

**NORMALIZATION IN RELATIONAL DATABASES :  
AUTOMATION UP TO FOURTH NORMAL  
FORM USING PROLOG**

603

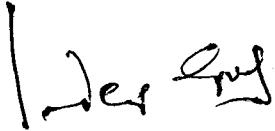
THESIS SUBMITTED BY  
**SANDEEP GOEL**  
IN PARTIAL FULFILMENT OF THE  
REQUIREMENTS FOR THE DEGREE OF  
**MASTER OF TECHNOLOGY**  
IN  
**COMPUTER SCIENCE**


SCHOOL OF COMPUTER AND SYSTEMS SCIENCES  
JAWAHARLAL NEHRU UNIVERSITY  
NEW DELHI  
JUNE 1990


CERTIFICATE

This is to certify that the thesis entitled 'Normalization in Relational Databases : Automation up to Fourth Normal Form using Prolog', being submitted by me to Jawaharlal Nehru University in the partial fulfilment of the requirements for the award of the degree of **Master of Technology**, is a record of original work done by me under the supervision of **Dr. P. C. Saxena**, Associate Professor, School of Computer and Systems Sciences, Jawaharlal Nehru University, during the year 1989 - 90.

The results reported in this thesis have not been submitted in part or full to any other University or Institute for the award of any degree or diploma etc.

  
SANDEEP GOEL

  
**Prof. N. P. Mukherjee**  
Dean,  
School of Computer and  
Systems Sciences,  
J.N.U.  
New Delhi.

  
**Dr. P. C. Saxena**  
Associate Professor,  
School of Computer and  
Systems Sciences,  
J.N.U.,  
New Delhi.

### ACKNOWLEDGEMENTS

I express my sincere thanks to Dr. P. C. Saxena, Associate Professor, School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi for suggesting such brilliant problem. I am indebted to him for his personal involvement with my project and his immense and eloquent guidance which has been indispensable in bringing about a successful fulfilment of the project.

I am also grateful to Dr. Saxena for providing me with his invaluable notes and papers related with the topic and also guiding me in my lookout for proper references.

I also want to express my thanks to my father Dr. G. C. Goel, Reader, Department of Mathematics, University of Delhi, for his constant encouragement and useful exchange of ideas with me throughout my project work. I would like to take the opportunity to thank my mother Smt. Sneha Lata also, for providing the moral support that I needed during my research and for taking good care of me in the hectic days of compilation of this dissertation.

**Sandeep Goel**

## CONTENTS

Certificate

Acknowledgements

1	Introduction	1
1.1	Introduction	1
1.2	Normalization and Prolog	2
1.3	Organization of work	4
2	Relational Data bases and Normalization	6
2.1	Introduction	6
2.2	Data base models	7
2.3	Relational Data bases and inconsistency problems	9
2.4	Normalization	13
2.4.1	Functional Dependencies	13
2.4.2	1NF	16
2.4.3	2NF	17
2.4.4	3NF	19
2.4.5	BCNF	21
3	Further Normalization	25
3.1	Introduction	25
3.2	Multivalued Dependencies	26
3.3	Fourth Normal Form (4NF)	30

4	Normalization Algorithms	36
4.1	Introduction	36
4.2	Synthesis and Decomposition approaches	36
4.3	Bernstein's Algorithm for 3NF	38
4.4	Decomposition Algorithms for BCNF and 4NF	41
5	The Prolog Program for automatic Normalization up to 4NF	49
5.1	Introduction	49
5.2	The Program	49
5.2.1	Declarations	49
5.2.2	Utility clauses	52
5.2.3	Closure	54
5.2.4	Extraneous attributes	55
5.2.5	Redundant FDs	57
5.2.6	Bernstein's Algorithm	58
5.2.7	Tsou & Fischer 's Algorithm	65
5.2.8	Minimizing a decomposition	67
5.2.9	Tanaka 's Algorithm for 4NF	68
	Appendix - I	75 - 97
	Appendix - II	98 - 110
	References	111 - 112

## CHAPTER - 1

### INTRODUCTION

#### 1.1 INTRODUCTION

The database management systems (DBMSs) that emerged in the early 1970s, have become an integral part of most business corporations. The DBMS technology has grown enormously in the two decades but despite its wide use and its indispensable place in big organizations, the art of **database design** has remained obscure to most of the users.

In fact, a lot of software is available in the market on database manipulation and use but all of these packages require human intelligence only for the initial creation of the database. It is the database designer who is supposed to go through the requirements of the problem and then design the database accordingly. The use of the software packages only starts after that viz. in data entry, data retrieval, query processing etc. Of all the database structures the relational database model is by far most widely used and is most widely accepted as the standard database model. A lot of software packages on the relational database are available in the market that are invaluable to the DBMSs. But all of these packages have one problem in common viz. they provide no assistance to the designer in the initial stages of design of the database.

Thus despite tremendous developments in the DBMS technology, one very fundamental problem remains without a proper and sure solution viz. given a body of data to be represented in a database, how to decide on a suitable logical structure for that data or in other words, how to decide what relations are needed and what their attributes should be ? This is the database design problem which one faces as the first hurdle in installing a data base management system.

Attempt is made in this work to help the database designer get rid of this basic database-design problem by developing a Prolog program that suggests to the designer a trouble free (Normalized) logical structure to the database. The only information required by the program is the list of various dependencies (functional as well as multi-valued) that exist in the problem and which are not hard to be made out by the designer who has gone through the database problem properly.

## **1.2 NORMALIZATION AND PROLOG**

Due to dependencies among various attributes of a relation, any database structure suffers from certain problems if not properly designed. The problems that are most likely to occur are update-anomalies and data inconsistencies. The first step towards solving the design problem was the introduction to the concept of Normal Forms. E. F. Codd was the pioneer in this field as he originally defined the first, second and third normal forms in 1971. Since then numerous normal forms have been

defined and it has been proved that the fifth normal form is the ultimate normal form as it removes all the inconsistency problems with the relational structure.

Normalization procedure is one in which we start with some given relation together with the information about its various constraints i.e. the dependencies among its attributes (functional dependencies, multi-valued dependencies and the join-dependencies), and then we systematically reduce that relation to a collection of smaller relations that are together equivalent to the original relation yet in some way preferable to it. In fact, the new smaller relations are in a higher Normal Form than the original relation and thus more preferable.

The various normal forms are first, second and third normal forms (1NF, 2NF and 3NF), Boyce-Codd normal form (BCNF), fourth normal form (4NF) and fifth normal form (5NF), in that order. The higher the normal form of a relation the more preferable it is. The BCNF is the ultimate normal form in the case of functional dependencies only. But this form is not free from problems in case the multi-valued dependencies are also present among the attributes. It is the 4NF which is the desirable form in the case of presence of multi-valued dependencies. The 5NF is a step further which solves the problems caused by join dependencies also. In this paper we don't consider the case of join dependencies and present a program that does normalization up to 4NF.

Prolog was the language chosen for writing the program



for this automatic normalization of a given relation database schema and the reason was that Prolog is the only language that provides one with tools to write an 'intelligent' program most essential in a difficult problem like this. The prolog approach is to describe known facts about a problem and then let the computer solve it by itself through backtracking, rather than to prescribe the sequence of steps to be taken by the computer to solve the problem. It is this feature of Prolog that gives its programs the feature of 'intelligence'. Moreover the Prolog is especially suited to the way the relations can be represented and manipulated in it and these are the reasons that made Prolog an obvious choice for this project.

The machine used was IBM compatible PC/XT, 640K, 8.58 MHz and the software used was Borland's Turbo Prolog, version 2.0.

### **1.3 ORGANIZATION OF WORK**

Chapter-2 gives first the brief introduction to various database models viz. relational, hierarchical and network models. It then discusses the relational structure of databases in detail. Problems caused by functional dependencies in relational databases are then discussed followed by an account of 1NF, 2NF, 3NF and BCNF.

