates on understanding and documenting data, which it is argued, represents the 'fundamental building block of systems'. Even if applications change, the data already collected may still be relevant to the new or revised system and therefore need not to be collected and validated again. The data model, the result of data analysis, is orientated towards that part of the real world it represents and should be implementation independent [3].

Underlying structure of a database is the concept of a data model, a collection of tools for describing data, data relationships, data semantics, and consistency constraints. The various data models, such as object-based logical models, record-based logical models, and physical data models, have been discussed elaborately in chapter two.

Relational data model is a record-based data model and is used in describing data at the conceptual and view levels. In this model database is structured in fixed format records of several types. Each record type defines a fixed number of attributes and each attribute is usually of a fixed length. The use of fixed-length records simplifies the physical-level implementation of the database. This makes the relational data models most popular data models.

A relational database consists of a collection of tables, each of which is assigned a unique name. A row in a table represents a relationship among a set of values. The structure of relational database has been discussed in chapter three.

### 1.2 The Goal of Relational Database Design

In general, the goal of a relational database design is to generate a set of relation schemes that allow us to store information without unnecessary redundancy, yet allow us to retrieve information easily. One approach is to design schemes that are in an appropriate normal form.

✓  
2021

Due to bad database design repetition of information and loss of information incur in databases. The repetition of information complicates updating, insertion and deletion in a database [14].

## **1.3 Different Normal Forms**

The various normal forms have been discussed in chapter five.

### **1.3.1 First Normal Form (1NF)**

First normal form ensures that all the attributes are atomic (that is, in the smallest possible components). This means that there is only one possible value for each domain and not a set of values. This is often expressed as the fact that relations must not contain repeating groups.

### **1.3.2 Second Normal Form (2NF)**

Second Normal Form ensures that all non-key attributes are functionally dependent on all of the keys.

### **1.3.3 Third Normal Form (3NF)**

A database is in 3NF if for every functional dependency, that holds on the relation/relations of that database, the antecedent is the key or the consequent is a part of any key for the relation/relations.

### **1.3.4 Boyce-Codd Normal Form (BCNF)**

A database is in BCNF if for every functional dependency of every relation, the antecedent is the key for the relation.

### **1.3.5 Fourth Normal Form (4NF)**

A database is in 4NF if for every multi valued dependency of every relation, the antecedent is the key for the relation.

## 1.4 Functional and Multi-Valued dependencies

Functional and multi-valued dependencies are constraints on the set of legal relations. They allow us to express facts about the enterprise that we are modeling with our database [16].

### 1.4.1 Functional Dependency

Functional dependency is illustrated by an arrow. The arrow will point from  $A$  to  $B$  in the functional dependency if the value of  $A$  uniquely determines the value of  $B$  and  $A$  and  $B$  are the attribute or the set of attributes of a relation  $R$ . Basic concept and the theories of functional dependencies have been focussed in chapter four.

### 1.4.2 Multi Valued Dependency

Multi valued dependency is illustrated by a double-arrow. The double-arrow will point from  $A$  to  $B$  in the multi valued dependency if the value of  $A$  multi determines the value of  $B$  and  $A$  and  $B$  are the attribute or the set of attributes of a relation  $R$ . Theory of multi-valued dependencies have been discussed in chapter five.

## 1.5 Normalization based on Functional and Multi Valued Dependencies

A given set of functional and multi-valued dependencies can be used in designing a given database in which most of the undesirable properties do not occur. In designing such a system, it may become necessary to decompose a relation to a number of smaller relations. Several normal forms such as 3NF, BCNF, and 4NF ensure “good” database designs. A database is a good database if it is either in BCNF or in 3NF. If there is any multi-valued dependency, holds on the relation/relations of a database, which is already either in BCNF or in 3NF we would not say it is a good database since the presence of multi-valued dependency/dependencies would cause repetition of information. We have to decompose this database into smaller databases in 4NF, i.e., decomposed relation/relations would not have any multi-valued dependency. Between 3NF and BCNF normal form BCNF is preferable, but in some cases decomposition towards BCNF causes loss of dependencies. In any case, we would not allow the loss of dependency. To avoid the loss of dependency we would prefer



the decomposition towards 3NF instead of BCNF when necessary [16]. We can use independent program for the decomposition of a relation towards 3NF, BCNF, and 4NF. The various algorithms for the characterization (decomposition) of databases using functional dependencies have been discussed in chapter five. Algorithm for the decomposition of a database using multi-valued dependencies has also been discussed in chapter five.

## **1.6 Machine Learning Algorithms for determining Functional and Multi Valued Dependencies**

Determining the set of functional and multi-valued dependencies those holds on the relations of a database is also a design issue. Definitely, the clue of finding the set of dependencies is not the data structure but the data itself. Collecting sufficient number of records (tuples) on a particular relation, the dependencies can be justified based on them. If a dependency is refuted by data we can say this dependency does not hold [9]. The process of refutation can be done using specific machine learning algorithms. Functional dependency and multi-valued dependency have separate algorithms both incremental and non-incremental. Machine learning theories for functional and multi-valued dependencies have been discussed in chapter six. The algorithms for the characterization of databases using machine learning techniques have also been discussed in chapter six.

## **1.7 Previous Work**

The first discussion of relational database design theory appeared in an early paper by Codd [1970]. During the 1970's a large number of dependencies and normal forms were introduced. Codd [1970] defined functional dependencies. Armstrong's axioms were introduced in Armstrong [1974]. Codd [1972a] introduced first, second and third normal forms.

BCNF was introduced in Codd [1972a]. The desirability of BCNF was discussed in Bernstein and Gooddman [1980b]. Beeri et al. [1977] gave a set of axioms for functional and multi valued dependencies and proved that their axioms are sound and complete. Biskup et al. [1979] gave the algorithm to find a loss less join dependency preserving decomposition into 3NF.

Maier [1983] presented the theory of relational databases in detail. Ullman [1982a] presented a more theoretic coverage of many of the normal forms.

Flach [1990] discussed Machine Learning theories for functional and multi valued dependencies of relational databases and gave Machine Learning algorithms for finding these dependencies.

## **1.8 Scope of the Work**

Designing a good relational database for a particular system is still a challenging job for database designer. The designer must have expert knowledge on relational database theories, functional and multi valued dependencies and normalization. Different design methodologies practiced at present do not use machine learning techniques. After having extensive investigation it has become evident that machine learning techniques would certainly be a useful tool in designing good relational databases and the designer would not necessary to be a theoretical expert. And the design effort would reduce merely to collect attributes and the values on the attributes from a particular system.

## **1.9 Proposed Design Framework and new Algorithms**

In the proposed design Framework designers have to collect all the attributes and the values on these attributes from the system. Machine learning software would find the set of functional and multi valued dependencies among the attributes. Taking these sets of dependencies as input software will give good relational database as output. New algorithms have been proposed to get relational database in 2NF using the set of functional and multi valued dependencies. Chapter seven discusses the complete design Frame work and new algorithms for the characterization of relational databases. Chapter eight shows the experimental results.

## Chapter 2

# Data Models

Underlying structure of a database is the concept of a data model, a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. The various data models that have been proposed fall into three different groups: object-based logical models, record-based logical models, and physical data models [16].

### 2.1 Object-Based Logical Models

Object-based logical models are used in describing data at the conceptual and view levels. They are characterized by the fact that they provide fairly flexible structuring capabilities and allow data constraints to be specified explicitly. There are many different models, and more are likely to come. Some of the more widely known ones are:

- The entity-relationship model.
- The object-oriented model
- The binary model.
- The semantic data model.
- The info logical model
- The functional data model.

The entity-relationship model and the object-oriented model represent the class of object-based logical models. The entity-relationship model has gained acceptance in database design and is widely used in practice. The object-oriented model includes many of the concepts of the entity-relationship model, but represents executable code as well as data. It is rapidly gaining acceptance in practice. Below are brief descriptions of both models.

#### 2.1.1 The Entity-Relationship Model

The entity-relationship (E-R) data model is based on a perception of a real world which consists of a collection of basic objects called entities, and relationships among these objects. An entity is an object that is distinguishable from other objects by a specific set of attributes.

For example, the attributes account number and balance describe one particular account in a bank. A relationship is an association-among several entities. For example, a *CustAcct* relationship associates a customer with each account that she or he has. The set of all entities of the same type and relationships of the same type are termed an entity set and relationship set, respectively.

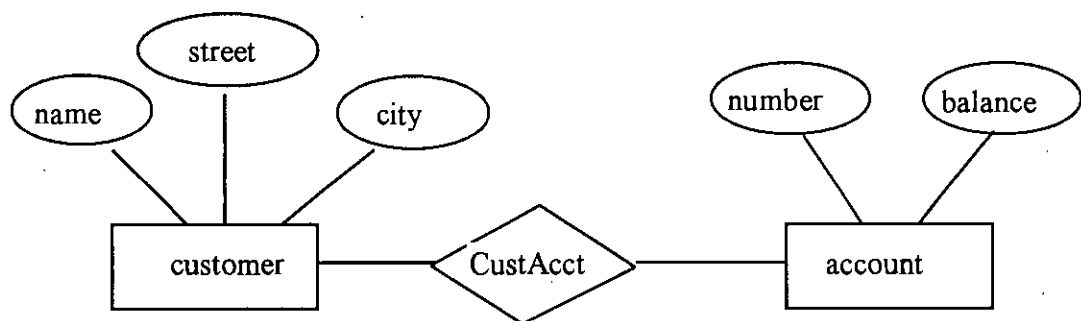
In addition to entities and relationships, the E-R model represents certain constraints to which the contents of a database must conform. One important constraint is mapping cardinalities, which express the number of entities to which another entity can be associated via a relationship set.

The overall logical structure of a database can be expressed graphically by an E-R diagram, which consists of the following components.

- Rectangles, which represent entity sets.
- Ellipses, which represent attributes.
- Diamonds, which represent relationships among entity sets.
- Lines, which link attributes to entity sets and entity sets to relationships.

Each component is labeled with the entity or relationship it represents.

The corresponding E-R diagram of a database banking system consisting of customers and accounts is shown in Figure 2.1



**Figure 2.1 A Sample E-R diagram.**

## 2.1.2 The Object-Oriented Model

Like the E-R model, the object-oriented model is based on a collection of objects. An object contains values stored in instance variables within the object. Unlike the record-oriented models, these values are themselves objects. Thus, objects contain objects to an arbitrary level of nesting. An object also contains bodies of code that operate on the object. These bodies of code are called methods.

Objects those contain the same types of values and methods are grouped together into classes. A class may be viewed as a type definition for objects. This combination of data and code into a type definition is similar to the programming language concept of abstract data types.

The only way in which one object can access the data of another object is by invoking a method of that object. This is called sending a message to the object. Thus, the call interface of the methods of an object defines its externally visible part. The internal part of the object, i.e., the instance variables and method code are not visible externally. This results in two levels of data abstraction.

To illustrate the concept, consider an object representing a bank account. Such an object contains instance variable number and balance, representing the account number and account balance. It contains a method pay-interest, which adds interest to the balance. Assume that the bank had been paying 6 percent interest on all accounts but now is changing its policy to pay 5 percent if the balance is less than \$1000 or 6 percent if the balance is \$1000 or greater. Under most data models, this would involve changing code in one or more application programs. Under the object-oriented model, the only change is made within the pay-interest method. The external interface to the object remains unchanged.

Unlike entities in the E-R model, each object has its own unique identity independent of the values it contains. Thus, two objects containing the same values are nevertheless distinct. The distinction among individual objects is maintained in the physical level through the assignment of distinct object identifiers.

## 2.2 Record-Based Logical Models

Record-based logical models are used in describing data at the conceptual and view levels. In contrast to object-based data models, they are used both to specify the overall logical structure of the database and to provide a higher-level description of the implementation.

Record-based models are so named because the database is structured in fixed-format records of several types. Each record type defines a fixed number of fields, or attributes, and each field is usually of a fixed length. The use of fixed-length records simplifies the physical-level implementation of the database. This is in contrast to many of the object-based models in which objects may contain other objects to an arbitrary depth of nesting. The richer structure of this database often leads to variable-length records at the physical level.

Record-based data models do not include a mechanism for the direct representation of code in the database. Instead, there are separate languages that are associated with the model to express database queries and updates. Some object-based models (including the object-oriented model) include executable code as an integral part of the data model itself.

The three most widely accepted data models are the relational, network, and hierarchical models. The relational model has gained favor over the other two in recent years. The network and hierarchical models, are still used in a large number of older databases.

### 2.2.1 Relational Model

The relational model represents data and relationships among data by a collection of tables, each of which has a number of columns with unique names. Figure 2.2 is a sample relational database showing customers and the accounts they have. It shows, for example, that customer Hodges lives at Sidehill in Brooklyn, and has two accounts, one numbered 647 with a balance of \$105,366, and the other numbered 801 with a balance of \$10,533. Note that customers Shiver and Hodges share account number 647 (they may share a business venture).

<i>name</i>	<i>Street</i>	<i>city</i>	<i>number</i>
Lowery	Maple	Queens	900
Shiver	North	Bronx	556
Shiver	North	Bronx	647
Hodges	Sidehill	Brooklyn	801
Hodges	Sidehill	Brooklyn	647

<i>Name</i>	<i>balance</i>
900	55
556	100000
647	105366
801	10533

Figure 2.2. A sample relational database.

## 2.2.2 Network Model

Data in the network model are represented by collections of records (in the Pascal or PL/I sense) and relationships among data are represented by links, which can be viewed as pointers. The records in the database are organized as collections of arbitrary graphs. Figure 2.3 presents a sample network database using the same information as in Figure 2.2.

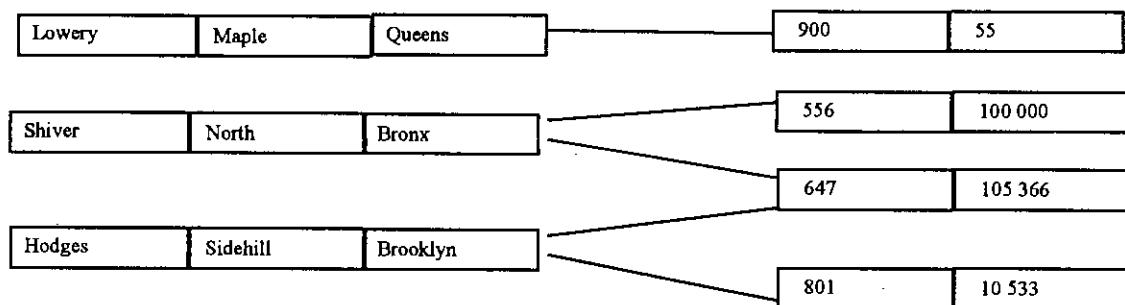


Figure 2.3 A sample of network database

### 2.2.3 Hierarchical Model

The hierarchical model is similar to the network model in the sense that data and relationships among data are represented by records and links respectively. It differs from the network model in that the records are organized as collections of trees rather than arbitrary graphs. Figure 2.4 presents a sample hierarchical database with the same information as in Figure 2.3.

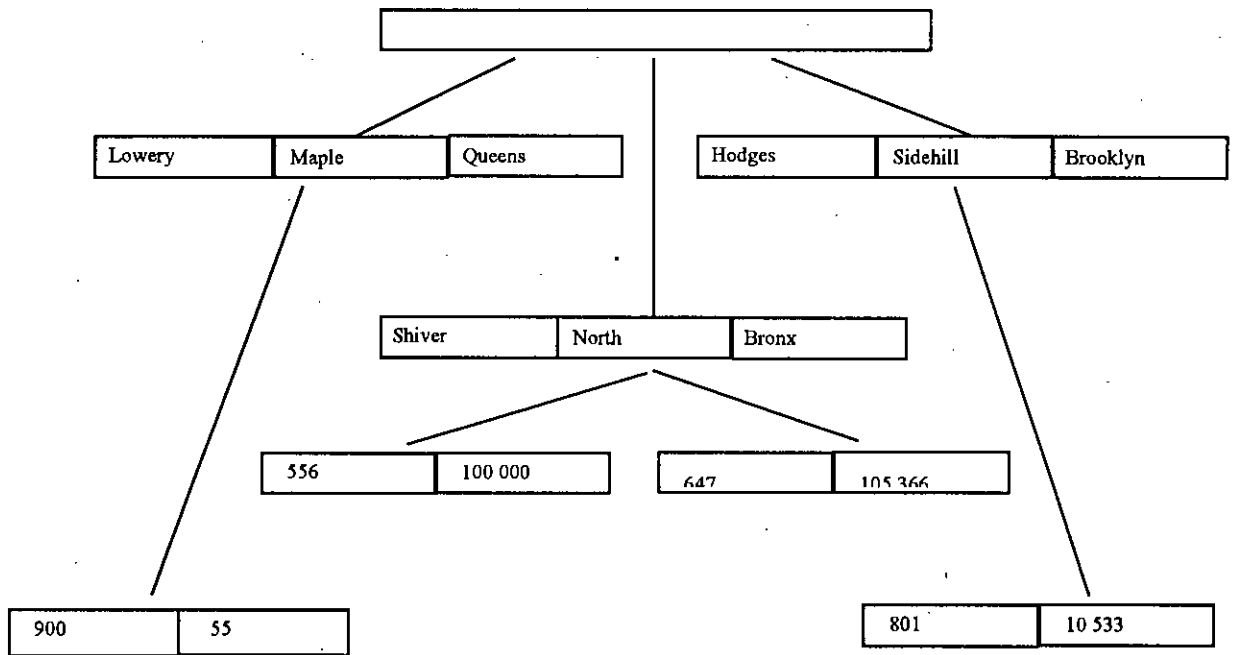


Figure 2.4 A sample of hierarchical database

### 2.3 Differences Between the Models

The relational model differs from the network and hierarchical models in that it does not use pointers or links. Instead, the relational model relates records by the values they contain. This freedom from the use of pointers allows a formal mathematical foundation to be defined.



## Chapter 3

# Structure of Relational Databases

A relational database consists of a collection of tables, each of which is assigned a unique name. A row in a table represents a relationship among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of table and the mathematical concept of relation, from which the relational data model takes its name [16].

### 3.1 Basic Structure

Consider the *deposit* table of Figure 3.1. It has four attributes: *branch-name*, *account-number*, *customer-name*, and *balance*. For each attribute, there is a set of permitted values, called the domain of that attribute. For the attribute *branch-name*, for example, the domain is the set of all branch names. Let  $D_1$  denote this set and let  $D_2$  denote the set of all account-numbers,  $D_3$  the set of all customer names, and  $D_4$  the set of all balances. Any row of *deposit* must consist of 4-tuples  $(v_1, v_2, v_3, v_4)$ , where  $v_1$  is a branch name (that is,  $v_1$  is in domain  $D_1$ ),  $v_2$  is an account number (that is,  $v_2$  is in domain  $D_2$ ),  $v_3$  is a customer name (that is,  $v_3$  is in domain  $D_3$ ), and  $v_4$  is a balance (that is,  $v_4$  is in domain  $D_4$ ). In general, *deposit* will contain only a subset of the set of all possible rows. Therefore *deposit* is a subset of

$$D_1 \times D_2 \times D_3 \times D_4$$

In general, a table of  $n$  columns must be a subset of

$$D_1 \times D_2 \times \dots \times D_{n-1} \times D_n$$

Mathematicians define a relation to be a subset of a Cartesian product of a list of domains. This corresponds exactly with our definition of table. The only difference is that we have assigned names to attributes, whereas mathematicians rely on numeric “names,” using the integer 1 to denote the attribute whose domain appears first in the list of domains, 2 for the

attribute whose domain appears second, and so on. Because tables are essentially relations, we can use the mathematical terms relation and tuple in place of the terms table and row.

In the *deposit* relation of Figure 3.1, there are eight-tuples. Let the tuple variable  $t$  refer to the first tuple of the relation. We use the notation  $t[\textit{branch-name}]$  to denote the value of  $t$  on the *branch-name* attribute. Thus,

<i>branch-name</i>	<i>Account-number</i>	<i>customer-name</i>	<i>Balance</i>
Downtown	101	Johnson	500
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Round Hill	305	Turner	350
Perryridge	201	Williams	900
Redwood	222	Lindsay	700
Brighton	217	Green	750
Downtown	105	Green	850

**Figure 3.1** The *deposit* relation.

$t[\textit{branch-name}] = \text{“Downtown”}$ . Similarly,  $t[\textit{account-number}] = 101$ , the value of  $t$  on the *account-number* attribute. *Customer-name* and *balance* follow suit. Alternatively, we may write  $t[1]$  to denote the value of tuple  $t$  on the first attribute (*branch-name*),  $t[2]$  to denote *account-number*, and so on. Since a relation is a set of tuples, we use the mathematical notation  $t \in r$  to denote that tuple  $t$  is in relation  $r$ .

We shall require that for all relations  $r$ , the domains of all attributes of  $r$  be atomic. A domain is atomic if elements of the domain are considered to be indivisible units. For example, the set of integers is an atomic domain, but the set of all sets of integers is a non-atomic domain. The distinction is that we do not normally consider integers to have subparts, but we consider sets of integers to have subparts, namely, the integers comprising the set. The important issue is not the domain itself, but the way we use domain elements in our database. The domain of

all integers would be non-atomic if we considered each integer to be an ordered list of digits. We would use atomic domains.

## 3.2 Database Scheme

When we talk about a database, we must differentiate between the database scheme, or, the logical design of the database, and a database instance, which is the data in the database at a given instant in time.

The concept of a relation scheme corresponds to the programming language notion of type definition. A variable of a given type has a particular value at a given instant in time. Thus, a variable in programming languages corresponds to the concept of an instance of a relation.

It is convenient to give a name to a relation scheme, just as we give names to type definitions in programming languages. We adopt the convention of using lower case names for relations and names beginning with an uppercase letter for relation schemes. Following this notation, we use *Deposit-scheme* to denote the relation scheme for relation *deposit*. Thus,

$$\textit{Deposit-scheme} = (\textit{branch-name}, \textit{account-number}, \textit{customer-name}, \textit{balance})$$

We denote the fact that *deposit* is a relation on the scheme *Deposit* by

$$\textit{deposit} (\textit{Deposit-scheme})$$

In general, a relation scheme is a list of attributes and their corresponding domains. To define domains for the relation *deposit*, we can use the notation

$$(\textit{branch-name} : \textit{string}, \textit{account-number} : \textit{integer}, \\ \textit{customer-name} : \textit{string}, \textit{balance} : \textit{integer})$$

As another example, consider the *customer* relation of Figure 3.2. The scheme for that relation is

*Customer-scheme = (customer-name, street, customer-city)*

<i>Customer-name</i>	<i>Street</i>	<i>customer-city</i>
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford

**Figure 3.2.** The *customer* relation.

Note that the attribute *customer-name* appears in both relation schemes. This is not a coincidence. Rather, the use of common attributes in relation schemes is one way of relating tuples of distinct relations. For example, suppose we wish to find the cities where depositors of the Perryridge branch live. We look first at the *deposit* relation to find all depositors of the Perryridge branch. Then, for each such customer, we would look in the *customer* relation to find the city in which he or she lives. Using the terminology of the entity-relationship model, we say that the attribute *customer-name* represents the same entity set in both relations.

It would appear that, for our banking example, we could have just one relation scheme rather than several. That is, it may be easier for a user to think in terms of one relation scheme rather than several. Suppose we used only one relation for our example, with scheme

*Account-info-scheme = (branch-name, account-number, customer-name, balance, street, customer-city).*

Observe that if a customer has several accounts, we must list her or his address once for each account. That is, we must repeat certain information several times. This repetition is wasteful and is avoided by the use of two relations, as in our example.

In addition, if a customer has one or more accounts, but has not provided an address, we cannot construct a tuple on *Account-info-scheme*, since the values for *street* and *customer-city* are not known. To represent incomplete tuples, we must use null values. Thus, in the above example, the values for *street* and *customer-city* must be null. By using two relations, one on *Customer-scheme* and one on *Deposit-scheme*, we can represent customers whose address is unknown, without using null values. We simply use a tuple on *Deposit-scheme* to represent the information until the address information becomes available.

It is not always possible to eliminate null values. Suppose, for example, that we include the attribute *phone-number* in the *Customer-scheme*. It may be that a customer does not have a phone number, or that the phone number is unlisted. We would then have to resort to null values to signify that the value is unknown or does not exist. Null values cause a number of difficulties in accessing or updating the database, and thus should be eliminated if possible.

### 3.3 Keys

A super-key is a set of one or more attributes which, taken collectively, allow us to identify uniquely a tuple in a relational scheme. For example, the *social-security* attribute of the *customer* relation in Figure 3.3 is sufficient to distinguish one customer tuple from another. Thus *social-security* is a super-key. Similarly, the combination of *customer-name* and *social-security* is a super-key for the *customer* relation. The *customer-name* attribute is not a super-key, as several people might have the same name. The concept of super-key is not sufficient for our purpose, since, as we saw above, a super-key may contain extraneous attributes. If  $K$  is a super-key, then so is any superset of  $K$ . We are often interested in super-keys for which no proper subset is a super-key. Such minimal super-keys are called candidate keys.

It is possible that several distinct sets of attributes could serve as a candidate key. For example a combination of *customer-name* and *street* is sufficient to distinguish among tuples

of the *customer* relation. Then both  $\{social-security\}$  and  $\{customer-name, street\}$  are candidate keys. Although the attributes *social-security* and *customer-name* together can distinguish tuples of *customer* relation, their combination does not form a candidate key, since the attribute *social-security* alone is a candidate key.

Primary key is a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation scheme.