Chapter-3 discusses further normalization in relational databases as it introduces the concept of multivalued dependencies and the problems caused by these. It then discusses the fourth normal form (4NF) and shows how it solves these problems.

Chapter-4 introduces the two approaches towards solving design-problems viz. the Synthesis approach and the Decomposition approach. First, it takes up the Synthesis approach for normalization up to BCNF in case of functional dependencies only and gives the algorithms for the same. Then it discusses also the Decomposition approach for normalization in case of multi-valued dependencies and discusses an algorithm for obtaining 4NF.

Chapter-5 discusses the actual program written in Prolog for automatic normalization upto 4NF.

In the end, the appendices list the Prolog program, a few examples showing its use and lastly the references.

## CHAPTER - 2

### RELATIONAL DATABASES AND NORMALIZATION

#### 2.1 INTRODUCTION

A data base as suggested by James Martin can be defined as a collection of interrelated data stored together with controlled redundancy to serve one or more applications in an optimal fashion; the data are stored so that they are independent of programs which use the data; a common and controlled approach is used in adding new data and modifying and retrieving existing data within the data base. A data base system is different from the orthodox files-of-records system in that it allows the same collection of data to serve as many applications as required. Thus a data base may be conceived of as a repository of information that permits not only retrieval and continuous modification of data but also answers to various queries put forward by the management from time to time.

The logical design of a database may be based on any of several known models. The three best known data base models are the relational, the hierarchical and the network approach models. Section 2.2 discusses each of these briefly. As we are concerned with only the relational data bases in this thesis, section 2.3

discusses relational data bases in more details and also explains the inconsistency problems caused in these data bases due to the presence of various functional dependencies. Section 2.4 formally defines the functional dependencies and the normal forms : first, second, third and Boyce-Codd.

## 2.2 DATA BASE MODELS

Data bases are most conveniently categorized into relational, hierarchical and network types depending upon the type of data structure used by the data base. In a relational database the data is organized into tables. A table is a two dimensional rectangular array with each column representing a particular field of the record. The rows contain the actual data entries. The columns in a table are homogeneous i.e. in any column all items are of the same kind. A data base may consist of more than one tables. In fact a data base represented by only one table may give rise to redundancy, inconsistency and updating problems. To get rid of these problems a relational data base is converted into a number of smaller tables instead of a single big table. This is called normalization. Proper normalization essentially removes inconsistency and updating problems but is incapable of fully eliminating redundancy of records. In fact, normalization itself gives rise to some redundancy which can be termed as the controlled redundancy intentionally introduced in the data base.

In a hierarchical data base the data is represented by a

simple tree structure. Each tree consists of a record at the top which is known as the 'root'. This root record may have a number of dependent record types. Every record type of the dependent records may have a number of records, each of which may again have a number of dependent record types in turn. Thus we can say that in hierarchical structure, every record may have any number of children but any child record can have only one parent. The hierarchical structure also contains 'links' which connect a parent node to a child node. These links have restriction on their directions as they can only point from a higher level to a smaller level. Quite similar to a hierarchical database, a network data base also consists of records connected with links. However the data structure in the network approach is a more general one as it need not follow a simple tree structure. Rather it contains a mesh structure in which there are no restrictions on the fixation of links. A link may be connected between any two records in any levels and in any direction. Thus any given record occurrence may have any number of immediate parents unlike the case in the hierarchical system. Thus the network approach allows one to model a many-to-many correspondence more directly than do the other two approaches. The network structure requires least of redundancy but it gives rise to many other complex problems.

Design problem or normalization problem exists in hierarchical and network database systems also, but in this thesis we concentrate only on normalization in relational data

base systems. The same is treated in the following text.

### **2.3 RELATIONAL DATABASES AND INCONSISTENCY PROBLEMS**

As mentioned earlier the data in a relational database is arranged in the form of tables called relations. The columns of the table correspond each to a unique field and are referred to as attributes. The rows in the table contain the actual data and are referred to as tuples. The number of rows is a variable quantity and changes with time in a dynamic database. Keys are subsets of the attributes of a relation whose values are unique within the relation and thus which can be used to uniquely identify the tuples of the relation. A primary key is minimal i.e. no proper subset of a primary key is by itself a key. There are certain restrictions on these relations which are as follows :

1. All rows should be distinct i.e. no two tuples in a table should contain identical information.
2. Each column in a particular relation should be assigned a distinct name.
3. Relations must be column homogeneous i.e. in any column all the items must be of the same kind.
4. The sequence of both rows and columns in any relation should be immaterial i.e. both the rows and the columns could be viewed in any sequence at any time without affecting either the information content or the semantics of any function using the table.

5. No component of a primary key may be null. This is called the rule of entity integrity.

In its most crude form, a relational data base may contain a single relation containing all the fields as it's columns and the data stored in tuples. It is what is called an un-normalized form of a relational database. Though being the simplest and thus being the most suitable form for information extraction and query processing, the unnormalized form suffers from many disadvantages. To be precise, these problems are redundancy, inserting, deleting and updating problems. To explain these we best consider a practical example.

Let us consider a data base containing information about a child's name, his roll number in the school, marks obtained by him in different subjects in a school test and his parents' names. Let us call it "Child-Marks-Parents" database. Thus the -----  
different fields required in this data base are Child's name (N), Roll No. (R#), Subject (S), Marks (M), Mother's name (MN) and Father's name (FN). One semantic constraint is that no two persons can have the same name. In the unnormalized form, all the fields are put together in a single relation named R. In fig. 2.1 a sample record of this relation R at a particular instance is shown. As we shall explain now it suffers from several problems notably redundancy, inconsistency problems etc. Inconsistency may arise in such a table by any of the fundamental operations like insertion, deletion and updating.

Now we discuss these problems associated with an unnormalized relation like R shown in fig. 2.1, one by one :

**REDUNDANCY PROBLEM.** An unnormalized problem suffers from a lot of uncontrolled redundancy. For example the fact that Montu is the name of the boy who has roll no. 5 is repeated every time there is an entry for R# 5. Also it is unnecessarily repeated in every entry that his parents' names are Manju & S.Gupta. Such redundancy not only causes loss of useful memory space but may also give rise to serious inconsistency problems in the data base.

R

Roll No.	Child's name	sub-ject	Marks	Mother's name	Father's name
R#	N	S	M	MN	FN
1	Pinku	Phys	61	Sneh Lata	G.C.Goel
1	Pinku	Chem	48	Sneh Lata	G.C.Goel
2	Sonu	Math	95	Roopa	Shivendra
3	Guggi	Phys	92	Roopa	Shivendra
3	Guggi	Chem	90	Roopa	Shivendra
4	Guriya	Math	99	Manju	S.Gupta
4	Guriya	Phys	90	Manju	S.Gupta
5	Montu	Phys	85	Manju	S.Gupta
5	Montu	Chem	85	Manju	S.Gupta
5	Montu	Math	92	Manju	S.Gupta

no two persons have the same name

fig. 2.1

Relation R showing its record at a particular instance



**INSERTING PROBLEM.** Suppose we want to enter the fact that Bobby got 65 marks in Chemistry, we cannot enter this until we knew his roll numbers, since by restriction 5 (rule of entity integrity) no component of a primary key may be null and in this case [R#,S] is one of the primary keys. Also say we make another entry for Sonu but with his mother's name printed wrongly, it is going to cause inconsistency in the data base.

**DELETING PROBLEM.** If we delete a particular item from a table like R, we cannot be sure of the safety of all other information that is contained in the table R. For example, there is only one entry for Sonu in fig. 2.1. If we want to delete the entry for Sonu's marks in Mathematics we have no choice but to delete the whole of that entry. This will not only make us lose the knowledge about his roll number but also his parents' names.

**UPDATING PROBLEM.** This is a direct outcome of the redundancy contained in an unnormalized relation like R. For example, the information that Guriya's roll number is 4 is contained in every tuple that contains information about Guriya's. Now if her roll number changes we are faced with either the problem of searching the whole of table R (an any instant, R may contain any number of tuples and also all the entries about Guriya may not be grouped together) to find every tuple containing information about Guriya or the possibility of producing an inconsistent result by say leaving out some of the tuples unmodified.

How these problems connected with an unnormalized relation are solved using normalization, is the subject of discussion in the following section.

## **2.4 NORMALIZATION**

Normalization in a relational data base refers to breaking of bigger relations into a number of smaller relations (having less number of fields) according to some rule so that the new relations are preferable to the original ones in that they solve some of the difficulties faced by the original relations. The smaller relations so obtained are necessarily of a higher normal-form than the original relation. However, this one step of normalization may not solve all the problems in a relation and some of the smaller relations may have to be further normalized in order to get higher and more desirable normal forms. We introduce here the concept of functional dependencies among the attributes of a relation; the concept of multi-valued dependencies being deferred to the next chapter. We also discuss various normal forms viz. first, second, third and Boyce-Codd and show with the help of an example how normalization solves the problem in a relational data base containing functional dependencies only.

### **2.4.1 Functional Dependencies (FDs)**

The concept of functional dependencies among the attributes of a relation is of prime importance in normalization

theory. In fact the very basis of breaking a given relation in a number of smaller relation is functional dependencies (and later multivalued and join dependencies also) among its various attributes only. Functional dependence is defined as follows :

An attribute B of a relation R is said to be functionally dependent on another attribute A of R if and only if each value of A is associated with precisely one value of B i.e. if each value of the attribute A uniquely determines it's corresponding value of the attribute B.

This is denoted as

$$R.A \twoheadrightarrow R.B$$

or more simply as

$$A \twoheadrightarrow B$$

The same definition of functional dependence applies to groups of attributes also. Thus a group of attributes may functionally determine another group of attributes.

Let us now take an example. Reconsider the "Child-Marks-Parents" data base discussed in the previous article. It is clearly mentioned that no two persons have the same name. This means the names of both the parents are uniquely determined by their child's name because each child has only one set of parents. In other words Mother's name (MN) and Father's name (FN) are both dependent on the Child's name (N). Incidentally the reverse is not true because a parent may have more than one child. These functional dependencies may be shown as :

$$R.M \twoheadrightarrow R.MN$$

&  $R.M \twoheadrightarrow R.FN$

Also since no two persons have the same names, a mother's name uniquely determines father's name and vice-versa. Thus we have MN and FN functionally determining each other. Thus,

$R.MN \twoheadrightarrow R.FN$

&  $R.FN \twoheadrightarrow R.MN$

Also marks obtained by a child in a particular subject are unique. So Child name (N) and Subject (S), together, uniquely determine the Marks (M). Thus,

$R.[N,S] \twoheadrightarrow R.M$

Also a child's name and roll number uniquely determine each other. So N and R# are functionally dependent on each other. Or,

$R.R\# \twoheadrightarrow R.N$  and  $R.N \twoheadrightarrow R.R\#$

This last pair of functional dependencies also gives rise to the fact that any attribute that is functionally dependent on N is also functionally dependent on R#. Thus we have the following set of additional functional dependencies :

$R.R\# \twoheadrightarrow R.MN$

$R.R\# \twoheadrightarrow R.FN$

$R.[R\#,S] \twoheadrightarrow R.M$

We can make a functional dependency diagram for the relation R as follows (remember a functional dependency diagram may not show all existing dependencies; it needs show only a minimal set of functional dependencies),

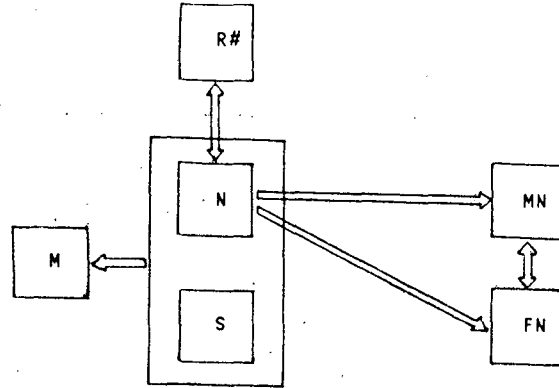


fig. 2.2

Functional Dependency diagram for relation R

A key is a set of attributes of a relation that functionally determines all of the attributes of the relation. If a key is minimal i.e. no proper subset of it possesses the same property it is said to be a primary key of the relation. A relation may have more than one primary keys. Any relation must have at least one key, as the set of all the attributes of a relation is definitely a key. The relation R contains two keys viz. [N,S] and [R#,S].

#### 2.4.2 First Normal Form (1NF)

A relation is said to be in First Normal Form if all of its tuples contain only atomic values for each of their attributes. In other words, all the occurrences of a record must contain the same number of fields in 1NF.

Thus a relation of the type

Child	Subject	Marks
Sonu	Physics	95
	Chemistry	94
	Maths	99
-----		
Guggi	Physics	94
	Chemistry	95

a relation not in 1NF

is not in first normal form. It is to be modified as follows to be in 1NF,

Child	Subject	Marks
Sonu	Physics	95
Sonu	Chemistry	94
Sonu	Maths	99
Guggi	Physics	94
Guggi	Chemistry	95

a relation in 1NF

We see that the relation R in fig. 2.1 is in 1NF. First normal form is the first requirement of any relation because as is clear from the above example it is very simple to convert any given relation into one in 1NF and, also because it simplifies the further database operations and manipulations to a great extent.

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

A relation R is said to be in Second Normal Form if no non-key attribute of it is functionally dependent on a proper subset of any primary key of the relation. A non-key attribute

is one which is not part of any of the primary keys of the relation. Thus, in other words, only a primary key and not any of its proper subsets should functionally determine any non-key attribute.

For example, the relation R of the "Child-Marks-Parents" data base shown in fig. 2.1 is not in 2NF. The reason is that [N,S] is a primary key of the relation (the other primary key being [R#,S]) whose proper subset i.e. the attribute N functionally determines two attributes viz. MN and FN.

The relation R may be converted into 2NF by splitting it into two relations namely, R.1(N,MN,FN) and R.2(R#,N,S,M). The functional dependency diagrams of the relations R.1 and R.2 are as shown in fig. 2.3.

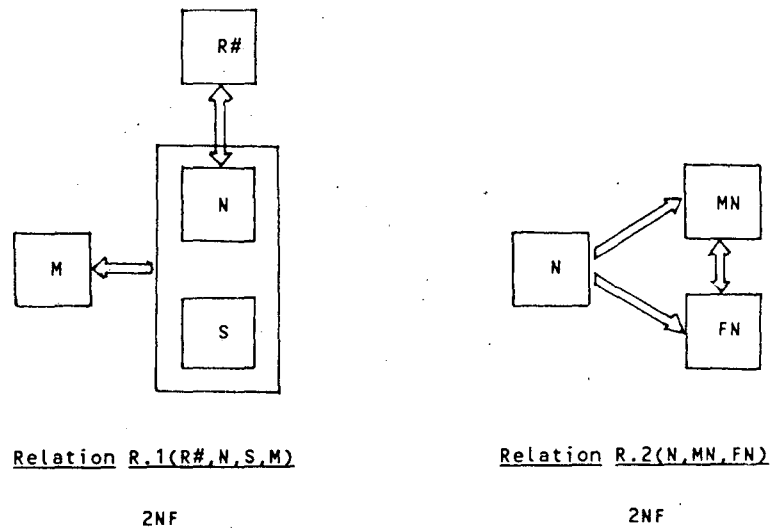


fig. 2.3

Functional Dependency diagrams for relations R.1 and R.2

The relations R.1 and R.2 both are in 2NF. The explanation goes as follows : Relation R.2 has N as its only primary key and since N doesn't have any proper subsets, the relation R.2 is in 2NF. Also the relation R.2 has two primary keys namely [N,S] and [R#,S]. M is the only non-key attribute and it is not functionally determined by any of the proper subsets of the primary keys. Hence R.2 also is in 2NF.