<i>Customer-name</i>	<i>social-security</i>	<i>street</i>	<i>Customer-city</i>
Oliver	654-32-1098	Main	Harrison
Harris	890-12-3456	North	Rye
Marsh	456-78-9012	Main	Harrison
Pepper	369-12-1518	North	Rye
Ratliff	246-80-1214	Park	Pitsfield
Brill	121-21-2121	Putnam	Stamford
Evers	135-79-1357	Nassau	Princeton

Figure 3.3 The *customer* relation

A banking enterprise can be assumed with the following schemes

*Branch-scheme* = (*branch-name*, *assets*, *branch-city*)

*Customer-scheme* = (*customer-name*, *street*, *customer-city*)

*Deposit-scheme* = (*branch-name*, *account-number*, *customer-name*, *balance*)

*Borrow-scheme* = (*branch-name*, *loan-number*, *customer-name*, *amount*)

The primary keys for the *Customer* and *Branch* entity sets are *customer-name* and *branch-name*, respectively.

Figures 3.4 and 3.6 show a sample *borrow* (*Borrow-scheme*) relation and a *branch* (*Branch-scheme*) relation, respectively.

In *Branch-scheme*,  $\{branch-name\}$  and  $\{branch-name, branch-city\}$  are both super-keys.  $\{branch-name, branch-city\}$  is not a candidate key, because  $\{branch-name\} \subseteq \{branch-$

*name, branch-city* and *{branch-name}* itself is a super-key. *{branch-name}*, however is a candidate key, which for our purpose will also serve as a primary key. The attribute *branch-city* is not a super-key, since two branches in the same city may have different names (and different assets figures). The primary key for *Customer-scheme* is *social-security* attribute.

<i>branch-name</i>	<i>loan-number</i>	<i>customer-name</i>	<i>Amount</i>
Downtown	17	Jones	1000
Redwood	23	Smith	2000
Perryridge	15	Hayes	1500
Downtown	14	Jackson	1500
Mianus	93	Curry	500
Round Hill	11	Turner	900
Pownal	29	Williams	1200
North Town	16	Adams	1300
Downtown	18	Johnson	2000
Perryridge	25	Glenn	2500
Brighton	10	Brooks	2200

Figure 3.4. The *borrow* relation.

<i>branch-name</i>	<i>Assets</i>	<i>branch-city</i>
Downtown	9000000	Brooklyn
Redwood	2100000	Palo Alto
Perryridge	1700000	Horseneck
Mianus	400000	Horseneck
Round Hill	8000000	Horseneck
Pownal	300000	Bennington
North Town	3700000	Rye
Brighton	7100000	Brooklyn

Figure 3.5 : The *branch* relation.

Let  $R$  be a relational scheme. If we say that a subset  $K$  of  $R$  is a super-key for  $R$ , we are restricting consideration to relations  $r(R)$  in which no two distinct tuples have the same values on all attributes in  $K$ . That is, if  $t_1$  and  $t_2$  are in  $r$  and  $t_1 \neq t_2$ , then  $t_1[K] \neq t_2[K]$ .

### 3.4 Query Languages

A query language is a language in which a user requests information from the database. These languages are typically at a higher level than standard programming languages. Query languages can be categorized as being either procedural or non-procedural. In a procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result. In a non-procedural language, the user describes the information desired without giving a specific procedure for obtaining that information.

Most commercial relational database systems offer a query language that includes elements of both the procedural and the non-procedural approaches. The relational algebra is procedural, while the tuple relational calculus and the domain relational calculus are non-procedural. These query languages are terse and formal, lacking the “syntactic sugar” of commercial languages; but they illustrate the fundamental techniques for extracting data from the database.

A complete data manipulation language includes not only a query language, but also a language for database modification. Such languages include commands to insert and delete tuples as well as commands to modify parts of existing tuples.



# Chapter 4

## Functional Dependency

### 4.1 Basic Concepts

Functional dependencies are one kind of constraint on the set of legal relations. They allow us to express facts about the enterprise that we are modeling with our database [16].

We have defined the notion of a super key as follows. Let  $R$  be a relation scheme. A subset  $K$  of  $R$  is a super key of  $R$  if, in any legal relation  $r(R)$ , for all pairs  $t_1$  and  $t_2$  of tuples in  $r$  such that  $t_1 \neq t_2$ ,  $t_1[K] \neq t_2[K]$ . That is, no two tuples in any legal relation  $r(R)$  may have the same value on attribute set  $K$ .

The notion of functional dependency generalizes the notion of super key. Let  $\alpha \subseteq R$  and  $\beta \subseteq R$ . The functional dependency  $\alpha \rightarrow \beta$  holds on  $R$  if for any legal relation  $r(R)$ , for all pairs of tuples  $t_1$  and  $t_2$  in  $r$  such that  $t_1[\alpha] = t_2[\alpha]$ , it is also the case the  $t_1[\beta] = t_2[\beta]$ .

Using the functional dependency notation, we say that  $K$  is a super key of  $R$  if  $K \rightarrow R$ . That is,  $K$  is a super key if whenever  $t_1[K] = t_2[K]$ , it is also case that  $t_1[R] = t_2[R]$  (that is,  $t_1 = t_2$ ).

Functional dependencies allow us to express constraints that cannot be expressed using super keys. Consider the scheme :

*Borrow-scheme = (branch-name, loan-number, customer-name, amount).*

If a given loan may be made to more than one customer (for example, to both of a husband/wife pair), then we would not expect the attribute loan number to be a super key. However, we do expect the functional dependency

*loan-number  $\rightarrow$  amount*

To hold, since we know that each loan-number is associated with precisely one amount.

We shall use functional dependencies in two ways :

1) To specify constraints on the set of legal relations. We shall thus concern ourselves only with relations that satisfy a given set of functional dependencies. If we wish to constrain ourselves to relations on scheme  $R$  that satisfy functional dependency  $F$ , we say that  $F$  holds on  $R$ .

2) To test relations to see if they are legal under a given set of functional dependencies. If a relation  $r$  is legal under a set  $F$  of functional dependencies, we say that  $r$  satisfies  $F$ .

Let us consider the relation  $r$  of Figure 4.1 and see which functional dependencies are satisfied. Observe that  $A \rightarrow C$  is satisfied. There are two tuples that have an  $A$ -value of  $a_1$ . These tuples have the same  $C$ -value, namely,  $c_1$ . Similarly, the two tuples with an  $A$ -value of  $a_2$  have the same

$A$	$B$	$C$	$D$
$a_1$	$b_1$	$c_1$	$d_1$
$a_1$	$b_2$	$c_1$	$d_2$
$a_2$	$b_2$	$c_2$	$d_2$
$a_2$	$b_3$	$c_2$	$d_3$
$a_3$	$b_3$	$c_2$	$d_4$

**Figure 4.1** Sample relation  $r$ .

$C$ -value,  $c_2$ . There are no other pairs of distinct tuples that have the same  $A$ -value. The functional dependency  $C \rightarrow A$  is not satisfied, however. To see this, consider the tuples  $t_1 = (a_2, b_3, c_2, d_3)$  and  $t_2 = (a_3, b_3, c_2, d_4)$ . These two tuples have the same  $C$ -value,  $c_2$ , but they have different  $A$ -values,  $a_2$  and  $a_3$ , respectively. Thus, we have found a pair of tuples  $t_1$  and  $t_2$  such that  $t_1[C] = t_2[C]$  but  $t_1[A] \neq t_2[A]$ .

Many other functional dependencies are satisfied by  $r$ , including, for example, the functional dependency  $AB \rightarrow D$ . Note that we use  $AB$  as a shorthand for  $\{A, B\}$ , to conform with standard practice. Observe that there is no pair of distinct tuples  $t_1$  and  $t_2$  such that  $t_1[AB] = t_2[AB]$ .

Therefore, if  $t_1[AB] = t_2[AB]$  it must be that  $t_1 = t_2$  and, thus,  $t_1[D] = t_2[D]$ . So,  $r$  satisfies  $AB \rightarrow D$ .

Some functional dependencies are said to be trivial because they are satisfied by all relations. For example,  $A \rightarrow A$  is satisfied by all relations involving attribute  $A$ . Reading the definition of functional dependency literally, we see that for all tuples  $t_1$  and  $t_2$  such that  $t_1[A] = t_2[A]$ , it is the case that  $t_1[A] = t_2[A]$ . Similarly,  $AB \rightarrow A$  is satisfied by all relations involving attribute  $A$ . In general, a functional dependency of the form  $\alpha \rightarrow \beta$  is trivial if  $\beta \subseteq \alpha$ .

In order to distinguish between the concepts of a relation satisfying a dependency and a dependency holding on a scheme, let us return to the banking example. If we consider, the *customer* relation (on *Customer-scheme*) as shown in Figure 4.2, we see that  $street \rightarrow customer-city$  is satisfied. However, we believe that, in the real world, two cities can have streets with the same name. Thus, it is possible, at some time, to have an instance of the *customer* relation in which  $street \rightarrow customer-city$  is not satisfied. So,

<i>Customer-name</i>	<i>street</i>	<i>customer-city</i>
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford

Figure 4.2 : The *customer* relation.

<i>Branch-name</i>	<i>loan-number</i>	<i>customer-name</i>	<i>Amount</i>
Downtown	17	Jones	1000
Redwood	23	Smith	2000
Perryridge	15	Hayes	1500
Downtown	14	Jackson	1500
Mianus	93	Curry	500
Round Hill	11	Turner	900
Pownal	29	Williams	1200
North Town	16	Adams	1300
Downtown	18	Johnson	2000
Perryridge	25	Glenn	2500
Brighton	10	Brooks	2200

Figure 4.3 : The *borrow* relation.

we would not include  $street \rightarrow customer-city$  in the set of functional dependencies that hold on *Customer-scheme*.

In the *borrow* relation (on *Borrow-scheme*) of Figure 4.3, we see that  $loan-number \rightarrow amount$  is satisfied. Unlike the case of *customer-city* and *street*, we do believe that the real-world enterprise that we are modeling requires each loan to have a unique amount. Therefore, we want to require that  $loan-number \rightarrow amount$  be satisfied by *borrow* relation all times. In other words, we require that the constraint  $loan-number \rightarrow amount$  holds on *Borrow-scheme*.

In the *branch* relation of Figure 4.4, we see that  $branch-name \rightarrow assets$  is satisfied, as is  $assets \rightarrow branch-name$ . We want to require that  $branch-name \rightarrow assets$  hold on *Borrow-*

*scheme*. However, we do not wish to require that  $assets \rightarrow branch\text{-}name$  hold, since it is possible to have several branches having the same assets value.

<i>Branch-name</i>	<i>assets</i>	<i>branch-city</i>
Downtown	9000000	Brooklyn
Redwood	2100000	Palo Alto
Perryridge	1700000	Horseneck
Mianus	400000	Horseneck
Round Hill	8000000	Horseneck
Pownal	300000	Bennington
North Town	3700000	Rye
Brighton	710000	Brooklyn

**Figure 4.4 : The *branch* relation.**

In what follows, we assume that when we design a relational database, we first list those functional dependencies-that must always hold. In the banking example, our list of dependencies includes the following :

- On *Branch-scheme*:

$branch\text{-}name \rightarrow branch\text{-}city$

$branch\text{-}name \rightarrow assets$

- On *Customer-scheme*

$customer\text{-}name \rightarrow customer\text{-}city$

$customer\text{-}name \rightarrow street$

- On *Borrow-scheme*

*loan-number* → *amount*

*loan-number* → *branch-name*

- On *Deposit-scheme*

*account-number* → *balance*

*account-number* → *branch-name*

## 4.2 Closure of a Set of Functional Dependencies

It is not sufficient to consider the given set of functional dependencies. Rather, we need to consider all functional dependencies that hold. We shall see that, given a set  $F$  of functional dependencies, we can prove that certain other functional dependencies hold. We say that such functional dependencies are logically implied by  $F$ .

Suppose we are given a relation scheme  $R = (A, B, C, G, H, I)$  and the set of functional dependencies

$A \rightarrow B$

$A \rightarrow C$

$CG \rightarrow H$

$CG \rightarrow I$

$B \rightarrow H$

The functional dependency

$A \rightarrow H$

is logically implied. That is, we can show that whenever our given set of functional dependencies holds,  $A \rightarrow H$  must also hold. Suppose that  $t_1$  and  $t_2$  are tuples such that

$$t_1[A] = t_2[A]$$

Since we are given that  $A \rightarrow B$ , it follows from the definition of functional dependency that

$$t_1[B] = t_2[B]$$

Then, since we are given that  $B \rightarrow H$ , it follows from the definition of functional dependency that

$$t_1[H] = t_2[H]$$

Therefore, we have shown that whenever  $t_1$  and  $t_2$  are tuples such that  $t_1[A] = t_2[A]$ , it must be that  $t_1[H] = t_2[H]$ . But that is exactly the definition of  $A \rightarrow H$ .

Let  $F$  be a set of functional dependencies. The closure of  $F$  is the set of all functional dependencies logically implied by  $F$ . We denote the closure of  $F$  by  $F^+$ . Given  $F$ , we can compute  $F^+$  directly from the formal definition of functional dependency. If  $F$  is large, this process would be lengthy and difficult. Such a computation of  $F^+$  requires arguments of the type given above to show that  $A \rightarrow H$  is in the closure of our example set of dependencies. There are simpler techniques for reasoning about functional dependencies.

The first technique is based on three axioms or rules of inference for functional dependencies. By applying these rules repeatedly, we can find all of  $F^+$  given  $F$ . In the rules below, we adopt the convention of using Greek letters ( $\alpha, \beta, \gamma, \dots$ ) for sets of attributes and uppercase Roman letters from the beginning of the alphabet for individual attributes. We use  $\alpha\beta$  to denote  $\alpha \cup \beta$ .

- Reflexivity rule. If  $\alpha$  is a set of attributes and  $\beta \subseteq \alpha$ , then  $\alpha \rightarrow \beta$  holds.
- Augmentation rule. If  $\alpha \rightarrow \beta$  holds and  $\gamma$  is a set of attributes, then  $\gamma\alpha \rightarrow \gamma\beta$  holds.
- Transitivity rule. If  $\alpha \rightarrow \beta$  holds, and  $\beta \rightarrow \gamma$  holds, then  $\alpha \rightarrow \gamma$  holds.

These rules are sound because they do not generate any incorrect functional dependencies. The rules are complete because for a given set  $F$  of functional dependencies, they allow us to generate all of  $F^+$ . This collection of rules is called Armstrong's axioms in honor of the person who first proposed them [2].

Although Armstrong's axioms are complete, it is tiresome to use them directly for the computation of  $F^+$ . To simplify matters further, we list some additional rules.

- Union rule. If  $\alpha \rightarrow \beta$  holds and  $\alpha \rightarrow \gamma$  holds, then  $\alpha \rightarrow \beta\gamma$  holds.
- Decomposition rule. If  $\alpha \rightarrow \beta\gamma$  holds, then  $\alpha \rightarrow \beta$  holds and  $\alpha \rightarrow \gamma$  holds.
- Pseudo-transitivity rule. If  $\alpha \rightarrow \beta$  holds and  $\gamma\beta \rightarrow \delta$  holds, then  $\alpha\gamma \rightarrow \delta$  holds.

Let us apply our rules to the example we presented earlier of scheme  $R = (A, B, C, G, H, I)$  and the set  $F$  of functional dependencies  $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$ . We list some members of  $F^+$  below.

- $A \rightarrow H$ . Since  $A \rightarrow B$  and  $B \rightarrow H$  holds, we apply the transitivity rule. Observe that it was much easier to use Armstrong's axioms to show that  $A \rightarrow H$  holds than it was to argue directly from the definitions as we did earlier.
- $CG \rightarrow HI$ . Since  $CG \rightarrow H$  and  $CG \rightarrow I$ , the union rule implies that  $CG \rightarrow HI$ .
- $AG \rightarrow I$ . We need several steps to show  $AG \rightarrow I$ . First, observe that  $A \rightarrow C$  holds. Using the augmentation rule, we see that  $AG \rightarrow CG$ . We are given that  $CG \rightarrow I$ , so by the transitivity rule  $AG \rightarrow I$  holds.

### 4.3 Closure of Attribute Sets

In order to test if a set  $\alpha$  is a super-key, we must devise an algorithm for computing the set of attributes functionally determined by  $\alpha$ . We shall see that such an algorithm is useful also as part of the computation of the closure of a set  $F$  of functional dependencies.

Let  $\alpha$  be a set of attributes. We call the set of all attributes functionally determined by  $\alpha$  under a set  $F$  of functional dependencies the closure of  $\alpha$  under  $F$  and denote it by  $\alpha^+$ . Figure 4.5 shows an algorithm, written in pseudo Pascal, to compute  $\alpha^+$ . The input is a set  $F$  of functional dependencies and the set  $\alpha$  of attributes. The output is stored in the variable result. This algorithm was presented by Korth & Silberschatz [1986] [15].

To illustrate how Algorithm 4.1 works, let us use the algorithm to compute  $(AG)^+$  with the functional dependencies defined above.

**Algorithm 4.1 : An Algorithm to compute  $\alpha^+$ , the closure of  $\alpha$  under  $F$ .**



**Input:** A set of functional dependencies  $F$ .

**Output:** Closure of  $\alpha$  under  $F$ , i.e.,  $\alpha^+$ .

```
proc compute( $F, \alpha^+$ )
     $result := \alpha$ ;
    while (changes to  $result$ ) do
        for each functional dependency  $\beta \rightarrow \gamma$  in  $F$  do
            if  $\beta \subseteq result$  then
                 $result := result \cup \gamma$ ;
            endif
        enddo
    enddo
endproc
```

We start with  $result = AG$ . The first time we execute the **while** loop to test each functional dependency we find that

- $A \rightarrow B$  causes us to include  $B$  in  $result$ . To see this, observe that  $A \rightarrow B$  is in  $F$ ,  $A \subseteq result$  (which is  $AG$ ), so  $result := result \cup B$ .
- $A \rightarrow C$  cause  $result$  to become  $ABCG$ .
- $CG \rightarrow H$  causes  $result$  to become  $ABCGH$ .
- $CG \rightarrow I$  causes  $result$  to become  $ABCGHI$ .

The second time we execute the while loop, no new attributes are added to  $result$  and the algorithm terminates.

Let us see why the Algorithm 4.1 is correct. The first step is correct since  $\alpha \rightarrow \alpha$  always holds (by the reflexivity rule). We claim that for any subset  $\beta$  of  $result$ , it is the case that  $\alpha \rightarrow \beta$ . Since we start the **while** loop with  $\alpha \rightarrow result$  being true, we can add  $\gamma$  to  $result$  only if  $\beta \subseteq result$  and  $\beta \rightarrow \gamma$ . But then  $result \rightarrow \beta$  by the reflexivity rule, so  $\alpha \rightarrow \beta$  by transitivity. Another application of transitivity shows that  $\alpha \rightarrow \gamma$  (using  $\alpha \rightarrow \beta$  and  $\beta \rightarrow \gamma$ ). The union rule implies that  $\alpha \rightarrow result \cup \gamma$ , so  $\alpha$  functionally determines any new result generated in the **while** loop. Thus, any attribute returned by the algorithm is in  $\alpha^+$ .

It is easy to see that the algorithm finds all of  $\alpha^+$ . If there is an attribute in  $\alpha^+$  not yet in *result*, then there must be a functional dependency  $\beta \rightarrow \gamma$  for which  $\beta \subseteq \textit{result}$  and at least one attribute in  $F$  is not in *result*.

It turns out that in the worst case his algorithm may take time quadratic in the size of  $F$ .

## 4.4 Canonical Cover

In order to minimize the number of functional dependencies that need to be tested in case of an update, we restrict a given set  $F$  of functional dependencies to a canonical cover  $F_c$ . A canonical cover for  $F$  is a set of dependencies such that  $F$  logically implies all dependencies in  $F_c$  and  $F_c$  logically implies all dependencies in  $F$ . Furthermore,  $F_c$  must have the following properties:

- Every functional dependency  $\alpha \rightarrow \beta$  in  $F_c$  contains no extraneous attributes in  $\alpha$ . Extraneous attributes are attributes that can be eliminated from  $\alpha$  without changing  $F_c^+$ . Thus  $A$  is extraneous in  $\alpha$  if  $A \in \alpha$  and  $F_c$  logically implies  $(F_c - \{\alpha \rightarrow \beta\}) \cup \{\alpha - A \rightarrow \beta\}$ .
- Every functional dependency  $\alpha \rightarrow \beta$  in  $F_c$  contains no extraneous attributes in  $\beta$ . Extraneous attributes are attributes that can be eliminated from  $\beta$  without changing  $F_c^+$ . Thus  $A$  is extraneous in  $\beta$  if  $A \in \beta$  and  $(F_c - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow \beta - A\}$  logically implies  $F_c$ .
- Each left side of a functional dependency in  $F_c$  is unique. That is, there are no two dependencies  $\alpha_1 \rightarrow \beta_1$  and  $\alpha_2 \rightarrow \beta_2$  in  $F_c$  such that  $\alpha_1 = \alpha_2$ .

To compute a canonical cover for  $F$ , use the union rule to replace any dependencies in  $F$  of the form  $\alpha_1 \rightarrow \beta_1$  and  $\alpha_1 \rightarrow \beta_2$  with  $\alpha_1 \rightarrow \beta_1\beta_2$ . Test each functional dependency  $\alpha \rightarrow \beta$  to see if there is an extraneous attribute in  $\alpha$ . For each dependency  $\alpha \rightarrow \beta$  see if there is an extraneous attribute in  $\beta$ . This process must be repeated until no changes occur in the loop.

Consider the following set  $F$  of functional dependencies on scheme  $(A, B, C)$ :

$$A \rightarrow BC$$

$$B \rightarrow C$$

$$A \rightarrow B$$

$$AB \rightarrow C$$

Let us compute the canonical cover for  $F$ .

- There are two functional dependencies with the same set of attributes on the left side of the arrow :

$$A \rightarrow BC$$

$$A \rightarrow B$$

We combine these into  $A \rightarrow BC$ .

- $A$  is extraneous in  $AB \rightarrow C$  because  $B \rightarrow C$  logically implies  $AB \rightarrow C$ , and thus  $(F - \{AB \rightarrow C\}) \cup \{B \rightarrow C\}$  logically implies  $F$ . As a result of removing  $A$  from  $AB \rightarrow C$ , we obtain  $B \rightarrow C$ , which is already in our set of functional dependencies.

At this point, our set of functional dependencies is:

$$A \rightarrow BC$$

$$B \rightarrow C$$

All the properties of a canonical cover are met by the above set of functional dependencies.

## Chapter 5

# Relational Database Design

In general, the goal of a relational database design is to generate a set of relation schemes that allow us to store information without unnecessary redundancy, yet allow us to retrieve information easily. One approach is to design schemes that are in an appropriate normal form. In order to determine whether a relation scheme is in one of the normal forms, we shall need additional information about the “real-world” enterprise that we are modeling with the database [16].

### 5.1 Pitfalls in Relational Database Design

Among the undesirable properties that a bad design may have are :

- Repetition of information.
- Inability to represent certain information.
- Loss of information

Below, we discuss these in greater detail using a banking example, with the following two relational schemes:

*Branch-scheme* = (*branch-name*, *assets*, *branch-city*)

*Borrow-scheme* = (*branch-name*, *loan-number*, *customer-name*, *amount*)

Figures 5.1 and 5.2 show an instance of the relations *branch* (*Branch-scheme*) and *borrow* (*Borrow-scheme*).

<i>branch-name</i>	<i>Assets</i>	<i>branch-city</i>
Downtown	9000000	Brooklyn
Redwood	2100000	Palo Alto
Perryridge	1700000	Horseneck
Mianus	400000	Horseneck
Round Hill	8000000	Horseneck
Pownal	300000	Bennington
North Town	3700000	Rye
Brighton	710000	Brooklyn

Figure 5.1 : Simple *branch* relation.

### 5.1.1 Representation of Information

Consider an alternative design for the bank database in which we replace *Branch-scheme* and *Borrow-scheme* with the single scheme:

*Lending-scheme* = (*branch-name*, *assets*, *branch-city*, *loan-number*,  
*customer-name*, *amount*)

Figure 5.3 shows an instance of the relation *lending* (*Lending-scheme*) produced by taking the natural join of the branch and borrow instances of Figures 5.1 and 5.2 . A tuple *t* in the *lending* relation has the following intuitive meaning:

<i>branch-name</i>	<i>Loan-number</i>	<i>customer-name</i>	<i>amount</i>
Downtown	17	Jones	1000
Redwood	23	Smith	2000
Perryridge	15	Hayes	1500
Downtown	14	Jackson	1500
Mianus	93	Curry	500
Round Hill	11	Turner	900
Pownal	29	Williams	1200
North Town	16	Adams	1300
Downtown	18	Johnson	2000
Perryridge	25	Glenn	2500
Brighton	10	Brooks	2200

Figure 5.2 : The borrow relation.

- $t[assets]$  is the asset figure for the branch named  $t[branch-name]$ .
- $t[branch-city]$  is the city in which the branch named  $t[branch-name]$  is located.
- $t[loan-number]$  is the number assigned to a loan made by the branch named  $t[branch-name]$  to the customer named  $t[customer-name]$ .
- $t[amount]$  is the amount of the loan whose number is  $t[loan-number]$ .

Suppose we wish to add a new loan to our database. Assume the loan is made by the Perryridge branch to Turner in the amount of \$1500. Let the loan-number be 31. In our original design, we would add the tuple

(Perryridge, 31, Turner, 1500)

to the *borrow* relation. Under the alternative design, we need a tuple with values on all the attributes of *Lending-scheme*. Thus, we must repeat the asset and city data for the Perryridge branch and add the tuple

(Perryridge, 1700000, Horseneck, 31, Turner, 1500)

to the *lending* relation. In general, the asset and city data for a branch must appear once for each loan made by that branch.