Conversion of R into R.1 and R.2 reduces the problems to the extent that now there is less redundancy as the names of the parents now are not to be repeated in every tuple concerning a particular child. Also we can add the information about the names of the parents of a child even if we do not know his roll number. Also the possible inconsistency concerning the names of the parents of a particular child is eliminated to some extent (it is still not fully eliminated as we will discuss in the next section).

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

Here we must introduce the concept of transitive dependencies. A dependency  $A \twoheadrightarrow B$  in R is said to be transitive iff A is neither a subset nor a superset of any primary key and B is a non-key attribute. The word 'transitive' comes from the fact that whenever such situation exists we must have the chain of dependencies  $K \twoheadrightarrow A \twoheadrightarrow B$  where K is any of the relation's primary keys.



A relation R is said to be in Third Normal Form (3NF) if and only if it is in 2NF and is free from any transitive dependencies.

We see that the relation R.1 in fig. 2.3 is in 3NF. The reason is that the only non-key attribute M depends on the primary keys only. Thus the conditions of 3NF are not violated and hence R.1 is in 3NF. But, at the same time, the other relation, R.2, is not in 3NF. The reason is that MN and FN functionally depend on each other while both are neither the subsets nor the supersets of the only primary key N. In other words, the following two transitive chains exist in R.2 that violate the conditions of 3NF,

N --> MN --> FN  
& N --> FN --> MN

To convert the relation R.2 in 3NF, it must be bifurcated into two relations viz. (N,MN) and (MN,FN) or (N,FN) and (FN,MN). The transitive dependencies are thus removed and the resulting two relations are in 3NF. This is shown in fig. 2.4.

The relation R.2 which was not in 3NF suffered from some problems. For example, the fact that Shivendra and Roopa are husband and wife was repeated every time a tuple concerning any of their children came and hence caused redundancy. This could lead to inconsistency problems too. But in the relations R.2.1 and R.2.2 (fig. 2.4), the first relates a child's name to his

mother's name and the second then in turn relates a mother's name to the father's name. Thus any scope of inconsistency is removed.

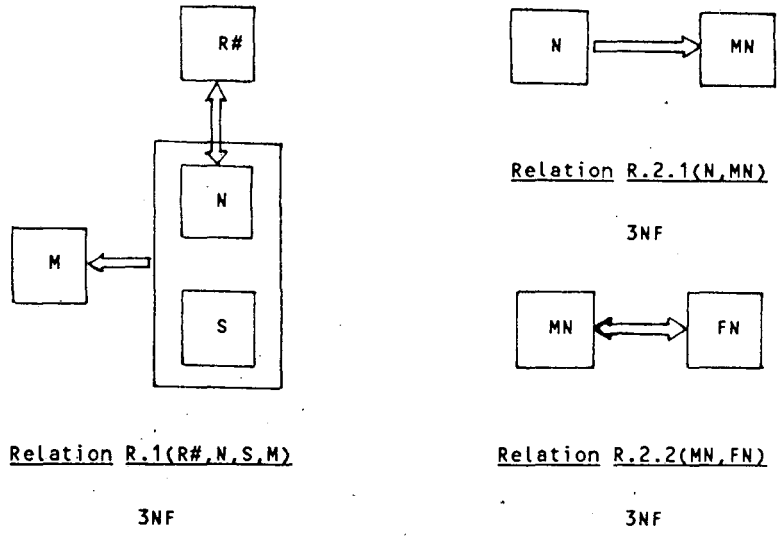


fig. 2.4

Functional Dependency diagrams for relations R.1 and R.2.1 & R.2.2

TH-3369

**2.4.5 Boyce-Codd Normal Form (BCNF)**

The original definition of the 3NF as given in the previous article suffers from certain problems in that it doesn't successfully eliminates inconsistency problems in all the situations. For example the relation R.1 in fig. 2.4 is in 3NF yet it suffers from certain problems as we shall discuss later in this article. Another normal form, named after its proposers

681.3.06 PROLOG *thesis*

G55  
no



Boyce and Codd is stronger than 3NF and eliminates all the problems arising due to functional dependencies.

A relation R is said to be in Boyce-Codd Normal Form (BCNF) iff every determinant in this relation is a key of that relation. A determinant is any attribute or a set of attributes that functionally determines any other attribute or set of attributes.

Conceptually, the definition of BCNF looks simpler than that of 3NF as it makes no explicit reference to the second normal form but in fact its definition is stronger than that of 3NF in that every BCNF relation will be in 3NF but the converse need not be true.

Let us come back to the "Child-Marks-Parents" data base. The relation R.2.1 and R.2.2 in fig. 2.4 are in BCNF as the only determinants in them viz. N and MN respectively are their primary keys. The relation R.1, however, though in 3NF is not in BCNF. This is so because N and R# are determinants (they functionally determine each other) but are not keys to the relation R.1. Thus for conversion of R.1 to BCNF it, too, is to be divided into two relations (R#,N) and (N,S,M) or (R#,N) and (R#,S,M). This is shown in fig. 2.5.

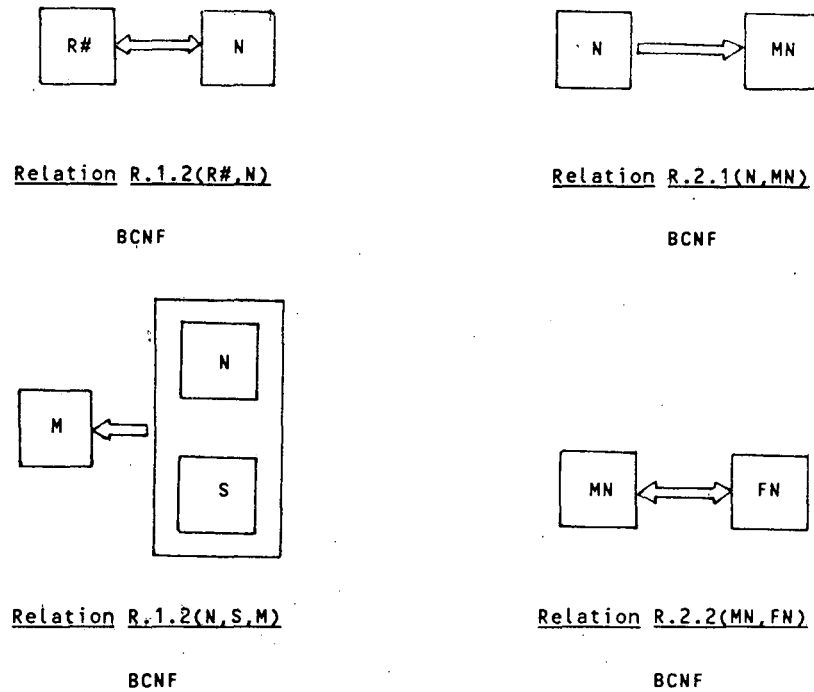


fig. 2.5

Functional Dependency diagrams

for relations R.1.1 & R.1.2 and R.2.1 & R.2.2

The relation R.1.1 and R.1.2 are in BCNF. R.1.1 is in BCNF because R# and N are the two determinants in this relation while both are also the relation's primary keys. R.1.2 is in BCNF because the primary key [N,S] is the only determinant in the relation. The relation R.1 could suffer from inconsistency problems as there was redundancy because of the repetition of the fact that a particular child had a particular roll number in every tuple concerning that student. The BCNF relations R.1.1 and R.1.2 remove this problem also.

Thus we saw how each step of normalization ridded the data base of some problems that existed earlier and how the BCNF stage removed all the problems that existed in the data base. In fact it has been proved that the BCNF is the ultimate normal form in the case the presence of only the functional dependencies in a data base. Further normalization is required in case multivalued dependencies are also present in some data base. This is the topic of discussion in the next chapter.

## CHAPTER - 3

### FURTHER NORMALIZATION

#### 3.1 INTRODUCTION

In the previous chapter we discussed about the need of normalization in a relational database having functional dependencies (FDs). It was shown that these dependencies cause a great deal of problems in maintaining the unnormalized records and to get rid of these difficulties the normalization process becomes inevitable. It was also mentioned that in such cases i.e. when a data base includes just the functional dependencies, the ultimate normal form is the BCNF as it eliminates all the inconsistency problems from the data base. A practical data base, however, does not contain just the functional dependencies but would, very often, include several of what we call multi-valued dependencies (MVDs) also. The multivalued dependencies cause inconsistency problems similar to those encountered in the case of functional dependencies. Moreover, when these dependencies are included in a data base, the BCNF is no longer the ultimate normal form. In other words, normalization up to BCNF doesn't solve the problems caused by the presence of multivalued dependencies. In such cases the normalization process has to go a step further and this is the topic of discussion of this chapter.

### 3.2 MULTIVALUED DEPENDENCIES (MVDs)

The functional dependencies defined in the previous chapter were concerned with only one-to-one relationships. For instance, in the "Child-Marks-Parents" data base discussed in that chapter, the attribute roll number (R#) functionally determined the child name attribute (N). This was a one-to-one relationship because each roll number is associated with exactly one student's name. Similarly, a child's name (N) functionally determined father's name (FN). This was so because every child has exactly one person as his father. But, in many situations we encounter relationships which are not one-to-one. For instance we might have a one-to-many or many-to-one or, for that matter, a many-to-many relationship also. Such relationships cannot be represented by just the functional dependencies. For the proper representation of such relationships we have to introduce the concept of multivalued facts and multivalued dependencies.

A 'multivalued fact' corresponds to a one-to-many relationship. For example, a father may have a number of children. So the relationship between father's name (FN) and child's name (N) is a multivalued fact about a father. But, at the same time, it is a single-valued fact about a child.

A relation may contain a number of multivalued facts. They may be about the same attribute or about different attributes. If in a relation, there are more than one

multivalued facts about the same attribute, they may either be independent of or dependent (non-independent) on one another. For example, consider the two relations shown in fig. 5.1. The first relation MSD is a record containing three fields viz. mother's name, son's name and daughter's name. Since a mother may have more than one sons as well as more than one daughters, this relation contains two multivalued facts about the same attribute i.e. mother's name. Similarly the other relation PTA too contains three fields viz. person, time and activity. Again we have two multivalued facts about the same attribute, here 'person'. But there is a basic difference between the two tables. While the multivalued facts in the table MSD are independent of each other, they are dependent in the table PTA. The independence of the multivalued facts in relation MSD arises from the fact that there is no direct connection between a son and a daughter except for the fact that both have the same mother. Thus all the boys who are sons of the same mother are brothers to all her daughters. Similarly all her daughters are sisters to all her sons. There is no special relationship between a particular daughter and a particular son. That is why, when we see that Indira has Girish, Titu & Pawan as her sons and Anupama & Preety as her daughters, all the six possible combinations of the sons and daughters are present in the table due to the fact that all the 'sons' are brothers of all the 'daughters' and all the 'daughters' are sisters of all the 'sons'.



Mother's name	Son's name	daughter's name
Indira	Girish	Anupama
Indira	Girish	Preety
Indira	Titu	Anupama
Indira	Titu	Preety
Indira	Pawan	Anupama
Indira	Pawan	Preety

Person's name	time	activity
Ashok	morning	games
Ashok	morning	meditate
Ashok	morning	study
Ashok	Evening	games
Ashok	Evening	study
Ashok	Night	sleep

Relation MSD (Mother, Son, Daughter)

Relation PTA (Person, Time, Activity)

independent multivalued facts

non-independent multivalued facts

fig. 5.1

In the relation PTA, on the other hand, the person : time and the person : activity relationships are not independent. It is due to the fact that a person may carry out only certain activities at a particular time while he may indulge in totally different activities at some other time. That is why we see in fig. 5.1 that while Ashok indulges in activities like games, meditation, study and sleep at different times viz. morning, evening and night, all the possible combinations of the different activities and different times are not present in the table. Thus he indulges in meditation only in the morning, sleeps only at night and plays games in the morning as well as the evening. Thus 'time' and 'activity', though multivalued facts about

'person' are not independent of each other.

It may be noted that it is the presence of all the possible combinations of values of multidependent facts among each other in the case of independent-multivalued-facts-relations that give these facts this property. Obviously, since every possible pairing of the values is present, there can be no information contained in these pairings, and hence the independence. In the case of non-independent facts, absence of a number of possible pairings makes the facts dependent on each other as was the case with the relation PTA (fig. 5.1).

The concept of multivalued dependencies is the same as that of independent multivalued facts. In fact, 'multivalued dependencies' is just the other name for 'independent multivalued facts'. The word independent is extremely important in this definition. Also since for independence we must have at least two multivalued facts about the same entity in a relation, multivalued dependencies also always go in pairs at least, and never in singles. The formal definition of multivalued dependency goes as follows :

In a relation R with attributes A,B and C, the multivalued dependence  $A \twoheadrightarrow B$  holds in R iff the set of B-values matching a given A-value & C-value pair, depends only on the A-value and is independent of the C-value. And in this case A is said to multidetermine B while B is said to be multidependent on A.

This formal definition, in fact, is not different from our earlier definition of the multivalued dependencies as being the same as independent multivalued facts. And with the same reasoning, it is easy to see that when in a relation  $R(A,B,C)$ , the MVD  $A \twoheadrightarrow B$  holds, another MVD  $A \twoheadrightarrow C$  must also hold. In fact, as we have mentioned earlier, MVDs always go at least in pairs in this way. For this reason, it is customary to express both the dependencies in a single statement, e.g.

$$A \twoheadrightarrow B \mid C$$

An attribute in a relation may multidetermine more than two attributes (attributes may, of course, be composite). For example in a relation say  $R(A,B,C,D)$  we may have

$$A \twoheadrightarrow B \mid C \mid D$$

### 3.3 THE FOURTH NORMAL FORM (4NF)

Multivalued dependencies give rise to the similar type of difficulties in the maintenance of a database as caused by the functional dependencies. These are inconsistency problems arising due to redundancy, inserting, deleting and updating etc. Let us discuss these problems one by one by taking a practical example of fig. 5.1.

**REDUNDANCY PROBLEM.** As is clear from fig. 5.1, in the table MSD, a lot of redundancy exists as there have to be all possible pair combinations of all the sons and daughters of a particular mother. This gives rise to two types of problems, first that by

mistake some inconsistency in the data may be caused and second that missing out even one of the possible combinations may lead to wrong interpretation of the data.

**INSERTING PROBLEM.** Suppose a lady employee of our firm is blessed with another child say a son, and we want to make entry for the new born in MSD, then since we have to scan the whole of existing table to find out all the existing entries concerning the lady and then suitably introducing the appropriate entries making all the required combinations of the boy with all his sisters, this not only makes a complicated procedure but also opens up possibilities of creating unwanted inconsistencies.