The repetition of information required by the use of our alternative design is undesirable. Repeating information wastes space. Furthermore,

<i>Branch-name</i>	<i>Assets</i>	<i>Branch-name</i>	<i>loan-number</i>	<i>customer-name</i>	<i>amount</i>
Downtown	9000000	Downtown	17	Jones	1000
Redwood	2100000	Redwood	23	Smith	2000
Perryridge	1700000	Perryridge	15	Hayes	1500
Downtown	9000000	Downtown	14	Jackson	1500
Mianus	400000	Mianus	93	Curry	500
Round Hill	8000000	Round Hill	11	Turner	900
Pownal	300000	Pownal	29	Williams	1200
North Town	3700000	North Town	16	Adams	1300
Downtown	9000000	Downtown	18	Johnson	2000
Perryridge	1700000	Perryridge	25	Glenn	2500
Brighton	7100000	Brighton	10	Brooks	2200

Figure 5.3 : *branch X borrow*.

the repetition of information complicates updating the database. Suppose, for example, that the Perryridge branch moves from Horseneck to Newtown. Under our original design, one tuple of the *branch* relation needs to be changed. Under our alternative design, many tuples of the *lending* relation need to be changed. Thus, updates are more costly under the alternative design than under the original design. When we perform the update in the alternative database, we must ensure that every tuple pertaining to the Perryridge branch is updated, or else our database will show two cities for the Perryridge branch.

The above observation is central to understanding why the alternative design is bad. We know that a bank branch is located in exactly one city. On the other hand, we know that a branch may make many loans. In other words, the functional dependency

*branch-name* → *branch-city*

holds on *Lending-scheme*, but we do not expect that the functional dependency *branch-name* → *loan-number* holds. The fact that a branch is located in a city and the fact that a branch makes a loan are independent and, as we have seen, these facts are best represented in separate relations. We shall see that functional dependencies can be used for specifying formally when a database design is good.

Another problem with the *Lending-scheme* design is that we cannot represent directly the information concerning a *branch* (*branch-name*, *assets*, *branch-city*) unless there exists at least one loan at the branch. This is because tuples in the *lending* relation require values for *loan-number*, *amount*, and *customer-name*.

One solution to this problem is to introduce null values. Recall, however, that null values are difficult to deal with. If we are not willing to deal with null values, then we can create the branch information only when the first loan application at that branch is made. Worse, we would have to delete this information when all the loans have been paid. Clearly this is undesirable, since under our original database design, the branch information would be available regardless of whether or not loans are currently maintained in the branch, and we could do so without resorting to the use of null values.

### 5.1.2 Loss of Information

The above example of a bad design suggests that we should decompose a relation scheme with many attributes into several schemes with fewer attributes. Careless decomposition, however, may lead to another form of bad design.

Consider an alternative design in which *Borrow-schemes* is decomposed into two schemes, *Amt-scheme* and *Loan-scheme*, as follows:



$Amt\text{-scheme} = (amount, customer\text{-name})$

$Loan\text{-scheme} = (branch\text{-name}, loan\text{-number}, amount)$

Using the *borrow* relation of Figure 5.2, we construct our new relations *amt* (*Amt-scheme*) and *loan* (*Loan-scheme*) as follows:

$$amt = \Pi_{amount, customer\text{-name}}(borrow)$$
$$loan = \Pi_{branch\text{-name}, loan\text{-number}, amount}(borrow)$$

We show the resulting *amt* and *loan* relations in Figure 5.4.

Of course, there are cases in which we need to reconstruct the *borrow* relation. For example, suppose that we wish to find those branches from which Jones has a loan. None of the relations in our alternative database contains this data. We need to reconstruct the *borrow* relation. It appears that we can do this by writing:

$$amt \times loan$$

Figure 5.5 shows the result of computing  $amt \times loan$ . When we compare this relation and the *borrow* relation with which we started (Figure 5.2), we notice some differences. Although every tuple that appears in *borrow* appears in  $amt \times loan$ , there are tuples in  $amt \times loan$  that are not in *borrow*. In our example,  $amt \times loan$  has the following additional tuples :

(Downtown, 14 Hayes, 1500)

(Perryridge, 15, Jackson, 1500)

(Redwood, 23, Johnson, 2000)

(Downtown, 18, Smith, 2000)

<i>Branch-name</i>	<i>loan-number</i>	<i>Amount</i>	<i>amount</i>	<i>customer-name</i>
Downtown	17	1000	1000	Jones
Redwood	23	2000	2000	Smith
Perryridge	15	1500	1500	Hayes
Downtown	14	1500	1500	Jackson
Mianus	93	500	500	Curry
Round Hill	11	900	900	Turner
Pownal	29	1200	1200	Williams
North Town	16	1300	1300	Adams
Downtown	18	2000	2000	Johnson
Perryridge	25	2500	2500	Glenn
Brighton	10	2200	2200	Brooks

Figure 5.4 : The relations *amt* and *laon*.

<i>Branch-name</i>	<i>loan-number</i>	<i>customer-name</i>	<i>amount</i>
Downtown	17	Jones	1000
Redwood	23	Smith	2000
Perryridge	15	Hayes	1500
Downtown	14	Jackson	1500
Mianus	93	Curry	500
Round Hill	11	Turner	900
Pownal	29	Williams	1200
North Town	16	Adams	1300
Downtown	18	Johnson	2000
Perryridge	25	Glenn	2500
Brighton	10	Brooks	2200
Downtown	14	Hayes	1500
Perryridge	15	Jackson	1500
Redwood	23	Johnson	2000
Downtown	18	Smith	2000

Figure 5.5 : The relation *amt*  $\times$  *loan*.

Consider the query, “Find those branches from which Hayes has a loan.” If we look back at Figure 5.2, we see that Hayes has only one loan, and that loan is from the Perryridge branch. However, when we apply the expression.

$$\Pi_{branch-name}(\sigma_{customer-name = \text{“Hayes”}}(amt \times loan))$$

We obtain two branch names: Perryridge and Downtown.

Let us examine this example more closely. If several loans happen to be in the same amount, we cannot tell which customer has which loan. Thus, when we join *amt* and *loan*, we obtain not only the tuples we had originally in *borrow*, but also several additional tuples. Although we have more tuples in *amt*  $\times$  *loan*, we actually have less information. We are no longer able, in general, to represent in the database which customers are borrowers from which branch. Because of this loss of information, we call the decomposition of *Borrow-scheme* into *Amt-scheme* and *Loan-scheme* a lossy decomposition or a lossy join decomposition. A decomposition that is not a lossy join decomposition is referred to as a loss less join decomposition. It should be clear from our example that a lossy join decomposition is, in general, a bad database design.

Let us examine the decomposition more closely to see why it is lossy. There is one attribute in common between *Loan-scheme* and *Amt-scheme*:

$$Loan-scheme \cap Amt-scheme = \{amount\}$$

The only way we can represent a relationship between *branch-name* and *customer-name* is through *amount*. This is not adequate because many customers may happen to have loans in the same amount, yet they do not necessary have these loans from the same branches. Similarly, many customers may happen to have loans from the same branch, yet the amounts of their loans may be unrelated to one another.

Contrast this with *Lending-scheme*, which we discussed earlier. We argued that a better design would result if we decompose *Lending-scheme* into *Borrow-scheme* and *Branch-scheme*. There is one attribute is common between these two schemes:

$$\text{Branch-scheme} \cap \text{Borrow-scheme} = \{\text{branch-name}\}$$

Thus, the only way we can represent a relationship between, for example, *customer-name* and *assets* is through *branch-name*. The difference between this example and the example above is that the assets of a branch are the same regardless of the customer to which we are referring, while the lending branch associated with a certain loan amount does depend on the customer to which we are referring. For a given *branch-name* there is exactly one *assets* value and exactly one *branch-city*, while a similar statement cannot be made for *amount*. That is, the functional dependency

$$\text{branch-name} \rightarrow \text{assets}, \text{branch-city}.$$

holds, but *amount* does not functionally determine *branch-name*.

The notion of loss-less joins is central to much of relational database design. Therefore, we restate the above examples below more concisely and more formally. Let  $R$  be a relation scheme. A set of relation schemes  $\{R_1, R_2, \dots, R_n\}$  is a decomposition of  $R$  if:

$$R = R_1 \cup R_2 \cup \dots \cup R_n$$

That is,  $\{R_1, R_2, \dots, R_n\}$  is a decomposition of  $R$  if every attribute in  $R$  appears in at least one  $R_i$ , for  $1 \leq i \leq n$ . Let  $r$  be a relation on scheme  $R$ , and let  $r_i = \prod R_i(r)$  for  $1 \leq i \leq n$ . That is,  $\{r_1, r_2, \dots, r_n\}$  is the database that results from decomposing  $R$  into  $\{R_1, R_2, \dots, R_n\}$ . It is always the case that:

$$r \subseteq r_1 \times r_2 \times \dots \times r_n$$

To see this, consider a tuple  $t$  in relation  $r$ . When we compute the relations  $r_1, r_2, \dots, r_n$ , the tuple  $t$  gives rise to one tuple  $t_i$  in each  $r_i$ ,  $1 \leq i \leq n$ . These  $n$  tuples combine to regenerate  $t$  when we compute  $r_1 \times r_2 \times \dots \times r_n$ . Therefore, every tuple in  $r$  appears in  $r_1 \times r_2 \times \dots \times r_n$ .

In general,  $r \neq r_1 \times r_2 \times \dots \times r_n$ . To illustrate this, consider the earlier example in which:

$$n = 2$$

$R = \text{Borrow-scheme}$

$R_1 = \text{Amt-scheme}$

$R_2 = \text{Loan-scheme}$ .

$r$  = the relation shown in Figure 5.2.

$r_1$  and  $r_2$  = the relations shown in Figure 5.4.

$r_1 \times r_2$  = the relation shown in Figure 5.5

Note that the relations in Figures 5.2 and 5.5 are not the same.

In order to have a loss less join decomposition, we need to impose some constraints on the set of possible relations. We found that decomposition *Lending-scheme into Borrow-scheme and Branch-scheme* is loss-less because of the functional dependency *branch-name*→*assets*, *branch-city*.

Let  $C$  represent a set of constraints on the database. A decomposition  $\{R_1, R_2, \dots, R_n\}$  of a relation scheme  $R$  is a *loss less join* decomposition for  $R$  if for all relations  $r$  on scheme  $R$  that are legal under  $C$ :

$$r = \Pi R_1(r) \times \Pi R_2(r) \times \dots \times \Pi R_n(r)$$

We shall show how to test whether a decomposition is a loss less join decomposition in the next sections.

## 5.2 Normalization Using Functional Dependencies

A given set of functional dependencies can be used in designing a given database in which most of the undesirable properties discussed Section 5.1 do not occur. In designing such systems, it may become necessary to decompose a relation to a number of smaller relations. Using functional dependencies, we can define several normal forms, which represents “good” database designs. There are a large number of normal forms such as BCNF and 3NF.

## 5.2.1 Desirable Properties of Decomposition

The *Lending-scheme* scheme of Section 5.1.1:

$$\textit{Lending-scheme} = (\textit{branch-name}, \textit{assets}, \textit{branch-city}, \textit{loan-number}, \\ \textit{customer-name}, \textit{amount})$$

The set  $F$  of functional dependencies that we require to hold on *Lending-scheme* are :

$$\textit{branch-name} \rightarrow \textit{assets}, \textit{branch-city}$$
$$\textit{loan-number} \rightarrow \textit{amount}, \textit{branch-name}$$

As discussed in Section 5.1.1, the *Lending-scheme* is an example of a bad database design. Assume that we decompose it to the following three relations:

$$\textit{Branch-scheme} = (\textit{branch-name}, \textit{assets}, \textit{branch-city})$$
$$\textit{Loan-info-scheme} = (\textit{branch-name}, \textit{loan-number}, \textit{amount})$$
$$\textit{Customer-loan-scheme} = (\textit{customer-name}, \textit{loan-number})$$

We claim that this decomposition has several desirable properties, which we discuss below.

### 5.2.1.1 Loss less join Decomposition

In Section 5.1.2, we have argued that it is crucial when decomposing a relation into a number of smaller relations that the decomposition be loss-less. We claim that the above decomposition is indeed loss-less. To demonstrate this, we must first present a criterion for determining whether a decomposition is lossy.

Let  $R$  be a relation scheme and  $F$  a set of functional dependencies on  $R$ . Let  $R_1$  and  $R_2$  form a decomposition of  $R$ . This decomposition is a loss less join decomposition of  $R$  if at least one of the following functional dependencies are in  $F^+$ :

- $R_1 \cap R_2 \rightarrow R_1$

- $R_1 \cap R_2 \rightarrow R_2$

We now show that our decomposition of *Lending-scheme* is a loss less join decomposition by showing a sequence of steps that generate the decomposition. We begin by decomposing *Lending-scheme* into two schemes:

*Branch-scheme* = (*branch-name*, *assets*, *branch-city*)

*Borrow-scheme* = (*branch-name*, *loan-number*, *customer-name*, *amount*)

Since *branch-name*  $\rightarrow$  *assets*, *branch-city*, the augmentation rule for functional dependencies implies that:

*branch-name*  $\rightarrow$  *branch-name*, *assets*, *branch-city*.

Since *Branch-scheme*  $\cap$  *Loan-scheme* = {*branch-name*}, it follows that our initial decomposition is a loss less join decomposition.

Next, we decompose *Borrow-scheme* into:

*Loan-info-scheme* = (*branch-name*, *loan-number*, *amount*)

*Customer-loan-scheme* = (*customer-name*, *loan-number*)

This step results in a loss less join decomposition, since *loan-number* is a common attribute and *loan-number*  $\rightarrow$  *amount*, *branch-name*

### 5.2.1.2 Dependency Preservation

There is another goal in relational database design to be considered: dependency preservation. When an update is made to the database, the system should be able to check that the update will not create an illegal relation, that is, one that does not satisfy all of the given functional dependencies. In order to check updates efficiently, it is desirable to design relational database schemes that allow update validation without the computation of joins.

In order to decide whether joins must be computed, we need to determine what functional dependencies may be tested by checking each relation individually. Let  $F$  be a set of functional dependencies on a scheme  $R$  and let  $R_1, R_2, \dots, R_n$  be a decomposition of  $R$ . The restriction of  $F$  to  $R_i$  is the set  $F_i$  of all functional dependencies in  $F^+$  that include only attributes of  $R_i$ . Since all functional dependencies in a restriction involve attributes of only one relation scheme, it is possible to test satisfaction of such a dependency by checking only one relation.

The set of restrictions  $F_1, F_2, \dots, F_n$  is the set of dependencies that can be checked efficiently. We now must ask whether testing only restrictions is sufficient. Let  $F' = F_1 \cup F_2 \cup \dots \cup F_n$ .  $F'$  is a set of functional dependencies on scheme  $R$ , but, in general,  $F' \neq F$ . However, even if  $F' \neq F$ , it may be that  $F'^+ = F^+$ . If this is true, then every dependency in  $F$  is logically implied by  $F'$  and if we verify that  $F'$  is satisfied, we have verified that  $F$  is satisfied. We say that a decomposition having the property  $F'^+ = F^+$  is a dependency preserving decomposition. Algorithm 5.1 shows an algorithm for testing dependency preservation.

**Algorithm 5.1 : Algorithm for testing dependency preservation.**

**Input:** The set of restrictions  $F_1, F_2, \dots, F_n$  is the set of dependencies.

**Output:** True/False.

```

proc dep-preserv ( $F_1, F_2, \dots, F_n$  )
    compute  $F^+$ ;
    for each scheme  $R_i$  in  $D$  do
         $F_i$  = the restriction of  $F^+$  to  $R_i$ ;
    enddo
     $F' := \emptyset$ 
    for each restriction  $F_i$  do
         $F' = F' \cup F_i$ 
    enddo
    compute  $F'^+$ ;
    if ( $F'^+ = F^+$ ) then return (true)
        else return (false);
    endif
endproc

```



The input is a set  $D = \{R_1, R_2, \dots, R_n\}$  of decomposed relation schemes, and a set  $F$  of functional dependencies. This algorithm was presented by Korth & Silberschatz [1986] [15].

We can now show that our decomposition of *Lending-schemes* is dependency preserving. To see this, we consider each member of the set  $F$  of functional dependencies that we require to hold on *Lending-scheme* and show that each one can be tested in at least one relation in the decomposition.

- The functional dependency:  $branch\text{-}name \rightarrow assets, branch\text{-}city$  can be tested using *Branch-scheme* = (*branch-name*, *assets*, *branch-city*)
- The functional dependency:  $loan\text{-}number \rightarrow amount, branch\text{-}name$  can be tested using *Loan-info-scheme* = (*branch-name*, *loan-number*, *amount*).

As the above example shows, it is often easier not to apply the Algorithm 5.1 to test dependency preservation, since the first step, computation of  $F^+$ , takes exponential time.

### 5.2.1.3 Repetition of Information

The decomposition of *Lending-scheme* does not suffer from the problem of repetition of information, which we discussed in Section 5.1.1. In *Lending-scheme*, it was necessary to repeat the city and assets of a branch for each loan. The decomposition separates branch and loan data into distinct relations, thereby eliminating this redundancy. Similarly, if a single loan is made to several customers, we must repeat the amount of the loan once for each customer (as well as the city and assets of the branch). In the decomposition, the relation on scheme *Customer-loan-scheme* contains the *loan-number*, *customer-name* relationship, and no other scheme does. Therefore, we have one tuple for each customer for a loan only in the relation on *Customer-loan-scheme*. In the other relations involving *loan-number* (those on schemes *Loan-info-scheme* and *Customer-loan-scheme*), only one tuple per loan need appear.

Clearly, the lack of redundancy exhibited by our decomposition is desirable.

## 5.3 Boyce-Codd Normal Form

One of the more desirable normal forms we can obtain is Boyce-Codd normal form (BCNF). A relation scheme  $R$  is in BCNF if for all functional dependencies that hold on  $R$  of the form  $\alpha \rightarrow \beta$ , where  $\alpha \subseteq R$  and  $\beta \subseteq R$ , at least one of the following holds:

- $\alpha \rightarrow \beta$  is a trivial functional dependency (that is,  $\beta \subseteq \alpha$ ).
- $\alpha$  is a super-key for scheme  $R$ .

A database design is in BCNF if each member of the set of relation schemes comprising the design is in BCNF.

To illustrate this, let us consider the following relation schemes, and their respective functional dependencies:

- *Branch-scheme* = (*branch-name*, *assets*, *branch-city*)  
*branch-name*  $\rightarrow$  *assets*, *branch-city*
- *Customer-scheme* = (*customer-name*, *street*, *customer-city*)  
*customer-name*  $\rightarrow$  *street*, *customer-city*
- *Deposit-scheme* = (*branch-name*, *account-number*, *customer-name*, *balance*)  
*account-number*  $\rightarrow$  *balance*, *branch-name*
- *Borrow-scheme* = (*branch-name*, *loan-number*, *customer-name*, *amount*)  
*loan-number*  $\rightarrow$  *amount*, *branch-name*

We claim that *Customer-scheme* is in BCNF. To see this, note that a candidate key for the scheme is *customer-name*. The only nontrivial functional dependencies that hold on *Customer-scheme* have *customer-name* on the left side of the arrow. Since *customer-name* is a candidate key, functional dependencies with *customer-name* on the left side do not violate the definition of BCNF. Similarly, it can be easily shown that the relation scheme *Branch-scheme* is in BCNF.

The scheme, *Borrow-scheme*, however, is not in BCNF. First, note that *loan-number* is not a super-key for *Borrow-scheme* since we could have a pair of tuples representing a single loan made to two people, as:

(Downtown,44, Mr. Bill, 1000)

(Downtown, 44, Mrs. Bill, 1000)

Because we did not list functional dependencies that rule out the above case, *loan-number* is not a candidate key. However, the functional dependency *loan-number*→*amount* is nontrivial. Therefore, *Borrow-scheme* does not satisfy the definition of BCNF.

We claim that *Borrow-scheme* is not in a desirable form since it suffers from the repetition of information problem described in Section 5.1.1. To illustrate this, observe that if there are several customer names associated with a loan, in a relation on *Borrow-scheme*, then we are forced to repeat the branch name and the amount once for each customer. We can eliminate this redundancy by redesigning our database so that all schemes are in BCNF. One approach to this problem is to take the existing non-BCNF design as a starting point and decompose those schemes that are not in BCNF. Consider the decomposition of *Borrow-scheme* into two schemes:

*Loan-info-scheme* = (*branch-name*, *loan-number*, *amount*)

*Customer-loan-scheme* = (*customer-name*, *loan-number*)

This decomposition is a loss less join decomposition.

To determine whether these schemes are in BCNF, we need to determine what functional dependencies apply to them. In this example, it is easy to see that

*loan-number* → *amount*, *branch-name*

applies to *Loan-info-scheme*, and that only trivial functional dependencies apply to *Customer-loan-scheme*. Although *loan-number* is not a super-key of *Borrow-scheme*, it is a candidate key for *Loan-info-scheme*. Thus, both schemes of our decomposition are in BCNF.

It is now possible to avoid redundancy in the case where there are several customers associated with a loan. There is exactly one tuple for each loan in the relation *on Loan-info-scheme*, and one tuple for each customer of each loan in the relation *on Customer-loan-scheme*. Thus, we do not have to repeat the branch name and the amount once for each customer associated with a loan.

In order for the entire design for the bank example to be in BCNF, we must decompose *Deposit-scheme* in a manner similar to our decomposition

**Algorithm 5.2 : BCNF decomposition Algorithm.**

**Input:** Relation  $R$  and the set of functional dependency  $F$  on the relation  $R$ .

**Output :** New smaller relations(decomposed) in BCNF

**proc** decomp\_bcnf ( $R, F$ )

*result* := { $R$ };

*done* := false;

compute  $F^+$

**while** (not *done*) **do**

**if** (there is a scheme  $R_i$  in *result* that is not in BCNF) **then**

        let  $\alpha \rightarrow \beta$  be a nontrivial functional dependency that holds

        on  $R_i$  such that  $\alpha \rightarrow R_i$  is not in  $F^+$ , and  $\alpha \cap \beta = \emptyset$ ;

*result* := (*result* -  $R_i$ )  $\cup$  ( $R_i$  -  $\beta$ )  $\cup$  ( $\alpha, \beta$ );

**else** *done* := true;

**endif**

**enddo**

**endproc**

of *Borrow-scheme*. When we do this decomposition, we obtain the two schemes:

*Account-info-scheme* = (*branch-name, account-number, balance*),

*Customer-account-scheme* = (*customer-name, account-number*)

We are now able to state a general method to generate a collection of BCNF schemes. If  $R$  is not in BCNF, we can decompose  $R$  into a collection of BCNF schemes  $R_1, R_2, \dots, R_n$  using the Algorithm 5.2 which generates not only a BCNF decomposition but also a loss less join decomposition. This algorithm was presented by Korth & Silberschatz [1986] [15]. To see why our algorithm generates only loss less join decompositions, notice that when we replace a scheme  $R_i$  with  $(R_i - \beta)$  and  $(\alpha, \beta)$ , the dependency  $\alpha \rightarrow \beta$  holds, and  $(R_i - \beta) \cap (\alpha, \beta) = \alpha$ . Let us apply the BCNF decomposition algorithm to the *Lending-scheme* scheme that we used earlier as an example of a poor database design.

*Lending-scheme* = (*branch-name*, *assets*, *branch-city*, *loan-number*,  
*customer-name*, *amount*)

The set of functional dependencies that we require to hold on *Lending-scheme* is

*branch-name*  $\rightarrow$  *assets*, *branch-city*  
*loan-number*  $\rightarrow$  *amount*, *branch-name*

A candidate key for this scheme is {*loan-number*, *customer-name*}.

We can apply Algorithm 5.2 to the *Lending-scheme* example as follows:

- The functional dependency :

*branch-name*  $\rightarrow$  *assets*, *branch-city*

holds on *Lending-scheme*, but *branch-name* is not a super-key. Thus, *Lending-scheme* is not in BCNF. We replace *Lending-scheme* by

*Branch-scheme* = (*branch-name*, *branch-city*, *assets*)  
*Deposit-scheme* = (*branch-name*, *loan-number*, *customer-name*, *amount*)

- The only nontrivial functional dependencies that hold on *Branch-scheme* include *branch-name* on the left side of the arrow. Since *branch-name* is a key for *Branch-scheme*, the relation *Branch-scheme* is in BCNF.

- The functional dependency

$$\textit{loan-number} \rightarrow \textit{amount}, \textit{branch-name}$$

holds on *Deposit-scheme*, but *loan-number* is not a key for *Deposit-scheme*. We replace *Deposit-scheme* by

$$\textit{Loan-info-scheme} = (\textit{branch-name}, \textit{loan-number}, \textit{amount})$$

$$\textit{Customer-loan-scheme} = (\textit{customer-name}, \textit{loan-number})$$

- *Loan-info-scheme* and *Customer-loan-scheme* are in BCNF.

Thus, the decomposition of *Lending-scheme* results in the three relation schemes *Branch-scheme*, *Loan-info-scheme*, and *Customer-loan-scheme*, each of which is in BCNF. These relation schemes are the same as those used in Section 5.2.1. We have demonstrated in that section that the resulting decomposition is both a loss less join decomposition and a dependency preserving decomposition.