**DELETING PROBLEM.** Suppose in the relation MSD, we want to keep the records of only those children who are below 21. And now a child of a particular employee turns 21. To delete his entry from the table now again requires the complicated and risky procedure of scanning the whole table for all the entries concerned with the boy. Leaving out even a single entry will lead to problems. There is also possibility of losing the information about his sisters altogether if he was the only son of his parents because removing all the entries having information about him will automatically wipe out the information about his sisters also. Also, in case, he was the only child of his parents, removing all his entries will amount to removing the name of his mother altogether from the record MSD.

**UPDATING PROBLEMS.** Updating causes the similar difficulties

arising due to the fact that this would also require the proper updating in all the entries connected with the particular entity. Since a simple updating in just one position doesn't suffice it leads to possible risks of causing inconsistency in case we leave out some entry uncorrected.

Thus we have shown how the presence of multivalued dependencies leads to problems in a data base. It must also be mentioned that only the multivalued dependencies i.e. independent multivalued facts lead to such problems. Presence of multivalued facts in a relation that are not independent doesn't lead to such inconsistency problems. For example in the relation PTA in fig. 5.1, the multivalued facts person : time and person : activity are dependent on each other, so that all the data entries in the table PTA are required to keep the whole of information. This is not redundancy because we cannot reduce the number of entries (by normalization process etc.) without losing some the information contained therein.

It was shown in the previous chapter that in the case of FDs only, the ultimate normal form is the BCNF. But since in the table MSD (fig. 5.1) there are no functional dependencies (FDs), there is no basis by which we can convert MSD into smaller tables. In fact, since there is no functional dependency in MSD, it is an all-key relation i.e. the set of all attributes is the primary key. And as such it is in BCNF. Thus we see that the presence of the MVDs has caused the same problems in even a BCNF

relation. Thus to tackle the MVDs we need a normal form which is stronger than the BCNF. This is the 'Fourth Normal Form (4NF)' that we are going to discuss now.

Under the Fourth Normal Form (4NF), a relation should not contain two or more independent multivalued facts about the same entity. In other words, the 4NF does not allow the presence of more than one multi-determined facts about the same entity. In addition to that, the relation must be in BCNF. When these two conditions are satisfied, the relation is said to be in 4NF. Thus the relation MSD in fig. 5.1 is not in 4NF because it contains two multi-determined facts about the same attribute M. The relation PTA is in 4NF however, because it does not contain multivalued dependencies at all (it contains non-independent multivalued facts). The relation MSD must be divided into two relations MS and MD to convert in into 4NF. This is shown in fig. 5.2.

MS	Mother's name	Son's name
	Indira	Girish
	Indira	Titu
	Indira	Pawan

MD	Mother's name	daughter's name
	Indira	Anupama
	Indira	Preety

Relation MS (Mother, Son)

Relation MD (Mother, Daughter)

4NF

4NF

fig. 5.2

Thus we see that the relation MSD is broken into two relations each including one of the multidetermined facts. The first relation contains information about the sons while the second relation contains information about the daughters. The relations MS and MD are in 4NF as they do not contain any multivalued dependencies (for a multivalued dependency, a relation, must contain at least two multivalued facts). We can see how normalization up to 4NF solved the difficulties that existed with the relation MSD. As is clear from the fig. 5.2, since there are two separate tables for sons and daughters of an employee there is not any unnecessary redundancy arising due to such requirements like mandatory keeping of all possible pair combinations. Moreover inserting, deleting and updating problems are solved because now these changes are to be made at only one place instead of searching the whole of table for updating all the concerned entries as was the case in the table MSD.

The formal definition of the fourth normal form (4NF), however, does not require the relation to be in BCNF. It goes as follows :

A relation R is in Fourth Normal Form (4NF) if and only if, whenever there exists an MVD in R, say  $A \twoheadrightarrow B$ , then all attributes of R are also functionally dependent on A (i.e.,  $A \rightarrow X$  for all attributes X of R).

The above definition and the earlier one are equivalent and in simple terms they mean that the problems arising due to

multivalued dependencies will be removed if we don't allow more than one independent multivalued facts about the same entity to remain in the same relation.

In the next chapter we discuss the actual algorithms for conversion of unnormalized relations to higher normal forms up to 4NF.



## CHAPTER - 4

### NORMALIZATION ALGORITHMS

#### 4.1 INTRODUCTION

Since its emergence, some twenty years back, Data base technology has come a long way. As an inseparable part of it, Normalization theory has also developed but not to the extent as it should have. Two basic approaches that have developed in the field of normalization theory are the synthesis approach and the decomposition approach. The decomposition approach was the first to come up but suffered from certain limitations in certain situations and this prompted the synthesis approach to come up some time later. Both the approaches have their 'plus' and 'minus' points and both are inevitable for the Normalization theory. We touch upon these approaches, to make out the difference between the two, and present certain specific algorithms involving both the approaches.

#### 4.2 SYNTHESIS AND DECOMPOSITION APPROACHES

The two major approaches to have come up in the logical schema design or the 'normalization theory' in the relational data bases are the synthesis and decomposition approaches. The difference lies in the directions that the two approaches follow to reach the same goal. What the decomposition approach does is

that it takes the relation in the unnormalized form and then step by step decomposes it into smaller relations by removing anomalies in it. On the other hand, the synthesis approach follows just the opposite way. In this approach, the set of FDs is chosen as the basis and the final relations are constructed from them. The core of the problem lies in determining the proper set of FDs that should be used for that purpose. Some of the salient plus and minus points of the two approaches are listed as follows :

- \* 1. The decomposition process often yields more relations than are actually needed.
- \* 2. The decomposition process may produce a design that does not enforce some of the FDs. A synthesis approach would never allow such a design.
- \* 3. Synthesis approach works well for FDs, but is not suitable for processing MVDs. Decomposition approach, however, is straightforwardly extendible to MVDs.
- \* 4. The highest normal form that the synthesis approach can achieve is 3NF. The decomposition approach, on the other hand, is not limited in this way. Thus for obtaining normal forms higher than 3NF, the only approach that can be followed is the decomposition approach.

In the next two articles we give algorithms for conversion of unnormalized relations into the third normal form (3NF), the Boyce-Codd normal form (BCNF) and the fourth normal

form (4NF). The procedure to achieve the 3NF is the 'Bernstein's algorithm' based on the synthesis approach. The procedures for the BCNF and the 4NF are respectively the 'Tsou & Fischer's algorithm' and the 'Tanaka's algorithm' both based on the decomposition approach.

#### **4.3 BERNSTEIN'S ALGORITHM FOR NORMALIZATION TO 3NF**

Bernstein's algorithm for normalization up to the third normal form uses the synthesis approach. A proper set of FDs is chosen and then 3NF relations are built from them. The actual algorithm is as follows :

**Input :** An unnormalized relation and a set  
of FDs (F)

**Output :** 3NF relations

**Step 1. Eliminate Extraneous attributes.** Eliminate the extraneous attributes from the left side of each FD in the set F, producing the set G. An attribute is extraneous if its elimination does not alter the closure of the set of FDs. By the closure of a set of FDs we mean the set of all the FDs that can be derived from that set. An equivalent check for an attribute to be extraneous is that an attribute A is extraneous in the FD : LHS  $\rightarrow$  RHS if it can be eliminated from the LHS so that the new dependency (LHS - {A})  $\rightarrow$  B holds.

**Step 2. Find a non-redundant cover.** Find a non-redundant cover H of G by eliminating redundant FDs from G. An FD is redundant in G if its elimination does not alter the closure of

the FDs present in G.

**Step 3. Partition into groups.** Partition the set of dependencies H into groups  $H_i$  such that all dependencies in each group have identical left sides.