Not every BCNF decomposition is dependency preserving. To illustrate this, consider the relational scheme:

$$\textit{Banker-scheme} = (\textit{branch-name}, \textit{customer-name}, \textit{banker-name})$$

indicates that a customer has a “personal banker” in a particular branch. The set  $F$  of functional dependencies that we require to hold on the *Banker-scheme* is

$$\textit{banker-name} \rightarrow \textit{branch-name}$$

$$\textit{customer-name} \textit{branch-name} \rightarrow \textit{banker-name}$$

clearly, *Banker-scheme* is not in BCNF since *banker-name* is not a super-key.

If we apply Algorithm 5.2, we may obtain the following BCNF decomposition:

$Banker\text{-}branch\text{-}scheme = (banker\text{-}name, branch\text{-}name)$

$Customer\text{-}banker\text{-}scheme = (customer\text{-}name, banker\text{-}name)$

The decomposed schemes preserve only  $banker\text{-}name \rightarrow branch\text{-}name$  (and trivial dependencies) but the closure of  $\{banker\text{-}name \rightarrow branch\text{-}name\}$  does not include  $customer\text{-}name, branch\text{-}name \rightarrow banker\text{-}name$ . The violation of this dependency cannot be detected unless a join is computed.

To see why the decomposition of  $Banker\text{-}scheme$  into the schemes  $Banker\text{-}branch\text{-}scheme$  and  $Customer\text{-}banker\text{-}scheme$  is not dependency preserving, we apply Algorithm 5.1. We find that the restrictions  $F_1$  and  $F_2$  of  $F$  to each scheme are as follows (we show only a canonical cover):

$F_1 = \{banker\text{-}name \rightarrow branch\text{-}name\}$

$F_2 = \emptyset$  (only trivial dependencies hold on  $Customer\text{-}banker\text{-}scheme$ )

Thus, a canonical cover for the set  $F'$  is  $F_1$ .

It is easy to see that the dependency  $customer\text{-}name, branch\text{-}name \rightarrow banker\text{-}name$  is not in  $F'^*$  even though it is in  $F^*$ . Therefore,  $F'^* \neq F^*$  and the decomposition is not dependency preserving.

The above example demonstrates that not every BCNF decomposition is dependency preserving. Moreover, it demonstrates that it is not always possible to satisfy all the three design goals:

- BCNF
- Loss-less join
- Dependency preservation.

This is because, every BCNF decomposition of  $Banker\text{-}scheme$  must fail to preserve  $customer\text{-}name, branch\text{-}name \rightarrow banker\text{-}name$ .

## 5.4 Third Normal Form

In those cases where we cannot meet all the three design criteria, we abandon BCNF and accept a weaker normal form called third normal form (3NF). We shall see that it is always possible to find a loss less join, dependency preserving decomposition that is in 3NF.

BCNF requires that all nontrivial dependencies be of the form  $\alpha \rightarrow \beta$  where  $\alpha$  is a super-key. 3NF relaxes this constraint slightly by allowing nontrivial functional dependencies whose left side is not a super-key.

A relation scheme  $R$  is in 3NF if for all functional dependencies that hold on  $R$  of the form  $\alpha \rightarrow \beta$ , where  $\alpha \subseteq R$  and  $\beta \subseteq R$ , at least one of the following holds:

- $\alpha \rightarrow \beta$  is a trivial functional dependency.
- $\alpha$  is a super-key for  $R$ .
- Each attribute  $A$  in  $\beta$  is contained in a candidate key for  $R$ .

The definition of 3NF allows certain functional dependencies that are not allowed in BCNF. A dependency  $\alpha \rightarrow \beta$  that satisfies only the third condition of the 3NF definition is not allowed in BCNF though it is allowed in 3NF. These dependencies are called transitive dependencies.

If a relation scheme is in BCNF, then all functional dependencies are of the form “super-key determines a set of attributes,” or the dependency is trivial. Thus, a BCNF scheme cannot have any transitive dependencies at all. As a result, every BCNF scheme is also in 3NF, and BCNF is therefore a more restrictive constraint than 3NF.

Let us return to our *Banker-scheme* example (Section 5.2.2). We have shown that this relation scheme does not have a dependency preserving, loss less join decomposition into BCNF. This scheme, however, turns out to be in 3NF. To see that this is so, note that  $\{\textit{customer-name}, \textit{branch-name}\}$  is a candidate key for *Banker-scheme*, so the only attribute not contained in a candidate key for *Banker-scheme* is *banker-name*. The only nontrivial functional dependency of the form  $\alpha \rightarrow \textit{banker-name}$  is



*customer-name, branch-name* → *banker-name*.

Since {*customer-name, branch-name*} is a candidate key, this dependency does not violate the definition of 3NF.

Algorithm 5.3 is an algorithm for finding a dependency preserving, loss less join decomposition into 3NF. This algorithm was presented by Korth & Silberschatz [1986] [15]. The fact that each relation scheme  $R_i$  is in 3NF follows directly from our requirement that the set  $F$  of functional dependencies be in canonical form (Section 5.3.4). The algorithm ensures preservation of dependencies by building explicitly a scheme for each given dependency. It ensures that the decomposition is a loss less join decomposition by guaranteeing that at least one scheme contains a candidate key for the scheme being decomposed.

To illustrate Algorithm 5.3, consider the following extension to *the Banker-scheme* introduced in Section 5.2.2:

*Banker-info-scheme* = (*branch-name, customer-name, banker-name, office-number*)

**Algorithm 5.3: Dependency preserving, loss less join decomposition into 3NF.**

**Input:** Relation  $R$  and the set of functional dependency  $F$  on the relation  $R$ .

**Output :** New smaller relations(decomposed) in 3NF

**proc** *decomp\_3nf*(  $R, F$  )

$i = 0$ ;

**for each** functional dependency  $\alpha \rightarrow \beta$  in  $F$  **do**

**if** none of the schemes  $R_j, 1 \leq j \leq i$  contains  $\alpha \beta$  **then**

$i := i + 1$ ;

$R_i := \alpha \beta$ ;

**endif**

**if** none of the schemes  $R_j, 1 \leq j \leq i$  contains a candidate key for  $R$  **then**

$i := i + 1$ ;

$R_i :=$  any candidate key of  $R$ ;

**Endif**

```
enddo
return (R1, R2, . . . , Rn)
endproc
```

The main difference here is that we include the banker's office-number as part of the information. The functional dependencies for this relation scheme are :

*banker-name* → *branch-name, office-number*  
*customer-name, branch-name* → *banker-name*.

The for loop in the algorithm causes us to include the following schemes in our decomposition :

*Banker-office-scheme* = (*banker-name, office-number*)  
*Banker-scheme* = (*customer-name, branch-name, banker-name*)

Since *Banker-scheme* contains a candidate key for *Banker-info-scheme*, we have done with the decomposition process.

## 5.5 Comparison of BCNF and 3NF

We have seen two normal forms for relational database schemes: 3NF and BCNF. There is an advantage to 3NF in that we know that it is always possible to obtain a 3NF design without sacrificing a loss-less join or dependency preservation. Nevertheless, there is a disadvantage to 3NF. If we do not eliminate all transitive dependencies, it may be necessary to use null values to represent some of the possible meaningful relationships among data items, and there is the problem of repetition of information.

<i>customer-name</i>	<i>Banker-name</i>	<i>Branch-name</i>
Jones	Johnson	Perryridge
Smith	Johnson	Perryridge
Hayes	Johnson	Perryridge
Jackson	Johnson	Perryridge
Curry	Johnson	Perryridge
Turner	Johnson	Perryridge

**Figure 5.6 : An instance of *Banker-scheme*.**

To illustrate, consider again the *Banker-scheme* and its associated functional dependencies. Since  $banker-name \rightarrow branch-name$ , we may want to represent relationships between values for *banker-name*, and values for *branch-name* in our database. However, in order to do so, either there must be a corresponding value for *customer-name* or we must use a null value for the attribute *customer-name*.

The other difficulty with the *Banker-scheme* is repetition of information. To illustrate, consider an instance of *Banker-scheme* shown in Figure 5.6. Notice that the information indicating that Johnson is working at the Perryridge branch is repeated redundantly.

If we are forced to choose between BCNF and dependency preservation with 3NF, it is generally preferable to opt for 3NF. If we cannot test for dependency preservation efficiently, we either pay a high penalty in system performance or risk the integrity of the data in our database. Neither of these alternatives is attractive. With such alternatives, the limited amount of redundancy imposed by transitive dependencies allowed under 3NF is the lesser evil. Thus, we normally choose to retain dependency preservation and sacrifice BCNF.

To summarize the above discussion, we note that our goal for a relational database design is:

- BCNF

- Loss-less join.
- Dependency preservation.

If we cannot achieve this, we accept:

- 3NF.
- Loss-less join
- Dependency preservation.

## 5.6 Normalization Using Multi Valued Dependencies

There are relation schemes that are in BCNF which do not seem to be sufficiently normalized in the sense that they still suffer from the problem of repetition of information. Consider again our banking example. Let us assume that in some alternative design for the bank database scheme, we have the scheme:

$$BC\text{-scheme} = (loan\text{-number}, customer\text{-name}, street, customer\text{-city})$$

This is a non-BCNF scheme because of the functional dependency

$$customer\text{-name} \rightarrow street, customer\text{-city}$$

that we asserted earlier, and the fact that *customer-name* is not a key for *BC-scheme*. However, let us assume that our bank is attracting wealthy customers who have several addresses (say, a winter home and a summer home). Then, we no longer wish to enforce the functional dependency *customer-name*  $\rightarrow$  *street, customer-city*. If we remove this functional dependency, we find *BC-scheme* to be in BCNF with respect to our modified set of functional dependencies. Despite the fact that *BC-scheme* is now in BCNF, we still have the problem of repetition of information that we had earlier.

In order to deal with this, we must define a new form of constraint, called a multi-valued dependency. As we did for functional dependencies, we shall use multi-valued dependencies to define a normal form for relation schemes. This normal form, called fourth normal form

(4NF), is more restrictive than BCNF. We shall see that every 4NF scheme is also in BCNF, but there are BCNF schemes that are not in 4NF.

### 5.6.1 Multi-valued Dependencies

Functional dependencies rule out certain tuples from being in a relation. If  $A \rightarrow B$ , then we cannot have two tuples with the same  $A$  value but different  $B$  values. Multi-valued dependencies do not rule out the existence of certain tuples. Instead, they require that other tuples of a certain form be present in the relation. For this reason, functional dependencies sometimes are referred to as “equality-generating” dependencies and multi-valued dependencies are referred to as “tuple-generating” dependencies.

Let  $R$  be a relation scheme and let  $\alpha \subseteq R$  and  $\beta \subseteq R$ . The multi-valued dependency

	$\alpha$	$\beta$	$R - \alpha - \beta$
$t_1$	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
$t_2$	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
$t_3$	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
$t_4$	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$

Figure 5.7 : Tabular representation of  $\alpha \twoheadrightarrow \beta$ .

holds on  $R$  if in any legal relation  $r(R)$ , for all pairs of tuples  $t_1$  and  $t_2$  in  $r$  such that  $t_1(\alpha) = t_2(\alpha)$ , there exist tuples  $t_3$  and  $t_4$  in  $r$  such that:

$$t_1(\alpha) = t_2(\alpha) = t_3(\alpha) = t_4(\alpha),$$

$$t_3(\beta) = t_1(\beta),$$

$$t_3(R - \beta) = t_2(R - \beta),$$

$$t_4(\beta) = t_2(\beta),$$

$$t_4(R - \beta) = t_1(R - \beta),$$

This definition is less complicated than it appears. In Figure 5.7, we give a tabular picture of  $t_1$ ,  $t_2$ ,  $t_3$ , and  $t_4$ . Intuitively, the multi-valued dependency  $\alpha \twoheadrightarrow \beta$  says that the relationship between  $\alpha$  and  $\beta$  is independent of the relationship between  $\alpha$  and  $R - \beta$ . If the multi-valued dependency  $\alpha \twoheadrightarrow \beta$  is satisfied by all relations on scheme  $R$ , then  $\alpha \twoheadrightarrow \beta$  is a trivial multi-valued dependency on scheme  $R$ . Thus,  $\alpha \twoheadrightarrow \beta$  is trivial if  $\beta \subseteq \alpha$  or  $\beta \cup \alpha = R$ .

To illustrate the difference between functional and multi-valued dependencies, consider again the *BC-scheme*, and the relation *bc* (*BC-scheme*) of Figure 5.8. We must repeat the loan number once for each address a customer has, and we must repeat the address for each loan a customer has. This repetition is unnecessary since the relationship between a customer and his or her address is independent of the relationship between that customer and a loan. If a customer, say Smith, has a loan, say loan number 23, we want that loan to be associated with all of Smith's addresses. Thus, the relation of Figure 5.9 is illegal. To make this relation

<i>loan-number</i>	<i>customer-name</i>	<i>street</i>	<i>Customer-city</i>
23	Smith	North	Rye
23	Smith	Main	Manchester
93	Curry	Lake	Horseneck

Figure 5.8 : Relation *bc*, an example of redundancy in a BCNF relation.

<i>loan-number</i>	<i>customer-name</i>	<i>street</i>	<i>Customer-city</i>
23	Smith	North	Rye
27	Smith	Main	Manchester

Figure 5.9 : An illegal *bc* relation.

legal, we need to add the tuples (23, Smith, Main, Manchester) and (27, Smith, North, Rye) to the *bc* relation of Figure 5.9.

Comparing the above example with our definition of multi-valued dependency, we see that we want the multi-valued dependency

$$customer-name \twoheadrightarrow street, customer-city$$

to hold. (The multi-valued dependency  $customer-name \twoheadrightarrow loan-number$  will do as well.)

As was the case for functional dependencies, we shall use multi-valued dependencies in two ways:

- 1) To test relations to determine whether they are legal under a given set of functional and multi-valued dependencies
- 2) To specify constraints on the set of legal relations. We shall thus concern ourselves only with relations that satisfy a given set of functional and multi-valued dependencies.

Note that if a relation  $r$  fails to satisfy a given multi-valued dependency, we can construct a relation  $r'$  that does satisfy the multi-valued dependency by adding tuples to  $r$ .

## 5.6.2 Theory of Multi-valued Dependencies

As was the case for functional dependencies and 3NF and BCNF, we shall need to determine all the multi-valued dependencies that are logically implied by a given set of multi-valued dependencies.

We take the same approach here that we did earlier for functional dependencies. Let  $D$  denote a set of functional and multi-valued dependencies. The closure  $D^+$  of  $D$  is the set of all functional and multi-valued dependencies logically implied by  $D$ . As was the case for functional dependencies, we can compute  $D^+$  from  $D$  using the formal definitions of functional dependencies and multi-valued dependencies. However, it is usually easier to reason about sets of dependencies using a system of inference rules.

The following list of inference rules for functional and multi-valued dependencies is sound and complete. Recall that soundness means that the rules do not generate any dependencies that are not logically implied by  $D$ . Completeness means that the rules allow us to generate all dependencies in  $D^+$ . The first three rules are Armstrong's axioms.

- 1) Reflexivity rule. If  $\alpha$  is a set of attributes and  $\beta \subseteq \alpha$ , then  $\alpha \rightarrow \beta$  holds.
- 2) Augmentation rule. If  $\alpha \rightarrow \beta$  holds and  $\gamma$  is a set of attributes, then  $\gamma\alpha \rightarrow \gamma\beta$  holds.
- 3) Transitivity rule. If  $\alpha \rightarrow \beta$  holds and  $\beta \rightarrow \gamma$  holds, then  $\alpha \rightarrow \gamma$  holds.
- 4) Complementation rule. If  $\alpha \rightarrow \rightarrow \beta$  holds, then  $\alpha \rightarrow \rightarrow R - \beta - \alpha$  holds.
- 5) Multi-valued augmentation rule. If  $\alpha \rightarrow \rightarrow \beta$  holds and  $\gamma \subseteq R$  and  $\delta \subseteq \gamma$ , then  $\gamma\alpha \rightarrow \rightarrow \delta\beta$  holds.
- 6) Multi-valued transitivity rule. If  $\alpha \rightarrow \rightarrow \beta$  holds and  $\beta \rightarrow \rightarrow \gamma$  holds, then  $\alpha \rightarrow \rightarrow \gamma - \beta$  holds.
- 7) Replication rule. If  $\alpha \rightarrow \beta$  holds then  $\alpha \rightarrow \rightarrow \beta$ .
- 8) Coalescence rule. If  $\alpha \rightarrow \rightarrow \beta$  holds and  $\gamma \subseteq \beta$  and there is a  $\delta$  such that  $\delta \subseteq R$  and  $\delta \cap \beta = \emptyset$  and  $\delta \rightarrow \gamma$ , then  $\alpha \rightarrow \gamma$  holds.

Let  $R = (A, B, C, G, H, I)$  be a relation scheme. Suppose  $A \rightarrow \rightarrow BC$  holds. The definition of multi-valued dependencies implies that if  $t_1(A) = t_2(A)$  then there exist tuples  $t_3$  and  $t_4$  such that :

$$t_1(A) = t_2(A) = t_3(A) = t_4(A)$$

$$t_3(BC) = t_1(BC)$$

$$t_3(GHI) = t_2(GHI)$$

$$t_4(GHI) = t_1(GHI)$$

$$t_4(BC) = t_2(BC)$$

The complementation rule states that if  $A \rightarrow \rightarrow BC$  then  $A \rightarrow \rightarrow GHI$ . Observe that  $t_3$  and  $t_4$  satisfy the definition of  $A \rightarrow \rightarrow GHI$  if we simply change the subscripts.

We can provide similar justification for rules 5 and 6 using the definition of multi-valued dependencies.



Rule 7, the replication rule, involves functional and multi-valued dependencies. Suppose that  $A \twoheadrightarrow BC$  holds on  $R$ . If  $t_1(A) = t_2(A)$  and  $t_1(BC) = t_2(BC)$ , then  $t_1$  and  $t_2$  themselves serve as the tuples  $t_3$  and  $t_4$  required by the definition of the multi-valued dependency  $A \twoheadrightarrow BC$ .

Rules 8, coalescence rule, is the most difficult of the eight rules to verify.

We can simplify the computation of the closure of  $D$  by using the following rules, which can be proved using rules 1 to 8.

- Multi-valued union rule. If  $\alpha \twoheadrightarrow \beta$  holds and  $\alpha \twoheadrightarrow \gamma$  holds, then  $\alpha \twoheadrightarrow \beta\gamma$  holds.
- Intersection rule. If  $\alpha \twoheadrightarrow \beta$  holds and  $\alpha \twoheadrightarrow \gamma$  holds, then  $\alpha \twoheadrightarrow \beta \cap \gamma$  holds.
- Difference rule. If  $\alpha \twoheadrightarrow \beta$  holds and  $\alpha \twoheadrightarrow \gamma$  holds, then  $\alpha \twoheadrightarrow \beta - \gamma$  holds and  $\alpha \twoheadrightarrow \gamma - \beta$  holds.

Let us apply our rules to the following example. Let  $R = (A, B, C, G, H, I)$  with the following set of dependencies  $D$  given:

$$\begin{aligned} A &\twoheadrightarrow B \\ B &\twoheadrightarrow HI \\ CG &\twoheadrightarrow H \end{aligned}$$

We list some members of  $D^+$  below:

- $A \twoheadrightarrow CGHI$ : Since  $A \twoheadrightarrow B$ , the complementation rule (rule 4) implies that  $A \twoheadrightarrow R - B - A$ .  $R - B - A = CGHI$ , so  $A \twoheadrightarrow CGHI$ .
- $A \twoheadrightarrow HI$ : Since  $A \twoheadrightarrow B$  and  $B \twoheadrightarrow HI$ , the multi-valued transitivity rule (rule 6) implies that  $A \twoheadrightarrow HI - B$ . Since  $HI - B = HI$ ,  $A \twoheadrightarrow HI$ .
- $B \twoheadrightarrow H$ : To show this fact, we need to apply the coalescence rule (rule 8).  $B \twoheadrightarrow HI$  holds. Since  $H \subseteq HI$  and  $CG \twoheadrightarrow H$  and  $CG \cap HI = \emptyset$ , we satisfy the statement of the coalescence rule with  $\alpha$  being  $B$ ,  $\beta$  being  $HI$ ,  $\delta$  being  $CG$ , and  $\gamma$  being  $H$ . We conclude that  $B \twoheadrightarrow H$ .
- $A \twoheadrightarrow CG$ : We already know that  $A \twoheadrightarrow CGHI$  and  $A \twoheadrightarrow HI$ . By the difference rule,  $A \twoheadrightarrow CGHI - HI$ . Since  $CGHI - HI = CG$ ,  $A \twoheadrightarrow CG$ .

### 5.6.3 Fourth Normal Form

Let us return to our *BC-scheme* example in which the multi-valued dependency  $customer-name \twoheadrightarrow street, customer-city$  holds, but no nontrivial functional dependencies hold. We saw earlier that, although *BC-scheme* is in BCNF, it is not an ideal design since we must repeat a customer's address information for each loan. We shall see that we can use the given multi-valued dependency to improve the database design, by decomposing *BC-scheme* into a fourth normal form (4NF).

A relation scheme  $R$  is in 4NF with respect to a set  $D$  of functional and multi-valued dependencies if for all multi-valued dependencies in  $D^+$  of the form  $\alpha \twoheadrightarrow \beta$ , where  $\alpha \subseteq R$  and  $\beta \subseteq R$ , at least one of the following hold :

- $\alpha \twoheadrightarrow \beta$  is trivial multi-valued dependency.
- $\alpha$  is a super-key for scheme  $R$ .

A database design is in 4NF if each member of the set of relation schemes comprising the design is in 4NF.

Note that the definition of 4NF differs from the definition of BCNF only in the use of multi-valued dependencies instead of functional dependencies. Every 4NF scheme is in BCNF. To see that this is so, note that if a scheme  $R$  is not in BCNF, then there is a nontrivial functional dependencies  $\alpha \rightarrow \beta$  holding on  $R$ , where  $\alpha$  is not a super-key. Since  $\alpha \rightarrow \beta$  implies  $\alpha \twoheadrightarrow \beta$  (by the replication rule),  $R$  cannot be in 4NF.

The analogy between 4NF and BCNF applies to the algorithm for decomposing a scheme into 4NF. Algorithm 5.4 shows the 4NF decomposition algorithm. This algorithm was presented by Korth & Silberschatz [1986] [15]. It is identical to the Algorithm 5.2 (BCNF decomposition algorithm) except for the use of multi-valued instead of functional dependencies.

If we apply Algorithm 5.4 to *BC-scheme*, we find that *customer-name*  $\twoheadrightarrow$  *loan-number* is a nontrivial multi-valued dependency and *customer-name* is not a super-key for *BC-scheme*. Following the algorithm, we replace *BC-scheme* by two schemes:

*Customer-loan-scheme* = (*customer-name*, *loan-number*)

*Customer-scheme* = (*customer-name*, *street*, *customer-city*)

This pair of schemes which are in 4NF eliminates the problem we have encountered with the redundancy of *BC-scheme*.

**Algorithm 5.4 : 4NF decomposition Algorithm.**

**Input:** Relation *R* and the set of multi-valued dependency *F* on the relation *R*.

**Output :** New smaller relations(decomposed) in 4NF

**proc** decomp\_4nf (*R*, *F*)

*result* := {*R*};

*done* = false;

compute  $F^+$ ;

**while** (not *done*) **do**

**if** (there is a scheme  $R_i$  in *result* that is not in 4NF) **then**

        let  $\alpha \twoheadrightarrow \beta$  be a nontrivial multi-valued dependency that holds

        on  $R_i$  such that  $\alpha \twoheadrightarrow R_i$  is not in  $F^+$ , and  $\alpha \cap \beta = \emptyset$ ;

*result* := (*result* -  $R_i$ )  $\cup$  ( $R_i$  -  $\beta$ )  $\cup$  ( $\alpha$ ,  $\beta$ );

**else** *done* := true;

**endif**

**enddo**

**endproc**

As was the case when we were dealing solely with functional dependencies, we are interested also in decompositions that are loss less join decompositions and that preserve dependencies. The following fact about multi-valued dependencies and loss-less joins shows that the Algorithm 5.4 generates only loss less join decompositions:

- Let  $R$  be a relation scheme and  $D$  a set of functional and multi-valued dependencies on  $R$ . Let  $R_1$  and  $R_2$  form a decomposition of  $R$ . This decomposition is a loss less join decomposition of  $R$  if and only if at least one of the following multi-valued dependencies is in  $D^+$  :

$$R_1 \cap R_2 \twoheadrightarrow R_1$$

$$R_1 \cap R_2 \twoheadrightarrow R_2$$

We stated earlier that if  $R_1 \cap R_2 \twoheadrightarrow R_1$  or  $R_1 \cap R_2 \twoheadrightarrow R_2$ , then  $R_1$  and  $R_2$  are a loss less join decomposition of  $R$ . The above fact regarding multi-valued dependencies is a more general statement about loss-less joins. It says that for every loss less join decomposition of  $R$  into two schemes  $R_1$  and  $R_2$ , one of the two dependencies  $R_1 \cap R_2 \twoheadrightarrow R_1$  or  $R_1 \cap R_2 \twoheadrightarrow R_2$  must hold.

The question of dependency preservation when we have multi-valued dependencies is not as simple as for the case in which we have only functional dependencies. Let  $R$  be a relation scheme and let  $R_1, R_2, \dots, R_n$  be a decomposition of  $R$ . Recall that for a set  $F$  of functional dependencies, the restriction  $F_i$  of  $F$  to  $R_i$  is all functional dependencies in  $F^+$  that include only attributes of  $R_i$ . Now consider a set  $D$  of both functional and multi-valued dependencies. The restriction of  $D$  to  $R_i$  in the set  $D_i$  consisting of:

- All functional dependencies in  $D^+$  that include only attributes of  $R_i$
- All multi-valued dependencies of the form

$$\alpha \twoheadrightarrow \beta \cap R_i$$

Where  $\alpha \subseteq R_i$  and  $\alpha \twoheadrightarrow \beta$  is in  $D^+$ .

A decomposition of scheme  $R$  into schemes  $R_1, R_2, \dots, R_n$  is a dependency preserving decomposition with respect to a set  $D$  of functional and multi-value dependencies if every set of relations  $r_1(R_1), r_2(R_2), \dots, r_n(R_n)$  such that for all  $i, r_i$  satisfies  $D_i$ , there exists a relation  $r(R)$  that satisfies  $D$  and for which  $r_i = \Pi_{R_i}(r)$  for all  $i$ .

Let us apply the 4NF decomposition Algorithm 5.4 to our example of  $R = (A, B, C, G, H, I)$  with  $D = \{A \twoheadrightarrow B, B \twoheadrightarrow HI, CG \twoheadrightarrow H\}$ .

$r_1$ :

$A$	$B$
$a_1$	$b_1$
$a_2$	$b_2$

$r_2$ :

$C$	$G$	$H$
$c_1$	$g_1$	$h_1$
$c_2$	$g_2$	$h_2$

$r_3$ :

$A$	$I$
$a_1$	$i_1$
$a_2$	$i_2$

$r_4$ :

$A$	$C$	$G$
$a_1$	$c_1$	$G_1$
$a_2$	$c_2$	$G_2$

Figure 5.10 : Projection of relation  $r$  onto a 4NF decomposition of  $R$ .

We shall then test the resulting decomposition for dependency preservation.

$R$  is not in 4NF. Observe that  $A \twoheadrightarrow B$  is not trivial, yet  $A$  is not a super-key. Using  $A \twoheadrightarrow B$  in the first iteration of the **while** loop, we replace  $R$  with two schemes,  $(A, B)$  and  $(A, C, G, H, I)$ . It is easy to see that  $(A, B)$  is in 4NF since all multi-valued dependencies that hold on  $(A, B)$  are trivial. However, the scheme  $(A, C, G, H, I)$  is not in 4NF. Applying the functional dependency  $CG \twoheadrightarrow H$  (which follows from the given functional dependency  $CG \twoheadrightarrow H$  by the replication rule), we replace  $(A, C, G, H, I)$  by the two schemes  $(C, G, H)$  and  $(A, C, G, I)$ . Scheme  $(C, G, H)$  is in 4NF, but scheme  $(A, C, G, I)$  is not. To see that  $(A, C, G, I)$  is not in 4NF recall that we showed earlier that  $A \twoheadrightarrow HI$  is in  $D^+$ . therefore  $A \twoheadrightarrow I$  is in the

restriction of  $D$  to  $(A, C, G, I)$ . Thus, in a third iteration of the **while** loop, we replace  $(A, C, G, I)$  by two schemes  $(A, I)$  and  $(A, C, G)$ . The Algorithm then terminates and the resulting 4NF decomposition is  $\{(A, B), (C, G, H), (A, I), (A, C, G)\}$ .

This 4NF decomposition is not dependency preserving since it fails to preserve the multi-valued dependency  $B \twoheadrightarrow HI$ . Consider the relations of Figure 5.10. It shows the four relations that may result from the projection of a relation on  $(A, B, C, G, H, I)$  onto the four schemes of our decomposition. The restriction of  $D$  to  $(A, B)$  is  $A \twoheadrightarrow B$  and some trivial dependencies. It is easy to see that  $r_1$  satisfies  $A \twoheadrightarrow B$  because there is no pair of tuples with the same  $A$  value. Observe that  $r_2$  satisfies all functional and multi-valued dependencies since no two tuples in  $r_2$  have the same value on any attribute. A similar statement can be made for  $r_3$  and  $r_4$ . Therefore, the decomposed version of our database satisfies all the dependencies in the restriction of  $D$ . However, there is no relation  $r$  on  $(A, B, C, G, H, I)$  that satisfies  $D$  and decomposes into  $r_1, r_2, r_3$  and  $r_4$ . Figure 5.15 shows the relation  $r = r_1 \times r_2 \times r_3 \times r_4$ . Relation  $r$  does not satisfy  $B \twoheadrightarrow HI$ . Any relations containing  $r$  and satisfying  $B \twoheadrightarrow HI$  must include the tuple  $(a_2, b_1, c_2, g_2, h_1, i_1)$ . However,  $\Pi_{CGH}(s)$  includes a tuple  $(c_2, g_2, h_1)$  that is not in  $r_2$ . Thus, our decomposition fails to detect a violation of  $B \twoheadrightarrow HI$ .

We have seen that if we are given a set of multi-valued and functional dependencies, it is advantageous to find a database design that meets the three criteria of:

- 4NF.
- Dependency preservation.
- Loss-less join.

If all we have are functional dependencies, the first criterion is just BCNF.

We have seen also that it is not always possible to achieve all three of these criteria. We succeeded in finding such a decomposition for the bank example, but failed for the example of scheme  $R = (A, B, C, G, H, I)$ .

When we cannot achieve our three goals, we compromise on 4NF, and accept BCNF or even 3NF, if necessary to ensure dependency preservation.

## Chapter 6

# Characterizing a Database Relation

### 6.1 Preliminaries

As stated earlier a relation scheme  $R$  is a set of attributes  $\{A_1, \dots, A_n\}$ . Each attribute  $A_i$  has a domain  $D_i$ ,  $1 \leq i \leq n$ , consisting of values. Domains are assumed to be countably infinite. A tuple on  $R$  is a mapping  $t: R \rightarrow \cup_i D_i$  with  $t(A_i) \in D_i$ ,  $1 \leq i \leq n$ . The values of a tuple  $t$  are usually denoted as  $\langle t(A_1), \dots, t(A_n) \rangle$  if the order of attributes is understood. A relation on  $R$  are is set of tuple on  $R$ . We will only consider finite relations. Any expression that is allowed for attributes is extended, by a slight abuse of symbols, to sets of attributes, e.g., if  $X$  is a subset of  $R$ ,  $t(X)$  denotes the set  $\{t(A) \mid A \in X\}$ . We will not distinguish between an attribute  $A$  and a set  $\{A\}$  containing only one attribute. A set of values for a set of attributes  $X$  is called an  $X$ -value. In general, attributes are denoted by the uppercase letters (possibly subscripted) from the beginning of the alphabet, sets of attributes are denoted by uppercase letters (possibly subscripted) from the end of the alphabet; values of (sets of) attributes are denoted by corresponding lowercase letters. Relations are denoted by lowercase letters (possibly subscripted) such as  $n, p, q, r, u$ ; tuples are denoted by  $t_1, t_2, t_3, \dots$ . If  $X$  and  $Y$  are sets of attributes, their juxtaposition  $XY$  means  $X \cup Y$ . We employ the usual notation for expressions of relational algebra.

As defined above, a relation  $r$  on a relation scheme  $R = \{A_1, \dots, A_n\}$  can be viewed as an extensional of an  $n$ -ary predicate symbol  $r$ ; alternatively,  $r$  is a possible model of the open formula  $r(X_1, \dots, X_n)$ . Likewise, a relation that satisfies a dependency  $D$  is a model of  $D$ . If  $r$  is a model of a formula  $\emptyset$ , we write  $r \models \emptyset$ . this is called the model theoretic view of database [9].

### 6.2 Characterization by functional dependencies

As stated earlier relation  $r$  satisfies a functional dependencies  $X \rightarrow Y$  if  $t_1 \in r$  and  $t_2 \in r$  and  $t_1(X) = t_2(X)$  imply  $t_1(Y) = t_2(Y)$ . Put differently, for every  $X$ -value  $x$ , the relation  $\pi_Y(\sigma_{X=x}(r))$

contains at most one tuple. The characterization of  $r$  in terms of functional dependencies is  $FD(r) = \{X \rightarrow Y \mid r \text{ satisfies } X \rightarrow Y\}$ . Thus,  $r$  is a relation that satisfies all fds in  $FD(r)$  and no other, hence  $r$  is an Armstrong relation for  $FD(r)$ . A relation contradicts any fd that it does not satisfy [2].

The empty relation  $\emptyset$  and any one-tuple relation  $\{t\}$  satisfy all fds. The universal relation  $u$  (the Cartesian product  $X_i D_i$  of all domains of attributes) satisfies only trivial fds. While  $u$  is infinite, there are also finite relations  $q$  that satisfy only trivial fds, thus  $FD(q) = FD(u)$ . This follows directly from the existence of Armstrong relations for any (consistent) set of fds [2][4].

As an illustration, let  $r$  be the relation depicted in Table 6.1.

$A$	$B$	$C$
$a_1$	$b_1$	$c_1$
$a_2$	$b_2$	$c_1$
$a_3$	$b_3$	$c_2$
$a_4$	$b_3$	$c_2$

**Table 6.1. A relation satisfying some functional dependencies.**

Then  $r$  satisfies  $A \rightarrow B$ ,  $A \rightarrow C$  (hence,  $A$  is a key for  $r$ ) and  $B \rightarrow C$ , among others.  $FD(r)$  contains these three fds, plus every fd that is implied by them (such as  $A \rightarrow BC$ , and  $AC \rightarrow B$ ), including trivial fds  $X \rightarrow Y$  where  $X \supseteq Y$  (such as  $AC \rightarrow A$ ). The set  $\{A \rightarrow B, A \rightarrow C, B \rightarrow C\}$  appears to be a cover for  $FD(r)$ , i.e.,  $FD(r)$  is the deductive closure of this set. It can be easily shown that each element of a non-redundant cover can be written in the form  $X \rightarrow A$ , where  $A$  is a single attribute such that  $A \notin X$ . Thus, we can solve the characterization problem for fds if we have an algorithm that constructs a set of fds  $X \rightarrow A$  which is a cover for  $FD(r)$ . In the following, we restrict ourselves to fds with a single attribute on the right-hand-side. The following theorem provides the clue for a fd-characterization algorithm.

**THEOREM 6.1** (More general fds). If a relation  $r$  satisfies fd  $X \rightarrow A$ , then it also satisfies any fd  $Y \rightarrow A$  such that  $Y \supseteq X$ .



Proof. If  $r$  contradicts  $Y \rightarrow A$ , then it contains two tuples with equal  $Y$ -values but unequal  $A$ -values. But then these tuples also have equal  $X$ -values, hence  $r$  contradicts  $X \rightarrow A$ .

Thus, if a non-redundant cover contains two fds  $X \rightarrow A$  and  $Y \rightarrow A$ , then  $X \subset Y$ . We call an fd  $X \rightarrow A$  as general as an fd  $Y \rightarrow A$  iff  $X \subseteq Y$ . This terminology is justified by theorem 1 : any model of a more general fd is a model of a less general one, hence the former implies the latter. The relation of generality is a partial ordering on the set of fds (partitioning it into sets of fds with equal right-hand-sides). Theorem 1 shows that the set of fds satisfied by  $r$  is bounded from above by a set of most general fds, such that any fd more specific than one of these is also satisfied by  $r$ . Thus, it suffices to maintain this upper boundary in a characterization algorithm.

There are essentially two approaches for determining  $FD(r)$  for a given  $r$ : (i) start with the (empty) cover of  $FD(u)$ , the set of trivial fds, and add those fds that are also satisfied by  $r$ ; (ii) start with a cover of  $FD(\emptyset)$ , the set of all fds, and remove those fds that are contradicted by  $r$ . The former approach will be called an upward approach, because it starts with the set of most specific fds; likewise, the latter approach will be called a downward approach. Which approach will be more efficient depends on the actual number of fds satisfied by the relation.

There is an important difference, however, between testing for satisfaction and testing for contradiction: contradiction can always be reduced to two witnessing tuples [25]; this can easily be seen if fds are expressed in Horn form. More importantly, these two witnesses give information about how to specialize the refuted fd, as will be detailed below. For these reasons, we restrict attention to the downward approach.

The satisfaction of an fd  $X \rightarrow A$  by a relation  $r$  can be expressed in Horn clause form as [13] :

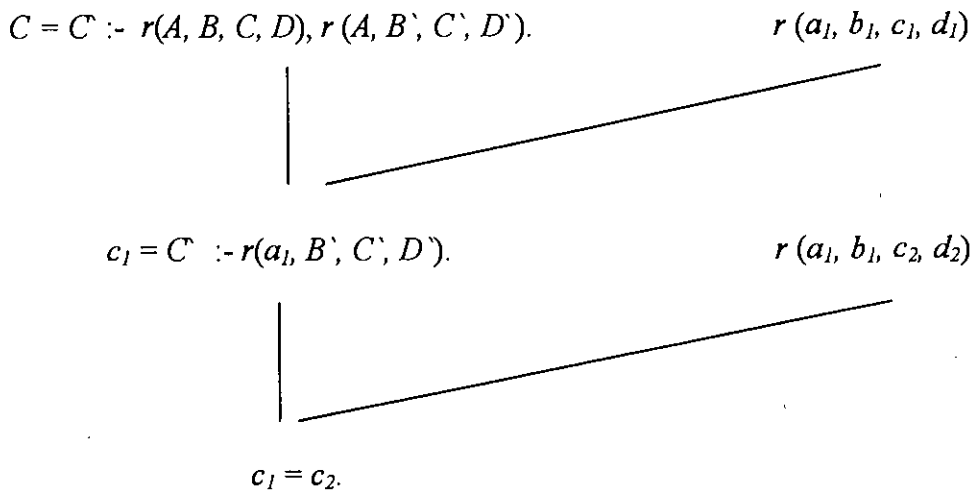
$$A = A' : - r_{XA}(X_1, \dots, X_n, A), r_{XA}(X_1, \dots, X_n, A') \quad (6.1)$$

where  $r_{XA}$  denotes the projection  $\pi_{XA}(r)$  of  $r$  on the attributes in  $X$  followed by  $A$ . The equality test for the  $A$ -values of both tuples can be implemented by syntactic unification, as in Prolog. For instance let  $R = \{A, B, C, D\}$ , then the fd  $A \rightarrow C$  is expressed as  $C = C' : - r_{AC}(A, C), r_{AC}(A, C')$

$C'$ ), or equivalently,  $C=C' :- r(A, B, C, D), r(A, B', C', D')$ . Thus, the general Horn form of a fd is  $A = A' :- Tuple1, Tuple2$ , and the test for contradiction are easily implemented in Prolog as

$$\text{fd\_contradicted}((A = A' :- Tuple1, Tuple2), Tuple1, Tuple2) :- \\ \text{tuple}(Tuple1), \text{tuple}(Tuple2), A \neq A'. \quad (6.2)$$

The goal  $?- \text{fd\_contradicted}(FD, Tuple1, Tuple2)$  succeeds if the fd  $FD$  is contradicted by the tuples  $Tuple1, Tuple2$ . For instance, a refutation of the satisfaction (and thus a proof of the contradiction) of the above fd by a relation containing the witnesses  $r(a_1, b_1, c_1, d_1)$  and  $r(a_1, b_1, c_2, d_2)$  is given in Figure 6.1.



**Figure 6.1. Refutation of the satisfaction of an fd with two tuples.**

The clause  $c_1 = c_2$  evaluates to false by definition.

Using this contradiction test, an algorithm for downward fd-characterization is given by Algorithm 6.1. This algorithm was presented by Flach [1990][9]. For simplicity, the right-hand-side attribute  $A$  is kept fixed. The algorithm leaves the way in which the relation is searched for two tuples contradicting an fd is left unspecified. Notice that a naive approach is easily implemented by calling  $\text{fd\_contradicted}(FD, Tuple1, Tuple2)$  with second and third argument non-instantiated. This goal will succeed with the first pair of tuples found to contradict  $FD$ , and on backtracking all other solutions will be generated. This approach is

naive in the sense that it investigates all  $n^2$  pairs of tuples, while only  $1/2n(n-1)$  pairs need to be investigated. The latter approach corresponds to calculating  $FD(r \cup \{t\})$  by first calculating  $FD(r)$ , followed by a comparison of  $t$  with each tuple in  $r$ . This is in fact equivalent to the inductive learning approach presented in section 6.4, which relies on the fact that  $FD(r) \supseteq FD(r \cup \{t\})$ .

**ALGORITHM 6.1. Downward characterization by functional dependencies.**

**Input:** A set  $r$  of tuples on a relational scheme  $R$  and an attribute  $A \in R$ .

**Output:** A non-redundant cover of the set of functional dependencies  $X \rightarrow A$ , satisfied by  $r$ .

```

proc fd_char( $r, A$ );
     $QUEUE := \{\emptyset \rightarrow A\}$ ;
     $FD\_SET := \emptyset$ ;
    while  $QUEUE \neq \emptyset$  do
         $FD =$  remove next fd from  $QUEUE$ ;
        for each pair  $T_1, T_2$  from  $r$  do
            if fd_contradicted( $FD, T_1, T_2$ )
                then add fd_specialize( $FD, T_1, T_2$ ) to  $QUEUE$ ;
            endif
        enddo
        If  $FD$  is not contradicted
            then  $FD\_SET := FD\_SET \cup FD$ ;
            endif
        enddo
    cleanup( $FD\_SET$ );
return ( $FD\_SET$ ).
endproc

```

The algorithm operates as follows. A *QUEUE* is maintained, fds still to be tested. for the first *FD* in the queue, a pair of contradicting witnesses is sought. If no such can be found, *FD* is satisfied by  $r$  and can be added to *FD\_SET*. If *FD* can be refuted, it is discarded and more specific fds are added to the queue. Finally, call cleanup(*FD\_SET*) removes those fds from *FD\_SET* that are subsumed by (less general than) others.

The procedure `fd_specialize` ( $FD, T_1, T_2$ ) should of course not jump over fds that are satisfied, but it should preferably jump over fds more special than  $FD$  that are also contradicted by  $\{T_1, T_2\}$ . As suggested above, this can be done. The key idea is, that if the fd  $A \rightarrow C$  is contradicted by the witnesses  $r(a_1, b_1, c_1, d_1)$  and  $r(a_1, b_1, c_2, d_2)$ , then we can immediately deduce from this refutation that  $AD \rightarrow C$  is a possible replacement, but  $AB \rightarrow C$  definitely is not. This conclusion can be reached by looking for attributes, apart from  $C$ , for which both witnesses have different values, i.e.,  $D$ . This set of attributes is called the disagreement of the two witnesses, and can be obtained by computing their anti-unification (the dual of unification) [22][23][24]. The anti-unification of  $r(a_1, b_1, c_1, d_1)$  and  $r(a_1, b_1, c_2, d_2)$  is  $r(a_1, b_1, C, D)$ , suggesting  $D$  as an extension to the left-hand-side of the fd. In the next iteration,  $AD \rightarrow C$  may itself turn out to be contradicted, if  $r(a_1, b_2, c_1, d_2)$  happens to be in  $r$ . But then we obtain  $r(a_1, B, C, d_2)$  as anti-unification of  $r(a_1, b_2, c_1, d_2)$  and  $r(a_1, b_1, c_2, d_2)$ , suggesting  $B$  as an extension to the left hand side of the fd. This yields the even more specific fd  $ABD \rightarrow C$ .

The procedure is slightly more involved than suggested above, because the disagreement might contain several attributes, each of which is a sufficient extension to the left-hand-side of the contradicted fd (at least for those two witnesses). Thus, each pair of witnesses can suggest a number of extensions, and each possible replacement should follow one suggestion for each pair of witnesses, e.g., if  $A \rightarrow C$  is contradicted by  $r(a_1, b_1, c_1, d_1)$  and  $r(a_1, b_1, c_2, d_2)$ , the disagreement is  $\{B, D\}$ , yielding the possible replacements  $AB \rightarrow C$  and  $AD \rightarrow C$ . The full algorithm is given below. This algorithm was presented by Flach [1990][9].

**ALGORITHM 6.2. Specialization of an fd contradicted by two tuples.**

**Input:** An fd  $X \rightarrow A$  and two tuples  $t_1, t_2$  contradicting it.

**Output:** The set of least specialization of  $X \rightarrow A$ , not contradicted by  $t_1, t_2$ .

**proc** `fd_specialize` ( $X \rightarrow A, t_1, t_2$ );

    SPECIALISED\_FDS :=  $\emptyset$ ;

    DISAGREEMENT := the set of attributes for which  $t_1$  and  $t_2$  have different values;

    DISAGREEMENT := DISAGREEMENT -  $\{A\}$ ;

**for each** *ATTR* in *DISAGREEMENT* **do**

```

    add  $(X \cup ATTR) \rightarrow A$  to SPECIALISED_FDS;
enddo
return (SPECIALISED_FDS).
endproc

```

The above characterization algorithm has been implemented in C programming language, applied separately to each possible right hand side. The database tuples are asserted on the object-level, proving contradiction of fds as specified in formula (6.2). On the other hand, the replacement procedure operates on a meta-level, on which fds are described by (lists of) attribute names. This greatly simplifies the manipulation and specialization of fds. There is a very straightforward procedure for translating fds on this meta-level to the object-level. We note that a cover for  $FD(r)$  is usually smaller than the union over  $A$  of covers for  $\{X \rightarrow A \mid X \rightarrow A \text{ is satisfied by } r\}$ , due to the pseudo-transitivity derivation rule ( $X \rightarrow A$  and  $AY \rightarrow B$  imply  $X \rightarrow B$ ). Thus, the characterization algorithm could be made more efficient by not splitting it into separate procedures for every possible right-hand-side.

### 6.3 Characterization by multi-valued dependencies

Relation  $r$  satisfies a multi-valued dependency (mvd)  $X \twoheadrightarrow Y$  if  $t_1 \in r$  and  $t_2 \in r$  and  $t_1(X) = t_2(X)$  imply that there exists a tuple  $t_3 \in r$  with  $t_3(X) = t_1(X)$ ,  $t_3(Y) = t_1(Y)$ , and  $t_3(Z) = t_2(Z)$ , where  $Z$  denotes  $R - XY$ . In words, the set of  $Y$ -values associated with a particular  $X$ -value must be independent of the values of the rest of the attribute ( $Z$ ). The symmetry of this definition implies that there is also a tuple  $t_4 \in r$  with  $t_4(X) = t_1(X)$ ,  $t_4(Y) = t_2(Y)$ , and  $t_4(Z) = t_1(Z)$ . Let  $r$  be the relation shown in Table 2, then  $r$  satisfies the mvds  $A \twoheadrightarrow B$  and  $A \twoheadrightarrow CD$ .

$A$	$B$	$C$	$D$
$a_1$	$b_1$	$c_1$	$d_1$
$a_1$	$b_2$	$c_2$	$d_2$
$a_1$	$b_1$	$c_2$	$d_2$
$a_1$	$b_2$	$c_1$	$d_1$
$a_2$	$b_2$	$c_2$	$d_1$

Table 6.2. A relation, satisfying some multi-valued dependencies.

Define  $MVD(r) = \{X \twoheadrightarrow Y \mid r \text{ satisfies } X \twoheadrightarrow Y\}$ . With mvds we only need a cover for  $MVD(r)$ , containing for instance no trivial mvds; if  $X \twoheadrightarrow Y \in MVD(r)$  then also  $X \twoheadrightarrow Z \in MVD(r)$ , with  $Z = R - XY$ . Such a cover can be represented by a set of dependency basis [19]. The dependency basis  $DEP(X)$  of  $X \subseteq R$  wrt  $MVD(r)$  is a partition of  $R$  containing  $X$ , such that  $X \twoheadrightarrow Y \in MVD(r)$  iff  $Y$  is the union of some sets in  $DEP(X)$ . For instance, if  $R = \{A, B, C, D\}$ , the dependency basis of  $A$  wrt  $\{A \twoheadrightarrow B\}$  is  $\{A, B, CD\}$ , implying the mvd  $A \twoheadrightarrow CD$ . Thus,  $MVD(r)$  can be completely described by  $\{DEP(X) \mid X \subseteq R\}$ .

Satisfaction and contradiction of mvds can be extended to dependency basis in the obvious way. We then have the following theorem.

**THEOREM 6.2** (More general than for mvds). If a relation  $r$  satisfies a dependency basis  $DEP1(X)$ , then it also satisfies any dependency basis  $DEP2(X')$  such that  $X' \supseteq X$ , and  $DEP1(X)$  is a finer partition than  $DEP2(X')$ .

Proof. If  $r$  contradicts  $DEP2(X')$ , then there is an mvd  $X' \twoheadrightarrow Y$  that is contradicted by  $r$ , such that it is the union of some sets in  $DEP2(X')$ . That is, there are  $t_1 \in r$  and  $t_2 \in r$  with  $t_1(X') = t_2(X')$ , such that no  $t_3 \in r$  satisfies  $t_3(X') = t_1(X')$ ,  $t_3(Y) = t_1(Y)$ , and  $t_3(Z') = t_2(Z')$ , where  $Z'$  denotes  $R - X'Y$ . But then no  $t_4 \in r$  satisfies  $t_4(X) = t_1(X)$ ,  $t_4(Y) = t_1(Y)$ , and  $t_4(Z) = t_2(Z)$ , where  $Z$  denotes  $R - XY$ , either. Thus  $X \twoheadrightarrow Y$  is also contradicted by  $r$ . But  $X \twoheadrightarrow Y$  is implied by  $DEP1(X)$ , which is therefore also contradicted by  $r$ .