**Step 4. Merge Equivalent keys.** Merge two groups  $H_i$  and  $H_j$  with left sides X and Y respectively if the keys X and Y are equivalent. Two keys X and Y are said to be equivalent when the dependencies  $X \twoheadrightarrow Y$  and  $Y \twoheadrightarrow X$  both hold. For merging the following process is to be adopted :

Set  $J := \emptyset$ . For each pair of groups  $H_i, H_j$  with left sides  $X_i$  and  $X_j$  respectively do the following : if  $X_i$  and  $X_j$  are equivalent, merge  $H_i$  and  $H_j$ , add the FDs  $X_i \twoheadrightarrow X_j$  and  $X_j \twoheadrightarrow X_i$  to J, and remove them from H.

**Step 5. Eliminate Transitive dependencies.** Find a minimal cover  $H' \subseteq H$  such that  $(H' + J)^+ = (H + J)^+$  where the superscript '+' denotes the closure of the set of FDs. Delete each FD in  $(H - H')$ , from the group in which it appears. Also for each FD in J, add it to the corresponding group. Thus we have obtained such a partitioning of groups which include all the equivalent keys and in which all other dependencies are non-redundant. Such groups are free from transitive dependencies.

**Step 6. Construct relations.** For each group, construct a relation consisting of all the attributes appearing in that group. The LHS common to all the FDs in that group will be a key of the constructed relation. The set of all relations so

constructed will constitute the required schema of 3NF relations.

Let us apply the Bernstein's algorithm to an example. Take the relation R of the "Child-Marks-Parents" data base discussed in the chapter 2 (fig. 2.1). The schema is :

```

Schema      R = (R#,N,S,M,MN, FN)
-----
F : R# --> N
    N --> R#
    N --> FN
    N --> MN
    MN --> FN
    FN --> MN
    [N,S] --> M
  
```

Various steps of the Bernstein's algorithm as applied to the above example will be as follows :

**Step 1 :** The FDs of the set F do not contain any extraneous attribute.

**Step 2 :** The FD :  $N \twoheadrightarrow MN$  is found to be redundant and hence removed.

**Step 3 :** Group formed are with LHSs :

$[R\#], [N], [N,S], [FN], [MN]$

**Step 4 :** Equivalent keys are

$N \leftrightarrow R\#$       and       $FN \leftrightarrow MN$

**Step 5 :** No transitive dependencies exist. So finally the merged groups are with LHSs :

$[R\#,N], [N,S], [FN,MN]$

**Step 6 :** Relations are formed from the definitive groups. The final relations are :

R_a = (R#,N, FN)	in	3NF
R_b = (N,S,M)	in	3NF
R_c = (FN,MN)	in	3NF

Same result is obtained using the Prolog Program. This is included in the examples of Appendix - II.

#### 4.4 DECOMPOSITION ALGORITHMS FOR NORMALIZATION TO BCNF and 4NF

As we have mentioned earlier, the Synthesis approach can attain normalization up to only 3NF. For obtaining normal forms higher than that and for handling MVDs we have to resort to the Decomposition approach. But the decomposition approach contains a lot of inherent problems like creating more tables than are needed and producing a design that does not enforce some of the initial FDs. A decomposition algorithm for creating BCNF relations directly from an unnormalized relation (1NF) was put forward by Tsou and Fischer. This algorithm takes care of the lossless join property of the decomposition and is presented, in a simple language as follows :

**Input :** 1NF relation with a set of FDs

**Output :** A set of BCNF relations

**Step 1.** Let S be the set of all attributes in the schema of the given relation R. Let us introduce an active set AS := S.

**Step 2.** Find a subset B of the set AS which has the following property : The set B doesn't contain any element which can be generated without the help of some other element of B. Also at

least one element, E of B must be capable of being generated by the rest of the elements of B. Construct a relation for the set B. We say that an element can be generated by a set of attributes if it belongs to the closure of that set of attributes. Modify AS as follows :

AS := AS - {E}

**Step 3.** Repeat the step 2 until AS reduces to just two elements. Construct a relation for this AS also.

**Step 4.** Output the relations constructed. They are in BCNF.

In the step 2 of the above algorithm, we have to find the set B from the set AS. The procedure to be followed for that is :

- a. Take B := AS.
- b. Check for an element A ∈ B such that

$A \in \text{clo}(B - \{A, C\})$  where  $C \in B$  and  $C \neq A$

[ 'clo' stands for the closure of the attributes of the set ]

If A exists then B := B - {C}  
 ----- repeat the step b.  
 else Output B.  
 -----

The Tsou and Fischer's algorithm is implemented in the Prolog Program and the actual implementation is discussed in the next chapter. This algorithm when directly applied to the relation R (fig. 2.1), yields the following set of BCNF relations :

R\_1 = (R#,N)            in BCNF  
 R\_2 = (N,S,M)           in BCNF                    contd..

R_3 = (FN, MN)	in	BCNF
R_4 = (N, FN)	in	BCNF
R_5 = (N, S)	in	BCNF

We see that R\_5 is an unwanted relation that is generated by the decomposition algorithm of Tsou & Fischer. Thus we have illustrated one of the disadvantages of decomposition algorithms viz. creation of more relations than are needed.

**Case of Multivalued Dependencies.** Normalization problem becomes more complicated when the data base includes Multivalued Dependencies as well as the Functional dependencies. The biggest problem that arises in using the decomposition approach in such cases is the **priority problem** i.e. whether to give priority to FDs over MVDs or vice-versa. It has been seen that none can be given priority over the other in all cases. For example, in certain cases, giving priority to FDs over MVDs causes redundancy in the result and in certain others, the reverse is observed. In fact, in these cases where FDs and MVDs are both present, the following very important result holds :

Redundancy in the decomposed result may occur if the decomposition by an MVD h precedes that by another MVD g such that the MVD-determinant of g is functionally dependent on that of h. It is to be remembered here that an FD is a special case of an MVD.

A comprehensive algorithm for decomposition up to the fourth normal form 4NF, was suggested by Y. TANAKA. His



decomposition algorithm produces 4NF relations without redundancy (though he himself mentions that redundancy is not removed in a strict sense by his algorithm, quoting an example to illustrate the same; see ref).

Before presenting the Tanaka's algorithm for normalization to 4NF, we must get familiar with some of the terminology used in the same and with some axioms connected with the FDs and the MVDs.

Z	will denote	context of a relation i.e. the set of all attributes in the relation
F	will denote	the given set of all FDs
M	will denote	the given set of all MVDs
$\Gamma$	will denote	the set of all given dependencies i.e. $\Gamma$ is union of F and M
$\Gamma^+$	will denote	the closure of $\Gamma$
$\Gamma:A$	will denote	the closure of $\Gamma$ with respect to the set of dependencies A
FD( $\Gamma:A$ )	will denote	the FD part of $\Gamma:A$
MD( $\Gamma:A$ )	will denote	the MVD part of $\Gamma:A$

For the design theory of the 4NF schema, we need a complete set of axioms for FDs as well as MVDs to act as inference rules to calculate all dependencies. The following is the list of axioms :

<b><u>FD1 (Reflexivity)</u></b>	If $Y \subseteq X$ then $X \twoheadrightarrow Y$
<b><u>FD2 (Augmentation)</u></b>	If $Z \subseteq W$ and $X \twoheadrightarrow Y$ then $XW \twoheadrightarrow YZ$
<b><u>FD3 (Transitivity)</u></b>	If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$ then $X \twoheadrightarrow Z$