We call  $DEP1(X)$  as general as  $DEP2(X')$ , because the former logically implies the latter. Thus, the set  $\{DEP(X) \mid X \subseteq R\}$  can be made non-redundant by removing those elements that are less general than others. For instance,  $DEP1(A) = \{A, B, C, D\}$  is more general than both  $DEP2(A) = \{A, B, CD\}$  and  $DEP3(AB) = \{AB, C, D\}$ . Consequently, the most general  $MVD(r)$  is represented by  $DEP(\emptyset) = R$ . This relation of generality forms the basis for a downward characterization algorithm for mvds, similar to Algorithm 6.1 above: if the current set of dependency basis implies an mvd that is falsified by a new tuple, the guilty dependency basis is removed and replaced by more specific ones.

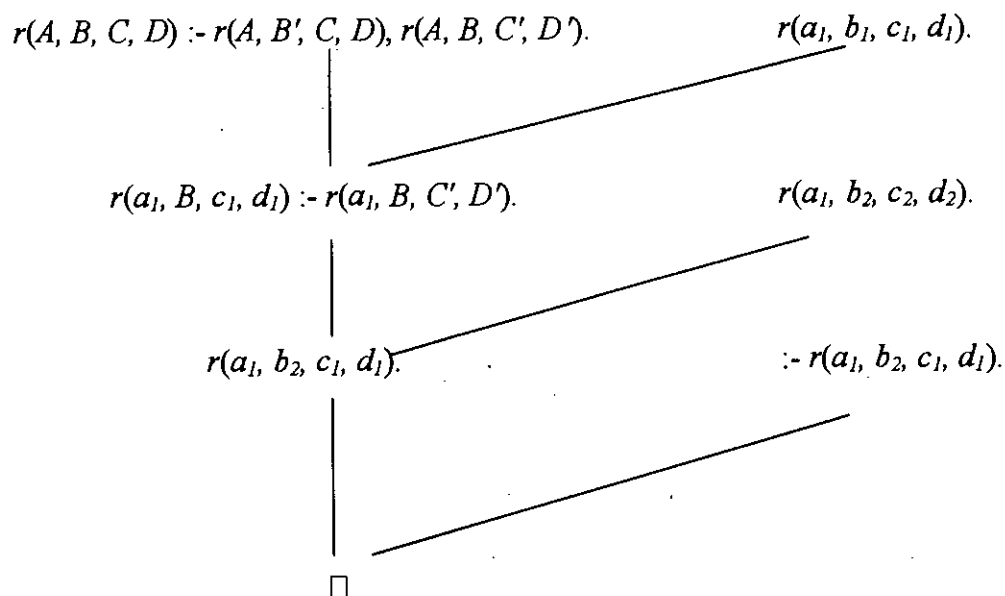
The satisfaction of an mvd  $X \twoheadrightarrow Y$  by a relation  $r$  is expressed in Horn form as:

$$r(X_1, \dots, X_n, Y_1, \dots, Y_m, Z_1, \dots, Z_k) :- \\ r(X_1, \dots, X_n, A_1, \dots, A_m, Z_1, \dots, Z_k), r(X_1, \dots, X_n, Y_1, \dots, Y_m, B_1, \dots, B_k). \quad (6.3)$$

assuming for notational convenience that  $X$  denotes the first  $n$  attributes of  $r$ , and  $Y$  denotes the next  $m$  attributes. For instance, let  $R$  be  $\{A, B, C, D\}$ . Then both mvds  $A \rightarrow\rightarrow B$  and  $A \rightarrow\rightarrow CD$  are expressed by  $r(A, B, C, D) : r(A, B', C, D), r(A, B, C', D')$ . Thus, the general Horn form of an mvd is *Tuple3*:- *Tuple1, Tuple2*, and the test for contradiction is easily implemented in Prolog as

$$\text{mvd\_contradicted}((\text{Tuple3} :- \text{Tuple1}, \text{Tuple2}), \text{Tuple1}, \text{Tuple2}) :- \\ \text{tuple}(\text{Tuple1}), \text{tuple}(\text{Tuple2}), \text{not tuple}(\text{Tuple3}). \quad (6.4)$$

For instance, refutation of the satisfaction of the above mvd by the positive witnesses  $r(a_1, b_1, c_1, d_1)$  and  $r(a_1, b_2, c_2, d_2)$  and the negative witness  $r(a_1, b_2, c_1, d_1)$  is given in Figure 6.2. Using this contradiction test, an algorithm for downward mvd-characterization is given by Algorithm 6.3. This algorithm was presented by Flach [1990][9]. It is analogous to Algorithm 6.1.



**Figure 6.2.** Refutation of the satisfaction of an mvd with three tuples.

**ALGORITHM 6.3. Downward characterization by multi-valued dependencies.****Input:** A relational scheme  $R$ , and a set  $r$  of tuples on  $R$ .**Output:** A set of dependency bases covering exactly those multi-valued dependencies satisfied by  $r$ .

```

proc mvd_char ( $R, r$ );
     $QUEUE := \{\emptyset, R\}$ ;
     $DEP\_SET := \emptyset$ ;
    while  $QUEUE \neq \emptyset$  do
         $DEP :=$  remove next dependency basis from  $QUEUE$ ;
        for each pair  $T_1, T_2$  from  $r$  do
            if mvd_contradicted ( $DEP, MVD, T_1, T_2$ )
                then add mvd_specialize ( $DEP, MVD, T_1, T_2$ ) to  $QUEUE$ ;
            endif
        enddo
        if  $DEP$  is not contradicted
            then  $DEP\_SET := DEP\_SET \cup DEP$ ;
        endif
    enddo
    cleanup( $DEP\_SET$ );
return ( $DEP\_SET$ );
endproc

```

The specialization queue  $QUEUE$  contains pairs  $(X, DEP(X))$ . The call `mvd_contradicted` ( $DEP, MVD, T_1, T_2$ ) differs from formula (6.4) in that it takes a dependency basis  $DEP$  and two tuples  $T_1$  and  $T_2$ , and succeeds if  $T_1, T_2$  contradict  $DEP$ , in which case it also returns the contradicted mvd  $MVD$  implied by  $DEP$ .

Specialization of a refuted dependency basis  $DEP(X)$  must be done in two ways, according to Theorem 6.2: by combining blocks in the partition (a right specialization), and by augmenting  $X$  (a left specialization). For instance, suppose the dependency basis  $DEP(A) = \{A, B, C, D\}$  and the mvd  $A \twoheadrightarrow B$  are refuted. First of all,  $DEP(A)$  must be changed, either to  $\{A, BC, D\}$  or to  $\{A, BD, C\}$ . The witnesses do not contain a clue for choosing between these two



candidates, but notice that they cannot both be satisfied : otherwise the originally refuted dependency basis would be implied. In one of the following iterations, the right one will be chosen (or again specialized). To prevent a specialization step that is too coarse, we must also add the dependency basis  $DEP(AB)=\{AB, C, D\}$ ,  $DEP(AC)=\{AC, B, D\}$  and  $DEP(AD)=\{AD, B, C\}$  to the specialization queue. The specialization procedure is given by Algorithm 6.4. This algorithm was presented by Flach [1990][9].

**ALGORITHM 6.4. Specialization of a dependency basis contradicted by two tuples.**

**Input:** A dependency basis  $DEP(X)$ , an mvd  $X \rightarrow \rightarrow Y$ , and two tuples  $t_1, t_2$  contradicting them.

**Output:** The set of least specialization of  $DEP(X)$ , not contradicted by  $t_1, t_2$ .

```

proc mvd_specialize ( $DEP(X), X \rightarrow \rightarrow Y, t_1, t_2$ )
     $SPECIALISED\_DEPS := \emptyset$  ;
/*right specialization/
    for each finest  $DEPI(X)$  such that  $DEP(X)$  is finer partition and  $Y$  is not a
        combination of blocks in  $DEP(X)$  do
        add ( $X, DEPI(X)$ ) to  $SPECIALISED\_DEPS$ ;
    enddo
/*left specialization/
    for each smallest augmentation  $X'$  of  $X$  do
        add ( $X', DEP(X)$ ) to  $SPECIALISED-DEPS$ ;
    enddo
return( $SPECIALISED\_DEPS$ ).
endproc

```

Notice that some of the left specialization may also be falsified by the same witnesses, e.g., in Figure 6.2, the  $C$ - and  $D$ -values of the second witness are immaterial, which therefore could also have been  $r(a_1, b_2, c_1, d_2)$ , falsifying the mvd  $AC \rightarrow \rightarrow B$ . If this is true, it will come out in one of the next iterations of the mvd characterization algorithm. To improve efficiency, it could also be tested within the specialization routine.

## 6.4 Inductive learning of functional dependencies

### 6.4.1 The problem of incremental characterization: monotonicity

In this section, we can reformulate the characterization problem as an inductive learning problem. That is, we switch from non-incremental characterization to incremental characterization by assuring that tuples of  $r$  are supplied one at a time. After each new tuple the current characterization should be updated. Thus, we can take advantage of the large body of work on inductive learning.

In the spirit of the majority of this work, we will assume that the last tuple of  $r$  is *not* signaled; thus, each intermediate characterization (or hypothesis) could turn out to be the final one. That is our learning criterion is identification in the limit: when given a sufficient set of examples, the learner should output the correct hypothesis after a finite number of steps and never change it afterwards [12]. Additionally, we allow for the possibility that the learning process is halted before such a sufficient set of examples (i.e. the complete relation) has been supplied. In such a case, the final hypothesis may not be the correct one, but it should be at least as close to the correct characterization as every hypothesis preceding it. Consequently, the sequence of intermediate hypothesis should demonstrate a global convergence towards the correct characterization. This global convergence is guaranteed by requiring the learning algorithm to be consistent (intermediate hypothesis make sense) and conservative (the current hypothesis is changed only when necessary).

The result of section 6.2 can also be interpreted in the context of inductive learning: non-incremental characterization can be viewed as batch learning, in which all examples are processed at once, and no intermediate hypothesis are generated. There is a very obvious method for transforming a batch learning algorithm into an incremental learning algorithm: maintain a list of examples seen so far, and on the advent of a new example, add it to the list and run the batch algorithm on the entire list. The resulting algorithm is consistent, but may not be conservative. Moreover, this approach is infeasible because of its storage and

computation time requirements. We can do better if the results of the previous run can be used in the next run, i.e., if we can take  $D(r)$  as the starting point for the calculation of  $D(r \cup \{t\})$ . In section 6.4 we prove that fds are monotonic in the following sense:  $r_1 \subseteq r_2$  implies  $FD(r_1) \supseteq FD(r_2)$ . Due to this property, we derive an inductive fd-characterization algorithm which resembles the batch algorithm very much, and which is guaranteed to be conservative.

In section 6.4.2 we show that mvds are not monotonic, which makes incremental mvd-characterization problematic, with respect to efficiency as well as convergence. A solution is by introducing possible multi-valued dependencies or pmvds, which can only be refuted by so-called negative tuples, i.e., tuples that are known to be not in the relation. We show that the set of pmvds shrinks monotonically when the sets of positive and negative tuples grow larger. An additional problem is that pmvds can not always uniquely be translated to mvds. This problem can be solved by an approach, which permits the system to query the user about crucial tuples.

**THEOREM 6.3** (monotonicity of fds).  $r_1 \subseteq r_2$  implies  $FD(r_1) \supseteq FD(r_2)$ .

Proof.  $r_1 \subseteq r_2$  implies  $\sigma_{X=x}(r_1) \subseteq \sigma_{X=x}(r_2)$  implies  $\pi_Y(\sigma_{X=x}(r_1)) \subseteq \pi_Y(\sigma_{X=x}(r_2))$ . Hence, if  $r_2$  satisfies  $X \rightarrow Y$  then  $r_1$  satisfies  $X \rightarrow Y$  and thus  $FD(r_2) \subseteq FD(r_1)$ .

An algorithm that calculates  $FD(r \cup \{t\})$  given  $r$ ,  $FD(r)$ , and  $t$  is shown in Algorithm 6.5. This algorithm was presented by Flach [1990][9]. Again, for simplicity the right-hand-side of the fds is fixed.

**ALGORITHM 6.5.** *Incremental Downward characterization by functional dependencies.*

**Input:** A set  $r$  of tuples on a relational scheme  $R$ , a non-redundant cover  $COVER$  of the set of functional dependencies  $X \rightarrow A$  satisfied by  $r$ , a tuple  $t$  on  $R$ , and an attribute  $A \in R$ .

**Output:** A non-redundant cover of the set of functional dependencies  $X \rightarrow A$ , satisfied by  $r \cup \{t\}$ .

```

proc fd_char (r, COVER, t, A)
    QUEUE := COVER;
    FD_SET :=  $\emptyset$ ;
    while QUEUE  $\neq \emptyset$  do

```

```

     $FD :=$  remove next  $fd$  from  $QUEUE$ ;
  for each tuple  $T$  from  $r$  do
    if  $fd\_contradicted(FD, t, T)$ 
    then add  $fd\_specialize(FD, t, T)$  to  $QUEUE$ ;
    endif
  enddo
  if  $FD$  is not contradicted
  then  $FD\_SET := FD\_SET \cup FD$ ;
  endif
enddo
cleanup( $FD\_SET$ );
return ( $FD\_SET$ ).
endproc

```

A non-incremental algorithm based on this incremental algorithm is shown in Algorithm 6.6. This algorithm was presented by Flach [1990][9].

**ALGORITHM 6.6. *Non-incremental downward characterization by functional dependencies.***

**Input:** A set of tuples on a relational scheme  $R$ , and an attribute  $A \in R$ .  
**Output:** A non-redundant cover of the set of functional dependencies  $X \rightarrow A$ , satisfied by  $r$ .

```

proc  $fd\_char\_non\_incr(r, A)$ ;
  if  $r = \emptyset$ 
  then  $FD\_SET := \emptyset \rightarrow A$ ;
  else select a tuple  $T$  from  $r$ ;
     $COVER := fd\_char\_non\_incr(r-T, A)$ ;
     $FD\_SET := fd\_char(r-T, COVER, T, A)$ .
  return ( $FD\_SET$ ).
endproc

```

## 6.4.2 Inductive learning of possible multi-valued dependencies

There is a very important difference between fds and mvds, as follows. An fd  $X \rightarrow Y$  says: if two tuples have equal  $X$ -values, then they also have equal  $Y$ -values. An mvd, however, says: if it is known that these two tuples are in the relation, then it is also known that two other tuples are in the relation. Put differently, mvds are tuple-generating dependencies rather than equality-testing dependencies [5]. Consequently, the analogue of theorem 6.3 does not hold for mvds.

**THEOREM 6.4** (non-monotonicity of mvds).  $MVD(r)$  does not monotonically decrease when  $r$  increases.

Proof. Let  $X \twoheadrightarrow Y \in MVD(r)$ , and let  $t_1 \in r$  and  $t_2 \notin r$  ( $t_1 \neq t_2$ ) have equal  $X$ -values, then  $X \twoheadrightarrow Y \notin MVD(r \cup \{t_1, t_2\})$ . Now, let  $t_3(X) = t_1(X)$ ,  $t_3(Y) = t_1(Y)$ ,  $t_3(Z) = t_2(Z)$  ( $Z = R - XY$ ),  $t_4(X) = t_1(X)$ ,  $t_4(Y) = t_2(Y)$ , and  $t_4(Z) = t_1(Z)$ , then again  $X \twoheadrightarrow Y \in MVD(r \cup \{t_1, t_2, t_3, t_4\})$ . (NB.  $t_1 \notin r$  and  $t_2 \notin r$  and  $X \twoheadrightarrow Y \in MVD(r)$  implies  $t_3 \notin r$  and  $t_4 \notin r$ .)

For an example, see the relation in Table 6.3: the first four tuples constitute  $r$ , and the other four are  $t_1, t_2, t_3$ , and  $t_4$ .

	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
	$A_1$	$B_1$	$C_1$	$d_1$
	$A_1$	$B_2$	$C_2$	$d_2$
	$A_1$	$B_1$	$C_2$	$d_2$
	$A_1$	$B_2$	$C_1$	$d_1$
$T_1$ :	$A_2$	$B_2$	$C_2$	$d_1$
$T_2$ :	$A_2$	$B_1$	$C_1$	$d_2$
$T_3$ :	$A_2$	$B_2$	$C_1$	$d_2$
$T_4$ :	$A_2$	$B_1$	$C_2$	$d_1$

**Table 6.3.**  $MVD$  is not monotonic:  $r$  satisfies  $A \twoheadrightarrow B$ ,  $r \cup \{t_1, t_2\}$  does not, and  $r \cup \{t_1, t_2, t_3, t_4\}$  again does.

Theorem 6.4 implies that  $MVD(r \cup \{t\})$  can not be constructed by simply removing falsified mvds from  $MVD(r)$ : some mvds not in the latter set might have to be added to the former. It is thus not possible to derive an incremental algorithm from the non-incremental one (Algorithm 6.3) in the same way as Algorithm 6.5 was derived from Algorithm 6.1 in the fd-case. On the contrary, an incremental approach would require reconsideration of all tuples upon arrival of a new tuple, and would therefore be considerably less efficient than the non-incremental algorithm. Moreover, the resulting algorithm would not be conservative. In the remainder of this section, an approach based on incorporating explicit negative information in the learning process is shown.

The behavior of non-monotonic characterizations can be explained by the Closed World Assumption (CWA), which states that everything that is not known to be true is assumed to be not true. This assumption is in conflict with an incremental approach, which assumes that if a tuple is not in the current, partial relation, it may still appear in a future extension. The only way to resolve this conflict, is to abandon the CWA by defining a mvd to be possibly satisfied by a partial relation  $r$  if it is satisfied by some extension  $r' \supseteq r$ . This introduces an additional problem: the universal extension  $r^*$  of  $r$ , i.e., the universal relation restricted to tuples containing attribute values appearing in  $r$ , satisfies every mvd. Because  $r^* \supseteq r$ , we have that  $r$  possibly satisfies every mvd as well. The only way to get around this, is to incorporate negative information in the characterization process, by supplying negative tuples that are not in the relation to be characterized. Thus, we define a possible multi-valued dependency (pmvd)  $X \twoheadrightarrow Y$  to be satisfied by a set  $p$  of positive tuples and a set  $n$  of negative tuples ( $p$  and  $n$  disjoint) iff for any three tuples  $t_1, t_2, t_3$  such that  $t_1(X)=t_2(X)=t_3(X)$ ,  $t_3(Y)=t_1(Y)$ , and  $t_3(R-XY)=t_2(R-XY)$ ,  $t_1 \in p$  and  $t_2 \in p$  implies  $t_3 \notin n$ ; alternatively the pmvd is contradicted iff  $t_1 \in p$ ,  $t_2 \in p$  and  $t_3 \in n$ . We also write  $\langle p, n \rangle \models X \twoheadrightarrow Y$  for a satisfied pmvd, and  $\langle p, n \rangle \not\models X \twoheadrightarrow Y$  for a contradicted pmvd. The set of pmvds satisfied by  $\langle p, n \rangle$  is denoted  $PMVD(p, n)$ . Clearly, this set is monotonic with regard to positive and negative tuples.

**THEOREM 6.5** (monotonicity of pmvds).

- (i)  $p_1 \subseteq p_2$  implies  $PMVD(p_1, n) \supseteq PMVD(p_2, n)$ .
- (ii)  $n_1 \subseteq n_2$  implies  $PMVD(p, n_1) \supseteq PMVD(p, n)$ .

Proof. Immediate from the definition of satisfaction of pmvds.

The generality relation for pmvds is given by Theorem 6.6.

**THEOREM 6.6.** (more general than for mvds)

(i) (augmentation)  $\langle p, n \rangle \models X \Rightarrow Y$  implies  $\langle p, n \rangle \models XZ \Rightarrow Y$ ;

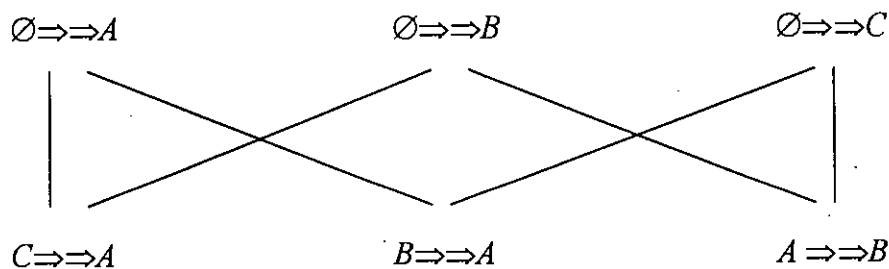
(ii) (complementation)  $\langle p, n \rangle \models X \Rightarrow Y$  implies  $\langle p, n \rangle \models X \Rightarrow R - XY$ .

Proof. (i) Let  $XZ \Rightarrow Y$  be contradicted by witnesses  $t_1, t_2, t_3$ , then  $t_1(XZ) =$

$t_2(XZ) = t_3(XZ)$  implies  $t_1(X) = t_2(X) = t_3(X)$ ;  $t_2(XZ) = t_3(XZ)$  and  $t_3(R - XZY) = t_2(R - XZY)$  implies  $t_3(R - XY) = t_2(R - XY)$ . Thus  $X \Rightarrow Y$  is contradicted by the same witnesses.

(ii) Immediate.

Theorem 6.6 (ii) shows that the pmvds  $X \Rightarrow Y$  and  $X \Rightarrow R - XY$  are equivalent (suggesting a dependency basis-like representation  $PDEP(X) = \{X, Y, R - XY\}$ ). Theorem 6.6 (i) shows, that a more specific pmvd is obtained by removing attributes from  $Y$  or  $R - XY$  and adding them to  $X$ . In Figure 6.3, the generality relation is depicted for pmvds on the relation scheme  $R = \{A, B, C\}$ . Only non-trivial pmvds, i.e.,  $X \Rightarrow Y$  with both  $Y$  and  $R - XY$  non-empty, are included also, for each pair of equivalent pmvds, only one representative is included.



**Figure 6.3.** Non-trivial pmvds on  $R = \{A, B, C\}$ , ordered by generality.

Note that this relation is a restriction of the generality relation for mvds, e.g., pseudo-transitivity,  $p \models \emptyset \rightarrow A$  and  $p \models A \rightarrow B$  imply  $p \models \emptyset \rightarrow B$ , but  $\langle p, n \rangle \models \emptyset \Rightarrow A$  and  $\langle p, n \rangle \models A \Rightarrow B$  do not imply  $\langle p, n \rangle \models \emptyset \Rightarrow B$ .

The contradiction test for pmvds is derived from the test for mvds (formula (6.4) by changing not tuple (*Tuple3*) (implementing the CWA by negation as failure) to neg\_tuple (*Tuple3*) (testing for explicit negative information).

$\text{pmvd\_contradicted}((\text{Tuple3:- Tuple1, Tuple2}), \text{Tuple1}, \text{Tuple2}, \text{Tuple3}) :-$   
 $\text{pos\_tuple}(\text{Tuple1}), \text{pos\_tuple}(\text{Tuple2}), \text{neg\_tuple}(\text{Tuple3}). \quad (6.3.1)$

Following is an incremental pmvd-characterization algorithm. This algorithm was presented by Flach [1990][9].

**ALGORITHM 6.7. Incremental downward characterization by possible multi-valued dependencies.**

**Input:** A set  $p$  of positive tuples and a set  $n$  of negative tuples on a relational scheme  $R$ , the set of most general possible multi-valued dependencies satisfied by  $\langle p, n \rangle$  and a pair  $\langle t, s \rangle$  consisting of a tuple  $t$  on  $R$ , and  $s \in \{+, -\}$ .

**Output:** The set of most general possible multi-valued dependencies satisfied by  $\langle p \cup \{t\}, n \rangle$  if  $s = +$ , or by  $\langle p, n \cup \{t\} \rangle$  if  $s = -$ .

```

proc pmvd_char_incr (p, n, IN_PMVDS, <t, s>)
    QUEUE: = IN_PMVDS;
    PMVD_SET: =  $\emptyset$ ;
    while QUEUE  $\neq \emptyset$  do
        PMVD: = remove next pmvd from QUEUE;
        if s = +
        then /*positive tuple*/
            for each tuple  $T_p$  from p and  $T_n$  from n do
                if pmvd_contradicted(PMVD, t,  $T_p$ ,  $T_n$ )
                    then add pmvd_specialize(PMVD, t,  $T_p$ ,  $T_n$ ) to
                        QUEUE;
                endif
            enddo
        else /*negative tuple*/
            for each pair  $T_{p1}, T_{p2}$  from p do
                if pmvd_contradicted(PMVD,  $T_{p1}, T_{p2}, t$ )
                    then add pmvd_specialize(PMVD,  $T_{p1}, T_{p2}, t$ ) to
                        QUEUE
                endif
            enddo
        endif
    endwhile
endproc

```



```

        endif
    enddo
endif
if PMVD is not contradicted
then PMVD_SET: = PMVD_SET  $\cup$  PMVD;
enddo
cleanup(PMVD_SET)
return(PMVD_DEPS).
endproc

```

The operation of Algorithm 6.7 depends on whether the new tuple is positive or negative. Notice that the call `pmvd_contradicted(PMVD,  $T_{p1}$ ,  $T_{p2}$ ,  $t$ )` now takes three tuples. The call `cleanup(PMVD_SET)` removes redundant pmvds with respect to the generality ordering.

The specialization algorithm for pmvds is shown in Algorithm 6.8. This algorithm was presented by Flach [1990][9].

**ALGORITHM 6.8.** *Specialization of a possible multi-valued dependency contradicted by three witnesses.*

**Input:** A possible multi-valued dependency  $X \Rightarrow \Rightarrow Y$  contradicted by three distinct witnesses  $t_1$ ,  $t_2$  (positive) and  $t_3$  (negative).

**Output:** The set of least specialization of  $X \Rightarrow \Rightarrow Y$ , not contradicted by  $t_1$ ,  $t_2$ ,  $t_3$ .

```

proc pmvd_specialize ( $X \Rightarrow \Rightarrow Y$ ,  $t_1$ ,  $t_2$ ,  $t_3$ );
    SPECIALISED_PMVDS: =  $\emptyset$ ;
    DISAGREEMENT1:= the attributes for which only  $t_1$  and  $t_3$  have the same
                    values;
    DISAGREEMENT2:= the attributes for which only  $t_2$  and  $t_3$  have the same
                    values;
    if DISAGREEMENT1 contains more than one attribute
    then
        for each attribute  $A$  in DISAGREEMENT1 do

```

```

    add  $(X \cup A) \Rightarrow (Y - A)$  to SPECIALISED_PMVDS;
  enddo
endif
if DISAGREEMENT2 contains more than one attribute
then
  for each attribute B in DISAGREEMENT2 do
    add  $(X \cup B) \Rightarrow (Y - B)$  to SPECIALISED_PMVDS;
  enddo
endif
return (SPECIALISED_PMVDS).
endproc

```

The main idea behind Algorithm 6.8 is, to augment the left hand side of a contradicted pmvd with an attribute for which not all three witnesses have the same value. Care must be taken to prevent generation of trivial pmvds. For instance, let  $t_1 = \langle a_1, b_1, c_1, d_1, e_1 \rangle$ ,  $t_2 = \langle a_1, b_1, c_2, d_2, e_2 \rangle$  and  $t_3 = \langle a_1, b_1, c_2, d_1, e_1 \rangle$ , refuting the pmvd  $A \Rightarrow C$ ; we have *DISAGREEMENT1* = *C* and *DISAGREEMENT2* = *DE* (each of these sets is necessarily non-empty, otherwise the negative witness would be identical to one of the positive witnesses). *DISAGREEMENT1* contains only one attribute; moving it to the left-hand-side would result in a trivial pmvd. Moving one attribute from *DISAGREEMENT2* to the left hand side results in the pmvds  $AD \Rightarrow C$  and  $AE \Rightarrow C$ . Notice that the complementary pmvds  $AD \Rightarrow BE$  and  $AE \Rightarrow BD$  are not generated; they would have been generated upon the call `pmvd_specialized( $A \Rightarrow BDE, t_1, t_2, t_3$ )`.

### 6.4.3 From possible multi-valued dependencies to satisfied multi-valued dependencies: a querying approach

We have to use an incremental pmvd-characterization algorithm, but it is mvds rather than pmvds that we are interested in. Thus, the remaining issue is: what is the relationship between  $MVD(r)$  and  $PMVD(p, n)$ , expressed in terms of the relationship between  $r$  on the one hand and  $p$  and  $n$  on the other? The obvious approach would be to take  $r=p$  and  $MVD(r)=PMVD(p, n)$ . However, this will lead to inconsistencies, because the implicational

structure of mvds is stronger than the implicational structure of pmvds. For instance, let  $p = \{ \langle a_1, b_1, c_1, d_1 \rangle, \langle a_1, b_2, c_2, d_2 \rangle \}$  and  $n = \{ \langle a_1, b_1, c_2, d_1 \rangle \}$ , then  $\langle p, n \rangle$  satisfies both  $A \Rightarrow B$  and  $A \Rightarrow BC$ , but contradicts  $A \Rightarrow C$ , while  $A \rightarrow B$  follows by projectivity from  $A \rightarrow B$  and  $A \rightarrow BC$ . Thus, we cannot include both  $A \rightarrow B$  and  $A \rightarrow BC$  in  $MVD(r)$ , but there are no reasons to choose either one of them.

There are, however, conditions under which  $MVD(r) = PMVD(p, n)$  is valid: if we have seen all negative tuples that are crucial. This idea is formalized as follows. We call  $\langle p, n \rangle$  necessary for  $r$  iff every positive example is in  $r$ , and every negative example is not in  $r$  but in  $r^*$ :  $r \subseteq p$  and  $r^* - r \subseteq n$ . Likewise, we call  $\langle p, n \rangle$  sufficient for  $r$  iff every tuple in  $r$  has been supplied as a positive example, and every tuple not in  $r$  but in  $r^*$  has been supplied as a negative example:  $r \subseteq p$  and  $r^* - r \subseteq n$ . We call  $\langle p, n \rangle$  complete for  $r$  iff  $\langle p, n \rangle$  is both necessary and sufficient for  $r$ .

**LEMMA 6.7.** Let  $t_1, t_2, t_3$  be three tuples with  $t_1(X) = t_2(X) = t_3(X)$ ,  $t_3(Y) = t_1(Y)$ , and  $t_3(R - XY) = t_2(R - XY)$ .

- (i) If  $\langle p, n \rangle$  is necessary for  $r$ ,  $t_1 \in p$ ,  $t_2 \in p$  and  $t_3 \in n$  implies  $t_1 \in r$ ,  $t_2 \in r$  and  $t_3 \notin r$ .
- (ii) If  $\langle p, n \rangle$  is sufficient for  $r$ ,  $t_1 \in r$ ,  $t_2 \in r$  and  $t_3 \notin r$  implies that  $t_1 \in p$ ,  $t_2 \in p$  and  $t_3 \in n$ .

Proof. Trivial.

In what follows,  $t_1, t_2$  and  $t_3$  are witnesses as in Lemma 6.7. Necessary tuples are needed for contradiction of pmvds.

**THEOREM 6.8.** If  $\langle p, n \rangle$  is necessary for  $r$ ,  $MVD(r) \subseteq PMVD(p, n)$ .

Proof. Suppose  $\langle p, n \rangle \models X \Rightarrow Y$ , i.e., there are witnesses  $t_1 \in p$ ,  $t_2 \in p$  and  $t_3 \in n$ . By Lemma 6.7 (i)  $t_1 \in r$ ,  $t_2 \in r$  and  $t_3 \notin r$ , hence  $r \models X \rightarrow Y$ .

For instance, the positive tuples  $p = \{ \langle a_1, b_1, c_1, d_1 \rangle, \langle a_1, b_2, c_2, d_2 \rangle \}$  and the negative tuple  $n = \{ \langle a_1, b_1, c_2, d_1 \rangle \}$  contradict the pmvds  $A \Rightarrow C$  and  $A \Rightarrow BD$ ;  $\langle p, n \rangle$  is necessary for the relation  $r$  depicted in Table 6.4, which therefore does not satisfy the corresponding mvds.

	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
<i>P</i> :	<i>a</i> <sub>1</sub>	<i>b</i> <sub>1</sub>	<i>c</i> <sub>1</sub>	<i>D</i> <sub>1</sub>
<i>P</i> :	<i>a</i> <sub>1</sub>	<i>b</i> <sub>2</sub>	<i>c</i> <sub>2</sub>	<i>D</i> <sub>2</sub>
<i>N</i> :	<i>a</i> <sub>1</sub>	<i>b</i> <sub>1</sub>	<i>c</i> <sub>2</sub>	<i>D</i> <sub>2</sub>

**Table 6.4.** A relation contradicting the mvds  $A \rightarrow \rightarrow C$  and  $A \rightarrow \rightarrow BD$ .

Analogously, a sufficient set of positive and negative tuples contradicts at least those pmvds which are (as mvd) not satisfied by the positive tuples.

**THEOREM 6.9.** If  $\langle p, n \rangle$  is sufficient for  $r$ ,  $MVD(r) \supseteq PMVD(p, n)$

Proof. Suppose the mvd  $X \rightarrow \rightarrow Y$  is not satisfied by  $r$ , i.e., there are witnesses  $t_1 \in r$ ,  $t_2 \in r$  and  $t_3 \notin r$ . By Lemma 6.7 (ii)  $t_1 \in p$ ,  $t_2 \in p$  and  $t_3 \in n$ ; hence  $\langle p, n \rangle \models X \Rightarrow Y$ .

A complete set of positive and negative tuples results in a set of pmvds, equivalent with the set of mvds satisfied by the positive tuples.

**COROLLARY 6.10.** If  $\langle p, n \rangle$  is complete for  $r$ ,  $MVD(r) = PMVD(p, n)$ .

For instance, a complete set of positive and negative tuples for the relation in Table 6.4 is depicted in Table 6.5.

	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
<i>p</i> :	<i>a</i> <sub>1</sub>	<i>B</i> <sub>1</sub>	<i>C</i> <sub>1</sub>	<i>d</i> <sub>1</sub>
<i>p</i> :	<i>a</i> <sub>1</sub>	<i>B</i> <sub>2</sub>	<i>c</i> <sub>2</sub>	<i>d</i> <sub>2</sub>
<i>p</i> :	<i>a</i> <sub>1</sub>	<i>B</i> <sub>1</sub>	<i>c</i> <sub>2</sub>	<i>d</i> <sub>2</sub>
<i>n</i> :	<i>a</i> <sub>1</sub>	<i>B</i> <sub>1</sub>	<i>c</i> <sub>2</sub>	<i>d</i> <sub>1</sub>
<i>n</i> :	<i>a</i> <sub>1</sub>	<i>B</i> <sub>1</sub>	<i>c</i> <sub>1</sub>	<i>d</i> <sub>2</sub>
<i>n</i> :	<i>a</i> <sub>1</sub>	<i>B</i> <sub>2</sub>	<i>c</i> <sub>1</sub>	<i>d</i> <sub>1</sub>
(*) <i>n</i> :	<i>a</i> <sub>1</sub>	<i>B</i> <sub>2</sub>	<i>c</i> <sub>2</sub>	<i>d</i> <sub>1</sub>
(*) <i>n</i> :	<i>a</i> <sub>1</sub>	<i>B</i> <sub>2</sub>	<i>c</i> <sub>1</sub>	<i>d</i> <sub>2</sub>

**Table 6.5.**  $\langle p, n \rangle$  is complete for the relation in Table 6.4.

In fact, the two negative tuples marked (\*) are superfluous, because every pmvd they refute can be refuted by means of one of the other three negative tuples (this is related to the symmetry of the definition of satisfaction of an mvd). This does not mean that the starred tuples might just as well have been positive; it means, that removing them from  $n$  does not influence the set of refuted pmvds. Thus, the starred negative tuples are not necessary for  $r$ , implying that the definition of necessary negative tuples is somewhat too strong.

In reality, we do not know  $r$ , the relation to be incrementally characterized. Corollary 6.10 shows that the set of pmvds can be consistently interpreted as a set of mvds if  $\langle p, n \rangle$  is complete for some  $r$  (i.e. if  $p \cup n = p^*$ , according to the definition of completeness given above). An approach of extending  $p$  and  $n$  such that  $\langle p, n \rangle$  is complete is that we have to let the system pose queries to the user. That is, the system generates typical tuples not yet in  $p$  or  $n$ , which the user must classify as either positive or negative. This process halts when  $p$  and  $n$  are complete for  $p$ , resulting in a set of pmvds that can consistently be interpreted as an mvd-characterization of  $p$ . The query-process can naturally be integrated with the specialization process. Several query-strategies are possible, e.g., a cautious approach (corresponding to the search for least specialization), or a divide-and-conquer approach, searching for a refutation of a pmvd somewhere between a most general satisfied pmvd and a trivial (most specific) pmvd. The cautious querying approach is illustrated below.

We assume that the user initially only supplies positive tuples; negative tuples are obtained by means of queries. As in the previous algorithms, we maintain a queue of most general pmvds and try to falsify each one of them, as follows: for each pair of positive tuples  $t_1, t_2$ , we construct the negative witness  $t_3$ . Now there are three possibilities: (i)  $t_3 \in p$ , i.e., the pmvd cannot be contradicted by means of  $t_1, t_2$ , and we proceed with the following pair of positive tuples; (ii)  $t_3 \in n$ , i.e., the pmvd is indeed contradicted and needs to be specialized; (iii)  $t_3 \notin p$  and  $t_3 \notin n$ , and we ask the user to classify  $t_3$  as either positive or negative;  $t_3$  is added to the appropriate set of tuples, and we proceed with either case (i) or case (ii). The process halts if every tuple thus constructed is in  $p$ . In this case,  $\langle p, n \rangle$  is complete for  $p$  (in the weaker sense discussed above), and the resulting set of pmvds can consistently be interpreted as a set of mvds.

We illustrate this approach with an example session with a Prolog program. The example is taken from [19]. A tuple  $\text{service}(f, d, p)$  means that flight number  $f$  flies on day  $d$  and can use plane type  $p$  on that day. User input is in bold.

?- **pmvd\_learn.**

Relation: **service(flight, day, plane).**

Dependencies:

Service: [ ]  $\rightarrow\rightarrow$  [plane]

Service: [ ]  $\rightarrow\rightarrow$  [flight]

Service: [ ]  $\rightarrow\rightarrow$  [day]

New tuple: **service (106, Monday, 747).**

New tuple: **service (106, Thursday, 1011).**

Is service (106, Thursday, 747) in the relation? **yes.**

Is service (106, Monday, 1011) in the relation? **yes.**

The user specifies the relation scheme, and the system shows the initial set of most general pmvds. The user types in the first two tuples, which concern flight number 106. The system tries to falsify the pmvds  $\emptyset \Rightarrow \Rightarrow PLANE$  and  $\emptyset \Rightarrow \Rightarrow DAY$  by asking for a classification for two other tuples. Both of these tuples are classified as positive, so none of the pmvds is contradicted (note that  $\emptyset \Rightarrow \Rightarrow FLIGHT$  cannot be contradicted, because all tuples have the same flight number). This results in a complete set of positive and negative tuples.

New tuple: **service (204, Wednesday, 707).**

Is service (106, Monday, 707) in the relation? **no.**

Specialize [ ]  $\rightarrow\rightarrow$  [plane]

service (204, Wednesday, 707)

service (106, Monday, 1011)

not service (106, Monday, 707)

Is service (204, Monday, 1011) in the relation? **no.**

Specialize [ ]  $\rightarrow\rightarrow$  [flight]

service (204, Wednesday, 707)

service (106, Monday, 1011)

not service (106, Monday, 1011)

Is service (106, Wednesday, 1011) in the relation? **no.**

Specialize [ ]  $\rightarrow\rightarrow$  [day]

service (204, Wednesday, 707)

service (106, Monday, 1011)

not service (106, Wednesday, 1011)

The next positive tuple introduces new values for all three attributes. Now, the system is able to refute each initial pmvd by constructing appropriate negative witnesses. They are replaced by more specific pmvds, which cannot be contradicted because there are no two distinct positive tuples with either flight number 204, day Wednesday, or plane type 707. Thus, the positive and negative tuples are complete, in the weaker sense.

New tuple: **service (204, Wednesday, 727).**

New tuple: **stop.**

Dependencies:

service: [day]  $\rightarrow\rightarrow$  [flight]

service: [flight]  $\rightarrow\rightarrow$  [day]

service: [plane]  $\rightarrow\rightarrow$  [day]

Yes

Adding one more positive tuple does not result in additional queries. After the learning process is halted by the user, the system shows its final hypothesis, which can consistently be interpreted as a set of mvds. Note, that this set of pmvds contains no redundant or trivial mvds.

# Chapter 7

## Proposed Design Frame Work and Algorithms

### 7.1 Design Framework

Instead of looking for the entity (relation) and the associated set of attributes for each entity (relation) it is quite easy to look only for the entire set of attributes not associating them with different entities (relations) i.e. not associating them with different database/tables. To do this job efficiently we must have clear conception of distinguishing an entity from an attribute.

During the system analysis, we will get both entity and attribute formally described as a noun. Clearly, if an item does not present more than once in the system it is neither an entity and nor an attribute. An entity has independent existence in the system whereas an attribute cannot exist independently. The existence of an attribute is dependent on the existence of one or more other attributes. This dependency defines an entity as a collection of attributes, which depends on each other for their mutual existence [6]. It is hard to find this dependency by simple observation, particularly, for a large and complex system.

Without trying to find this dependency it is better to define a single scheme putting all the attributes in it. Thus the initial data model would have only one scheme and one relation on that scheme. At first, We have to collect tuples on this relation. After collecting tuples on this single relation (database) it has to be analyzed for characterization (decomposition). Some tuples may have NULL values on one or more attribute/attributes. We have to divide tuples into several groups putting the tuples in a same group those have NULL values on the same attribute/attributes.

For each group of tuples we have to define a new database scheme discarding those attributes which have NULL values, and distribute tuples on those newly defined database scheme so that no relation has NULL value in any of its attributes. This ad-hoc scheme would need to be characterized further for getting normalized database scheme.

Applying Algorithm 6.1 or Algorithm 6.5 or Algorithm 6.6 and Algorithm 6.3 or Algorithm 6.7 on this ad-hoc scheme we can get the set of functional and multi-valued dependencies.



Applying Algorithm 7.1 we have to merge those functional dependencies, whose antecedent attribute/ attributes is/are same into a single dependency. We have to list multi-valued dependencies separately. Applying Algorithm 7.2 we need to derive the set of relations in 2NF and keys and the set of functional and multi-valued dependencies holds on these relations respectively. If any attribute of ad-hoc scheme were not taking part in any of the functional or multi-valued dependencies that attribute would be discarded from the new scheme in 2NF.

We have to analyze these 2NF relations (database) to see whether they are in BCNF or not. If these relations (database) are not in BCNF we have to apply Algorithm 5.2 to decompose them into new relations (database) in BCNF. Applying Algorithm 5.1 we have to confirm that these new relations (databases) are also dependency preserving. If they are not dependency preserving we have to discard the decomposition and have to get the original 2NF relations (databases) back and analyze them to see whether they are in 3NF or not. If these relations (database) are not in 3NF we have to apply Algorithm 5.3 to decompose them into new relations (database) in 3NF. The Algorithm 5.3 also provides dependency preserving relations.

After getting the relations (databases) either in BCNF or in 3NF we have to check whether there is any multi-valued dependency or not. If there is any multi-valued dependency we have to decompose that relation into new relations in 4NF using Algorithm 5.4.

## 7.2 Proposed New Algorithms

Algorithm 7.1 is for computing the merged set of functional dependencies. A set of functional dependencies is given input to this procedure. It will merge two dependencies into one if the antecedent attribute/attributes is/are same. This merging is done in iteration simply using the union rule of functional dependencies

**Algorithm 7.1: The Algorithm for merging functional dependencies**

**Input:** A set of functional dependencies  $F$ ; the format of each dependency is  $X \rightarrow Y$ .

**Output:** A set of functional dependencies  $F'$  merging those functional dependencies in  $F$  whose antecedent attribute/ attributes is/are same into a single dependency.

**proc** merge ( $F(n), F'(n)$ )

**for each** pair of functional dependencies  $F_i : X_i \rightarrow Y_i$  and  $F_j : X_j \rightarrow Y_j$  **do**

**if**  $X_i = X_j$  **then**

$F'_i : X_i \rightarrow Y_i \cup Y_j$

**else**

$F'_i : X_i \rightarrow Y_i$

$F'_{i+1} : X_j \rightarrow Y_j$

**endif**

**enddo**

**endproc**

Algorithm 7.2 is for constructing relations in 2NF from a given set of functional dependencies. This procedure is taking a set of functional and multi-valued dependencies as input. Each dependency in the set is checked to see whether all the attributes of it are already in any of the relations derived so far or not. If attributes are found to be in a relation, this program will mention that this dependency holds on that particular relation. If attributes are not in any of the relations, this program will create a new relation comprising all the attributes of the dependency. The antecedent attribute/attributes of this dependency will be mentioned as the key and the dependency as the dependency of this new relation. The clean\_up procedure will delete that relation which is already in a bigger relation.

**Algorithm 7.2 : The Algorithm for constructing relations in 2NF.**

**Input:** A set of functional dependencies  $F$ ; format of each dependency is  $X \rightarrow Y$ .

**Output:** A set of relations  $R$  in 2NF: format of each relation is

Relation  $R(X_1, X_2, \dots)$

Key  $K_1(X_1, X_2, \dots) K_2(X_1, X_2, \dots)$

Dependencies  $D(F_1, F_2, \dots)$

```

proc 2NF relations ( $F(n_1), R(n_2)$ )
for every functional dependency  $F_i : X_i \rightarrow Y_i$  do
  for each relation  $R_j$  do
    if  $X_i$  and  $Y_i$  are in  $R_j(X_1, X_2, \dots)$  then
       $D_j(F_1, F_2, \dots) = D_j(F_1, F_2, \dots) \cup F_i$ 

      if ( $\text{num}(X_i) + \text{num}(Y_i) = \text{num}(R_j(X_1, X_2, \dots))$ ) then
        if  $X_i$  is not in any of  $K_{j,l}(X_1, X_2, \dots)$  then
           $K(X_i)$  is a key of  $R_j$ 
        endif
      endif
    endif
  enddo
  if  $F_i$  does not hold in any of  $R_j$  then
    Create new  $R_{n_2+1}$ 
     $R_{n_2+1}(K) = X_i$ 
     $D_{n_2+1}(F_1) = F_i$ 
  endif
enddo
clean_up ()
endproc

```

## Chapter 8

# Experimental Results

### 8.1 Target System

A university has several departments. Each department has a unique name called the department name that is the identification of that department. Each department has an administrative head. Head of the department is described by his/her name. Two departments can have heads of the same name though they are different person. Each department has several students. Each student is uniquely identified by his/her ID. Each student has an advisor. But one advisor could be the advisor of many students. Advisors have unique identification comprising his/her department and designation. Every student is allowed to registrar several courses. A course has its predefined course number, teacher, and credit. Course number is unique for a particular course. But two or more courses have the same teacher and/or the same credit. Course teacher's designation is also known to all. GPA obtained by a student in a particular course is also recorded.

Books are borrowed by research students in the research library. One student can make several borrows. Each borrow is treated independently with a unique borrow number. A borrow is recorded using borrow number, student identification number, and the research group to which the student belongs to. A student can do only course work or research or both.

### 8.2 Results

To get a relational database model we have collected all possible attributes present in the system. At first, we put all the collected attributes in a single scheme. We called this scheme the *University-scheme* as follows:

*University-Scheme* = (*std\_id*, *std\_name*, *std\_address*, *department*, *advisor\_id*, *head\_name*, *course\_number*, *course\_credit*, *gpa*, *teacher*, *borrow\_number*, *research\_group*)

Thus, our initial data model has only one scheme. And we can define *university* relation as

*university (std\_id, std\_name, std\_address, department, advisor\_id,  
head\_name, course\_number, course\_credit, gpa, teacher, borrow\_number,  
research\_group)*

After getting the initial data model that is comprising only one relation and that is *university* in this case, we have collected tuples on this relation. This is shown in the Figure 8.1. Some of the attributes of these tuples have NULL values. We have divided tuples into several groups putting the tuples in a same group those have NULL values on the same attribute/attributes. In this case, we got two groups.

<i>std_id</i>	<i>Std_name</i>	<i>Std_address</i>	<i>department</i>	<i>Advisor_id</i>	<i>Head_name</i>	<i>course #</i>	<i>Credit</i>	<i>gpa</i>	<i>teacher</i>	<i>Borrow_number</i>	<i>Research_group</i>
50-111	Nipa	Mirpore	CSE	01-50-111	Khaled	CSE 103	3	A	Mamun	Null	Null
50-222	Nipa	Malibag	CST	02-50-222	Khaled	CSE 103	3	A-	Mamun	Null	Null
50-333	Jabed	Kakrile	CSE	01-50-333	Khaled	CSE 101	2	B	Mamun	Null	Null
50-444	Babu	Palashi	CS	03-50-444	Almas	CSE 101	2	A	Mamun	06	Patt. Recognition
50-555	Chaman	Malibag	ECS	04-50-555	Iqbal	CSE 105	3	A	Masum	Null	Null
50-111	Nipa	Mirpore	CSE	01-50-111	Khaled	CSE 105	3	A+	Masum	Null	Null
50-333	Jabed	Kakrile	CSE	01-50-333	Khaled	CSE 105	3	A	Masum	Null	Null
50-444	Babu	Palashi	CS	03-50-444	Almas	CSE 105	3	B	Masum	06	Patt. Recognition
50-555	Chaman	Malibag	ECS	04-50-555	Iqbal	CSE 106	2	A+	Haque	Null	Null
50-666	Aumy	Palashi	CSE	01-50-666	Khaled	CSE 106	2	A	Haque	Null	Null
50-111	Nipa	Mirpore	CSE	01-50-111	Khaled	CSE 106	2	A	Haque	Null	Null
50-111	Nipa	Mirpore	CSE	01-50-111	Khaled	CSE 108	2	A	Almas	Null	Null
40-888	Mostafa	M. Pore	EEE	05-40-888	Bakir	EEE 101	3	A	Hasib	Null	Null
40-999	Papri	Maghbazar	EEE	05-40-999	Bakir	EEE 101	3	B	Hasib	07	Sig. Processing
40-999	Papri	Maghbazar	EEE	05-40-999	Bakir	EEE 101	3	B	Hasib	08	Sig. Processing
40-666	Jashim	Eskaton	EEE	05-40-666	Bakir	EEE 205	3	A	Nasir	Null	Null
40-777	Pallabi	Dhanmondi	EEE	05-40-777	Bakir	EEE 205	3	A	Nasir	Null	Null
90-111	Aman	Palashi	Null	Null	Null	Null	Null	Null	Null	01	G. Theory
90-111	Aman	Palashi	Null	Null	Null	Null	Null	Null	Null	01	Networking
90-111	Aman	Palashi	Null	Null	Null	Null	Null	Null	Null	02	G. Theory
90-111	Aman	Palashi	Null	Null	Null	Null	Null	Null	Null	02	Networking
90-222	Beauty	Palashi	Null	Null	Null	Null	Null	Null	Null	03	G. Theory
90-222	Beauty	Palashi	Null	Null	Null	Null	Null	Null	Null	03	Patt. Recognition
90-333	Mahboob	Mirpore	Null	Null	Null	Null	Null	Null	Null	04	Networking
90-444	Aman	B. Bazar	Null	Null	Null	Null	Null	Null	Null	05	Networking

**Figure 8.1: The university relation**

<i>Std_id</i>	<i>Std_name</i>	<i>Std_address</i>	<i>department</i>	<i>advisor_id</i>	<i>head_name</i>	<i>course #</i>	<i>credit</i>	<i>gpa</i>	<i>teacher</i>
50-111	Nipa	Mirpore	CSE	01-50-111	Khaled	CSE 103	3	A	Mamun
50-222	Nipa	Malibag	CST	02-50-222	Khaled	CSE 103	3	A-	Mamun
50-333	Jabed	Kakrile	CSE	01-50-333	Khaled	CSE 101	2	B	Mamun
50-444	Babu	Palashi	CS	03-50-444	Almas	CSE 101	2	A	Mamun
50-555	Chaman	Malibag	ECS	04-50-555	Iqbal	CSE 105	3	A	Masum
50-111	Nipa	Mirpore	CSE	01-50-111	Khaled	CSE 105	3	A+	Masum
50-333	Jabed	Kakrile	CSE	01-50-333	Khaled	CSE 105	3	A	Masum
50-444	Babu	Palashi	CS	03-50-444	Almas	CSE 105	3	B	Masum
50-555	Chaman	Malibag	ECS	04-50-555	Iqbal	CSE 106	2	A+	Haque
50-666	Aumy	Palashi	CSE	01-50-666	Khaled	CSE 106	2	A	Haque
50-111	Nipa	Mirpore	CSE	01-50-111	Khaled	CSE 106	2	A	Haque
50-111	Nipa	Mirpore	CSE	01-50-111	Khaled	CSE 108	2	A	Almas
40-888	Mostafa	M. Pore	EEE	05-40-888	Bakir	EEE 101	3	A	Hasib
40-999	Papri	Maghbazar	EEE	05-40-999	Bakir	EEE 101	3	B	Hasib
40-666	Jashim	Eskaton	EEE	05-40-666	Bakir	EEE 205	3	A	Nasir
40-777	Pallabi	Dhanmondi	EEE	05-40-777	Bakir	EEE 205	3	A	Nasir
40-777	Jalal	Palashi	ME	06-40-777	Shafiqul	Me 109	3	C	Habib
40-666	Akbar	Palashi	ME	06-40-666	Shafiqul	Me 109	3	B	Habib

**Figure 8.2: The *student* relation**

<i>std_id</i>	<i>Std_name</i>	<i>Std_address</i>	<i>borrow_number</i>	<i>Research_group</i>
50-444	Babu	Palashi	06	Pattern Recog.
40-999	Papri	Maghbazar	07	Sig. Processing
40-999	Papri	Maghbazar	08	Sig. Processing
90-111	Aman	Palashi	01	G. Theory
90-111	Aman	Palashi	01	Networking
90-111	Aman	Palashi	02	G. Theory
90-111	Aman	Palashi	02	Networking
90-222	Beauty	Palashi	03	G. Theory
90-222	Beauty	Palashi	03	Pattern Recog
90-333	Mahboob	Mirpore	04	Networking
90-444	Aman	Bakshi Bazar	05	Networking

**Figure 8.3: The *borrow* relation**

<i>std_id</i>	<i>Std_name</i>	<i>std_address</i>	<i>department</i>	<i>advisor_id</i>	<i>head_name</i>
50-111	Nipa	Mirpore	CSE	01-50-111	Khaled
50-222	Nipa	Malibag	CST	02-50-222	Khaled
50-333	Jabed	Kakrile	CSE	01-50-333	Khaled
50-444	Babu	Palashi	CS	03-50-444	Almas
50-555	Chaman	Malibag	ECS	04-50-555	Iqbal
50-666	Aurny	Palashi	CSE	01-50-666	Khaled
40-888	Mostafa	M. Pore	EEE	05-40-888	Bakir
40-999	Papri	Maghbazar	EEE	05-40-999	Bakir
40-666	Jashim	Eskaton	EEE	05-40-666	Bakir
40-777	Pallabi	Dhanmondi	EEE	05-40-777	Bakir
40-777	Jalal	Palashi	ME	06-40-777	Shafiqul
40-666	Akbar	Palashi	ME	06-40-666	Shafiqul

**Figure 8.4: The *student\_info* relation**

<i>Std_id</i>	<i>course #</i>	<i>gpa</i>
50-111	CSE 103	A
50-222	CSE 103	A-
50-333	CSE 101	B
50-444	CSE 101	A
50-555	CSE 105	A
50-111	CSE 105	A+
50-333	CSE 105	A
50-444	CSE 105	B
50-555	CSE 106	A+
50-666	CSE 106	A
50-111	CSE 106	A
50-111	CSE 108	A
40-888	EEE 101	A
40-999	EEE 101	B
40-666	EEE 205	A
40-777	EEE 205	A
40-777	Me 109	C
40-666	Me 109	B

**Figure 8.5: The *student\_course* relation**



<i>Course #</i>	<i>Credit</i>	<i>Teacher</i>
CSE 103	3	Mamun
CSE 101	2	Mamun
CSE 105	3	Masum
CSE 106	2	Haque
CSE 108	2	Almas
EEE 101	3	Hasib
EEE 205	3	Nasir
Me 109	3	Habib

**Figure 8.6: The *course* relation**

In the first group we got NULL values on attributes *borrow\_number* and *research\_group*. And we got definite values on attribute *std\_id*, *std\_name*, *std\_address* *department*, *advisor\_id*, *head\_name*, *course\_number*, *course\_credit*, *gpa*, *teacher*. We have discarded attributes *borrow\_number* and *research\_group* from this group due to the presence of NULL values on them. Taking the remaining attributes we have defined a new scheme called *Student-Scheme* as

*Student-Scheme*=(*std\_id*, *std\_name*, *std\_address* *department*, *advisor\_id*, *head\_name*, *course\_number*, *course\_credit*, *gpa*, *teacher*)

and *student* relation as

*student* (*std\_id*, *std\_name*, *std\_address* *department*, *advisor\_id*, *head\_name*, *course\_number*, *course\_credit*, *gpa*, *teacher*).

In the second group we got NULL values on attributes *advisor\_id*, *department*, *head\_name*, *course\_number*, *course\_credit*, *gpa*, *teacher*. And we got definite values on attributes *std\_id*, *std\_name*, *std\_address* *borrow\_number*, *research\_group*. We have discarded attributes *advisor\_id*, *department*, *head\_name*, *course\_number*, *course\_credit*, *gpa*, *teacher* from this group due to the presence of NULL values on them. Taking the remaining attributes we have defined a new scheme called *Borrow Scheme* as

$Borrow\ Scheme = (std\_id, std\_name, std\_address, borrow\_number, research\_group)$

and *borrow* relation as

$borrow(std\_id, std\_name, std\_address, borrow\_number, research\_group)$ .

We have distributed tuples on these newly defined *student* and *borrow* relations so that no tuple has NULL value on any one of its attributes. The *student* relation is shown in the Figure 8.2 and the *borrow* relation is shown in the Figure 8.3.

Applying Algorithm 6.1 and Algorithm 6.3 on *student* relation we got the following set of functional and multi-valued dependencies

Functional dependencies

$std\_id \rightarrow std\_name$   
 $std\_id \rightarrow std\_address$   
 $std\_id \rightarrow department$   
 $std\_id \rightarrow advisor\_id$   
 $std\_id \rightarrow head\_name$   
 $std\_name, std\_address \rightarrow std\_id$   
 $std\_id, department \rightarrow advisor\_id$   
 $advisor\_id \rightarrow department$   
 $std\_id, course\_number \rightarrow gpa$   
 $course\_number \rightarrow course\_credit$   
 $course\_number \rightarrow teacher$

Multi-valued dependencies

Nil.

At these stage we got the complete set of dependencies on *student* relation. Applying Algorithm 7.1 we have merged those functional dependencies whose antecedent attribute/attributes is/are same into a single dependency. We also listed multi-valued dependencies separately. And we got the following set

$std\_id \rightarrow std\_name, std\_address, department, advisor\_id, head\_name.$   
 $std\_name, std\_address \rightarrow std\_id$   
 $std\_id, department \rightarrow advisor\_id$   
 $advisor\_id \rightarrow department$   
 $std\_id, course\_number \rightarrow gpa$   
 $course\_number \rightarrow course\_credit, teacher$

Applying Algorithm 7.2 we have constructed relational database or tables directly in 2NF and their keys and the set of dependencies that holds on the tables respectively. We got three tables as follows:-

1. *Student\_info* Table/Relation

Table/Relation:

*student\_info*(*std\_id, std\_name, std\_address, department, advisor\_id, head\_name*)

Key: *std\_id*

Set of dependencies:

$std\_id \rightarrow std\_name, std\_address, department, advisor\_id,$   
 $head\_name$   
 $std\_name, std\_address \rightarrow std\_id$   
 $std\_id, department \rightarrow advisor\_id$   
 $advisor\_id \rightarrow department$

The *student\_info* relation is shown in the Figure 8.4

2. *Student\_Course* Table/Relation

Table/Relation:

*student\_course*(*std\_id, course\_number, gpa*)

Key: *std\_id, course\_number*

Set of dependencies:

$$std\_id, course\_number \rightarrow gpa$$

The *student\_course* relation is shown in the Figure 8.5

### 3. Course Table/Relation

Table/Relation:

$$course(course\_number, course\_credit, teacher)$$

Key: *course\_number*

Set of dependencies:

$$course\_number \rightarrow course\_credit, teacher$$

The *course* relation is shown in the Figure 8.6.

We observed that for the functional dependency *advisor\_id*  $\rightarrow$  *department* the *student\_info* relation is not in BCNF, since *advisor\_id* is not a key, and not in 3NF either, since *department* is not part of a key; and for the functional dependency *std\_id, department*  $\rightarrow$  *advisor\_id* the *student\_info* relation is not in BCNF, since *std\_id, department* is not a key, and not in 3NF either, since *advisor\_id* is not part of a key. For this reason, we got several duplications of data in the Figure 8.4.

Applying Algorithm 5.2, we decomposed *student\_info* relation into two new relations as

i) The *student\_regular* relation

Table/Relation:

$$student\_regular(std\_id, std\_name, std\_address, advisor\_id, head\_name)$$

Key: *std\_id*

Set of dependencies:

$$std\_id \rightarrow std\_name, std\_address, advisor\_id, head\_name,$$
$$std\_name, std\_address \rightarrow std\_id$$

ii) The *advisor* relation

Table/Relation:

*advisor*(*std\_id*, *advisor\_id*, *department*)

Key: *std\_id*, *department*

Set of dependencies:

*std\_id*, *department*  $\rightarrow$  *advisor\_id*

*advisor\_id*  $\rightarrow$  *department*

Applying Algorithm 5.1 we have tested that the relations *student\_regular* and *advisor* are dependency preserving. We observe that though the relation *student\_regular* is in BCNF but the relation *advisor* is not in BCNF due to the functional dependency *advisor\_id*  $\rightarrow$  *department*. Since, *advisor\_id* is not a key in the relation *advisor*. If we try to give input in Algorithm 5.2 by swapping the order of the functional dependencies *std\_id*, *department*  $\rightarrow$  *advisor\_id* and *advisor\_id*  $\rightarrow$  *department*, i.e., the functional dependency *advisor\_id*  $\rightarrow$  *department* before the functional dependency *std\_id*, *department*  $\rightarrow$  *advisor\_id*, we could generate the *advisor\_alt* relation instead of *advisor* relation.

iiia) The *advisor\_alt* relation

Table/Relation:

*advisor\_alt*(*advisor\_id*, *department*)

Key: *advisor\_id*

Set of dependencies:

*advisor\_id*  $\rightarrow$  *department*

Applying Algorithm 5.1 we found that the relation *advisor\_alt* is not dependency preserving, since the functional dependency *std\_id, department*  $\rightarrow$  *advisor\_id* is lost. Alternatively, if we try to apply Algorithm 5.2 on the relation *advisor* we could have the relation *advisor\_alt* and the relation *department* as

iiib) The *department* relation

Table/Relation:

*department (std\_id, department)*

Key: *std\_id, department*

Set of dependencies:

Nil

Applying Algorithm 5.1 we have tested that these decompositions are not dependency preserving. Since, the functional dependency *std\_id, department*  $\rightarrow$  *advisor\_id* is lost. Instead of applying Algorithm 5.2 on the *student\_info* relation, we had applied Algorithm 5.3 and got the relations *student\_regular* and *advisor* as described above. At this stage, the relations *student\_regular* and *advisor* are in 3NF as well as dependency preserving.

The relations *student\_course* and *course* are in BCNF. Applying Algorithm 5.1 we have tested that these relations are also dependency preserving.

Applying Algorithm 6.1 and Algorithm 6.3 on borrow relation we got following set of functional and multi-valued dependencies

Functional dependencies

*std\_id*  $\rightarrow$  *std\_name*

*std\_id*  $\rightarrow$  *std\_address*

*borrow\_number*  $\rightarrow$  *std\_id*

*borrow\_number*  $\rightarrow$  *std\_name*

*borrow\_number*  $\rightarrow$  *std\_address*

Multi-valued dependencies

$$std\_id \twoheadrightarrow research\_group$$

At these stage we got the complete set of dependencies on *borrow* relation. Applying Algorithm 7.1 we have merged those functional dependencies whose antecedent attribute/ attributes is/are same into a single dependency. We also listed the multi-valued dependencies separately. And we got the following set

Functional dependencies

$$std\_id \rightarrow std\_name, std\_address,$$
$$borrow\_number \rightarrow std\_id, std\_name, std\_address$$

Multi-valued dependencies

$$std\_id \twoheadrightarrow research\_group$$

Applying Algorithm 7.2 we have constructed relational database or tables directly in 2NF and their keys and the set of dependencies that holds on the table respectively. We got one table as follows:-

4) The *borrow-1* relation

Table/Relation:

$$Borrow\_1 (, borrow\_number, std\_id, std\_name, std\_address)$$

Key: *borrow\_number*

Set of dependencies:

$$std\_id \rightarrow std\_name, std\_address$$
$$borrow\_number \rightarrow std\_name, std\_address, std\_id$$

The relation *borrow-1* is shown in the Figure 8.7

<i>std_id</i>	<i>Std_name</i>	<i>Std_address</i>	<i>Borrow_number</i>
50-444	Babu	Palashi	06
40-999	Papri	Maghbazar	07
40-999	Papri	Maghbazar	08
90-111	Aman	Palashi	01
90-111	Aman	Palashi	02
90-222	Beauty	Palashi	03
90-333	Mahboob	Mirpore	04
90-444	Aman	Bakshi Bazar	05

**Figure 8.7: The *borrow-1* relation**

Applying Algorithm 5.4 we got the relation *student\_research* as follows

5) The *student\_research* relation

Table/Relation:

*student\_research ( std\_id, research\_group)*

Set of dependencies:

*std\_id*  $\rightarrow\rightarrow$  *research\_group*

The relation *borrow-1* is not in BCNF due to the presence of functional dependency *std\_id*  $\rightarrow\rightarrow$  *std\_name, std\_address*. Applying Algorithm 5.3 we got two relations as follows

4-i) The *student\_borrow* relation

Table/Relation:

*Student\_borrow (std\_id, borrow\_number)*

Key: *borrow\_number*

Set of dependencies:

*borrow\_number*  $\rightarrow$  *std\_id*



4-ii) The *student\_special* relation

Table/Relation:

*student\_special* (*std\_id*, *std\_name*, *std\_address*)

Key: *std\_id*

Set of dependencies:

*std\_id* → *std\_name*, *std\_address*

Both the relation *student\_borrow* and *student\_special* are in 4NF. Applying Algorithm 5.1 we have tested that these relations are also dependency preserving

The resultant relations *student\_regular*, *advisor*, *student\_course*, *course*, *student\_borrow*, *student\_special* and *student\_research*, are depicted in the Figure 8.8, Figure 8.9, Figure 8.10, Figure 8.11, Figure 8.12, Figure 8.13, Figure 8.14 respectively.

<i>Std_id</i>	<i>Std_name</i>	<i>std_address</i>	<i>advisor_id</i>	<i>Head_name</i>
50-111	Nipa	Mirpore	01-50-111	Khaled
50-222	Nipa	Malibag	02-50-222	Khaled
50-333	Jabed	Kakrile	01-50-333	Khaled
50-444	Babu	Palashi	03-50-444	Almas
50-555	Chaman	Malibag	04-50-555	Iqbal
50-666	Aumy	Palashi	01-50-666	Khaled
40-888	Mostafa	M. Pore	05-40-888	Bakir
40-999	Papri	Maghbazar	05-40-999	Bakir
40-666	Jashim	Eskaton	05-40-666	Bakir
40-777	Pallabi	Dhanmondi	05-40-777	Bakir
40-777	Jalal	Palashi	06-40-777	Shafiqul
40-666	Akbar	Palashi	06-40-666	Shafiqui

**Figure 8.8 : The *student\_regular* relation**

<i>std_id</i>	<i>department</i>	<i>advisor_id</i>
50-111	CSE	01-50-111
50-222	CST	02-50-222
50-333	CSE	01-50-333
50-444	CS	03-50-444
50-555	ECS	04-50-555
50-666	CSE	01-50-666
40-888	EEE	05-40-888
40-999	EEE	05-40-999
40-666	EEE	05-40-666
40-777	EEE	05-40-777
40-777	ME	06-40-777
40-666	ME	06-40-666

**Figure 8.9: The *advisor* relation**

<i>std_id</i>	<i>course #</i>	<i>gpa</i>
50-111	CSE 103	A
50-222	CSE 103	A-
50-333	CSE 101	B
50-444	CSE 101	A
50-555	CSE 105	A
50-111	CSE 105	A+
50-333	CSE 105	A
50-444	CSE 105	B
50-555	CSE 106	A+
50-666	CSE 106	A
50-111	CSE 106	A
50-111	CSE 108	A
40-888	EEE 101	A
40-999	EEE 101	B
40-666	EEE 205	A
40-777	EEE 205	A
40-777	Me 109	C
40-666	Me 109	B

**Figure 8.10 : The *student\_course* relation**

<i>Course #</i>	<i>Credit</i>	<i>Teacher</i>
CSE 103	3	Mamun
CSE 101	2	Mamun
CSE 105	3	Masum
CSE 106	2	Haque
CSE 108	2	Almas
EEE 101	3	Hasib
EEE 205	3	Nasir
Me 109	3	Habib

**Figure 8.11: The *course* relation**

<i>std_id</i>	<i>Borrow_number</i>
50-444	06
40-999	07
40-999	08
90-111	01
90-111	02
90-222	03
90-333	04
90-444	05

**Figure 8.12: The *student\_borrow* relation**

<i>std_id</i>	<i>std_name</i>	<i>Std_address</i>
50-444	Babu	Palashi
40-999	Papri	Maghbazar
90-111	Aman	Palashi
90-222	Beauty	Palashi
90-333	Mahboob	Mirpore
90-444	Aman	Bakshi Bazar

**Figure 8.13: The *student\_special* relation**

<i>std_id</i>	<i>Research_group</i>
50-444	Pattern Recog.
40-999	Sig. Processing
90-111	G. Theory
90-111	Networking
90-222	G. Theory
90-222	Pattern Recog
90-333	Networking
90-444	Networking

**Figure 8.14 : The *student\_research* relation**

## Chapter 9

# Conclusions

In this research work Relational Database design is viewed as a Machine Learning problem. Initially, a relational database of a single relation containing all the attributes, those have been collected from the target system, have been defined. Examples have been collected on this relation from the target system. Though the presence of NULL values in different attributes of a particular example has made it difficult to use machine learning algorithms for characterizing the relation this has provided with the hints of initial decomposition of the relation. We have divided the set of examples into several groups putting the examples in the same group those have NULL values on the same set of attributes. For each group of examples we have defined a new relation discarding those attributes which have NULL values, and have distributed examples on those newly defined relations so that no relation has NULL value in any one of its attributes. Machine learning algorithms have been used on these relations to get the corresponding set of functional and multi valued dependencies for each relation. Derived set of functional dependencies has been used in an algorithm to construct relations in 2NF and corresponding set of keys and dependencies. Several algorithms have been used to normalize these 2NF relations. These normalization algorithms have normalized all the 2NF relations either into 3NF or BCNF. For the presence of multi valued dependency separate algorithm has been used to normalize the relation into 4NF. We have used popular normalization algorithms in this research work. We have also used already established machine learning algorithms for characterizing relations. There are numerous algorithms in this connection. We have tried few of them. Others can be tried to get better results. These machine learning algorithms can also be improved to cope with the presence of Null values in a relation. We have proposed a decomposition method using Null values in a relation and algorithms for constructing 2NF relations and corresponding set of keys and dependencies. We have gotten satisfactory results from these method and algorithms. Further research can be done to construct 3NF or BCNF or 4NF relations directly avoiding the creation of 2NF relations.

## References

- [1] Angluin D. & Smith C. H., 'Inductive inference: theory and methods', *Computing Surveys*, 1983.
- [2] Armstrong W.W., 'Dependency structures of data base relationships', in *Proc. IFIP 74*, North Holland, Amsterdam, 1974.
- [3] Avison D. E., Fitzgerald G., *Information Systems Development: Methodologies, Techniques and Tools*, ALFRED WALLER LTD, Orchards, Fawley, Henley-on-Thames, 1988.
- [4] Beeri C., Dowd M., Fagin R. & Statman R., 'On the structure of Armstrong relations for functional dependencies', *JACM*, 1984.
- [5] Beeri C. & Vardi M. Y., 'The implication problem for data dependencies', in *Proc. 8th Int. Conf. On Automata, Languages, and Programming*, Lecture note in Computer Science 115, Springer-Verlag, New York, 1981.
- [6] Crinnion J., *Evolutionary Systems Development: a Practical Guide to the use of Prototyping within a Structured System Methodology*, PITMAN PUBLISHING, 128 Long Acre, London, 1991.
- [7] Flach P. A. & Veelenturf L. P. J., *Concept learning from examples: theoretical foundations*, ITK Research Report 2, Institute for Language Technology & Artificial Intelligence, Tilburg University, the Netherlands, 1989.
- [8] Flach P. A., *Second-order inductive learning*, ITK Research Report 10, Institute for Language Technology & Artificial Intelligence, Tilburg University, the Netherlands; a preliminary version appeared in *Proc. Workshop on Analogical and Inductive Inference AII'89*, K. P. Jantke (ed), Lecture Notes in Computer Science 397, Springer-Verlag Berlin, 1989.
- [9] Flach P. A., 'Inductive Characterization of Database Relations', in *Proc. International Symposium on Methodologies for Intelligent Systems* Z.W. Ras, M. Zemankowa & M. L. Emrich (eds), North-Holland, Amsterdam. Full version appeared as ITK Research Report no 23, 1990.
- [10] Gallaire H., Minker J. & Nicolas J. M., 'Logic and databases: a deductive approach', *Computing surveys*, 1984.

- [11] Genesereth M. R. & Nilsson N. J., *Logical foundations of Artificial Intelligence*, Morgan Kaufmann, Los Altos, 1987.
- [12] Gold E. M., 'Language Identification in the limit', *Information and Control*, 1967.
- [13] Grant J. & Jacobs B. E., 'On the family of generalized dependency constraints', *JACM*, 1982.
- [14] Grant J., *Logical Introduction to Databases*, Harcourt Brace Jovanovich, Publishers and its subsidiary, Academic Press, 1987.
- [15] Kabir M. H., Rahman C. M., 'Database Normalization using Machine Learning Techniques', *NCCI 97*, Bangladesh, 1997.
- [16] Korth H. F., Silberschatz A., *Database System Concepts*, McGraw-Hill Book Company, Singapore, 1986.
- [17] Laird P.D., 'Inductive inference by refinement', *Proceedings AAAI-86*, 1986.
- [18] Laird P.D., *Learning from good and bad data*, Kluwer, Boston, 1988.
- [19] Maier D., *The theory of relational databases*, Computer Science Press, Rockville, 1983.
- [20] Mannila H. & Ra'iha' K. J., 'Design by example; an application of Armstrong relations', *J. Comp. Syst Sc.* 33, 1986.
- [21] Mitchell T. M., 'Generalization as search', *Artificial Intelligence* 18:2, 1982.
- [22] Plotkin G. D., 'A note on inductive generalization', in *Machine Intelligence 5*, B. Meltzer & D. Michie (eds.), Edinburgh University Press, Edinburg, 1970.
- [23] Plotkin G. D., 'A further note on inductive generalization', in *Machine Intelligence 6*, B. Meltzer & D. Michie (eds.), Edinburgh University Press, Edinburg, 1971.
- [24] Reynolds J. C., 'Transformational systems and the algebraic structure of atomic formulas', in *Machine Intelligence 5*, B. Meltzer & D. Michie (eds.), Edinburgh University Press, Edinburg, 1970.
- [25] Sagiv Y., Delobel C., Parker D. S., Fagin JR. & R., 'An equivalence between relational database dependencies and a fragment of propositional logic', *JACM*, 1981.
- [26] Shapiro E. Y., *Inductive inference of theories from facts*, Techn. Rep. 192, Comp. Sc. Department, Yale University, 1981.
- [27] Shapiro E. Y., *Algorithmic program debugging*, MIT Press, 1981.