<u>MVD0 (Complementation)</u>	If $X \twoheadrightarrow Y$ in $Z$ then $X \twoheadrightarrow Z - Y$ in $Z$
<u>MVD1 (Reflexivity)</u>	If $Y \subseteq X \subseteq Z$ then $X \twoheadrightarrow Y$ in $Z$
<u>MVD2 (Augmentation)</u>	If $V \subseteq W \subseteq Z$ and $X \twoheadrightarrow Y$ in $Z$ then $XW \twoheadrightarrow YV$ in $Z$
<u>MVD3 (Transitivity)</u>	If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow W$ in $Z$ then $X \twoheadrightarrow W - Y$ in $Z$
<u>MVD4 (Embedding)</u>	If $X \subseteq W \subseteq Z$ and $X \twoheadrightarrow Y$ in $Z$ then $X \twoheadrightarrow Y \cap W$ in $W$
<u>MVD5 (Extension)</u>	If $X \twoheadrightarrow Y$ in $Z$ & $(Z-Y) \twoheadrightarrow Y$ in $W$ , where $W \supseteq Z$ then $X \twoheadrightarrow Y$ in $W$
<u>MVD6 (Reconnection)</u>	If $X \twoheadrightarrow Y$ in $Z$ , $V \twoheadrightarrow Y$ in $W$ and $(Z \cap W) \twoheadrightarrow X$ in $XV(Z \cap W)$ then $XV \twoheadrightarrow Y$ in $XV(Z \cap W)$
<u>FD-MVD1</u>	If $X \twoheadrightarrow Y$ and $X, Y \subseteq Z$ then $X \twoheadrightarrow Y$ in $Z$
<u>FD-MVD2</u>	If $X \twoheadrightarrow Y$ in $Z$ and $(Z-Y) \twoheadrightarrow Y$ then $X \twoheadrightarrow Y$

The set of axioms FD1-3, MVD0-3, FD-MVD1-2 is known as the complete set of axioms for the FDs and MVDs. In the design process, it is required to know the closure of FDs and MVDs i.e. all the dependencies inferable from sets  $F$  and  $M$ . For the convenience of computation, the axioms MVD3-5 and FD-MVD2 are replaced respectively by MVD7 and FD-MVD3, which are introduced as under :

<u>MVD7 (MVD interaction)</u>	If $X \twoheadrightarrow Y$ in $Z$ , $U \twoheadrightarrow V$ in $W$ where $X \subseteq W$ and $U \subseteq Z$ then $X(Y \cap U) \twoheadrightarrow Y \cap V$ in $Z-(Y-W)$ and $U(V \cap X) \twoheadrightarrow Y \cap V$ in $Z-(Y-W)$
<u>FD-MVD3 (FD-MVD interaction)</u>	If $X \twoheadrightarrow Y$ , $U \twoheadrightarrow V$ in $W$ and $X \subseteq W$ then $U(V \cap X) \twoheadrightarrow Y \cap V$

Also there are two very important results regarding these dependency rules. They are as follows :

**Lemma 1**

A set A of axioms FD1-3, MVD0-6 and FD-MVD1-2 is equivalent to a set B of axioms FD1-3, MVD0-2, MVD6-7 and FD-MVD1,3.

**Lemma 2**

Let D be the set of dependencies {FD1-3, MVD0, FD-MVD3}, i.e., D has no rules about MVD interaction. Then the following relation holds true :

$$FD(\tau:D) = FD(\tau:C)$$

Both of these results are by Y.Tanaka.

For the convenience of computation of dependency closure, it is useful to introduce a standard combined representation for an FD or an MVD or both. The standard representation of a dependency f with a context (i.e. the set of all attributes) Z has a form :

$$X : [Y_0] Y_1 | Y_2 | Y_3 | \dots | Y_n ,$$

where  $\{X, Y_0, Y_1, Y_2, \dots, Y_n\}$  is a partition of Z,

$X \twoheadrightarrow Y_0$  and  $X \twoheadrightarrow Y_i$  in Z for any i .

Also three functions are defined for this standard representation :

$$\text{context}(f) = Z,$$

$$\text{left}(f) = X,$$

$$\text{right}(i, f) = Y_i \text{ if } i \leq n \text{ else } \emptyset.$$

Now, the axiom MVD6 is reasonably neglectable in most of the practical applications. We therefore adopt a decomposition algorithm for constructing 4NF relations neglecting MVD6. This algorithm is due to Tanaka. The algorithm is as follows :

**Input** : Un normalized relation R with context Z,  
 set of functional dependencies F and  
 set of multivalued dependencies M

**Output** : Set of 4NF relations

**Step 1.** Let C be the dependency set equal to B - {MVD6} i.e. the set { FD1-3, MVD0-2,7 , FD-MVD1,3 }. We want to calculate  $\Gamma:C$ . For this, let F' and M'' be the FD and MVD parts of  $\Gamma:C$ . To get F', calculate  $FD(\Gamma:D)$  as by Lemma 2,  $F' = FD(\Gamma:C) = FD(\Gamma:D)$ , where  $D = \{FD1-3, MVD0, FD-MVD3\}$ .

**Step 2.** Obtain an intermediate set M' as follows :

For each f in M, if there exists a functional dependency (in F') from a subset X of left(f) to all the attributes in left(f), replace left(f) by X and move left(f)-X from left(f) to right(0,f). Move also all attributes in right(i,f), for all  $i > 0$ , that are functionally dependent on left(f) to right(0,f). Then, for each functional dependency  $f : X \twoheadrightarrow Y$  satisfying that Y is a maximum set dependent on X and X is a minimal set that determines (X U Y), we add an MVD g to M' that is defined as follows :

context(g) = Z,  
 left(g) = X,  
 right(0,g) = Y,  
 right(1,g) = Z - X - Y,

$\text{right}(i,g) = \emptyset$  for  $i > 1$ .

**Step 3.** Calculate  $M'' = \text{closure of } M' \text{ w.r.t. } E$  i.e.  $M':E$   
where  $E = \{\text{MVD0-2, MVD7}\}$ .

**Step 4.** Get a dependency  $g$  in  $M''$  such that  $\text{right}(0,g)$  is minimal i.e.  $\text{right}(0,f) \subseteq \text{right}(0,g)$  for no  $f \in M''$  and also  $Z \subseteq \text{context}(g)$  and  $\text{left}(g) \subseteq Z$ , where  $Z$  is the context of the relation  $R$  to be decomposed. Then form relations  $R_i$  with contexts :

$$\text{context}(R_i) = Z \cap (\text{left}(g) \cup \text{right}(i,g))$$

for all  $i$ .

The relation  $R_0$  so obtained is in 4NF and also  $(Z \cap \text{left}(g))$  is its key. Rest of the relations decomposed need not be in 4NF and hence apply **step 4** to each of these decomposed relations except  $R_0$ . If, while applying step 4 to any relation, we cannot find the dependency  $g$  satisfying all the required conditions, this means that the relation concerned is in 4NF.

Go on doing step 4 unless all the decomposed relations are obtained in 4NF.

The actual implementation of Tanaka's algorithm in Prolog is discussed in the next chapter. Also the algorithm is applied to a few examples which are given in appendix - II.

## CHAPTER - 5

### THE PROLOG PROGRAM FOR NORMALIZATION UP TO 4NF

#### 5.1 INTRODUCTION

This chapter discusses the actual Prolog program that can automatically normalize a given relation up to the Fourth Normal Form (4NF). The input required to the program is the list of the attributes i.e. the context of the relation and the set of all the dependencies associated with it. We can get the relation normalized up to the 3NF or the BCNF depending on our wish. The relations containing MVDs too, are to be normalized up to 4NF. The software used for the program is Turbo Prolog 2.0.

#### 5.2 THE PROGRAM

We now take up the actual program code. First of all, the various declarations are discussed. After that the rules are touched upon, with elaborate explanation wherever needed.

##### 5.2.1 Declarations

Various declarations in a Prolog program, before the clauses begin, are the domains, database and the predicates declarations. The domains defined in this program are :

**sym, list, listoflists and int**

The 'sym' and 'int' domains are just the other names for the standard symbol and integer domains. 'list' is defined as :  
list = sym\* i.e. 'list' has been defined as list of symbols.  
'listoflists' is defined as : listoflists = list\* i.e.  
'listoflists' is list of lists of symbols.

After the domains section, databases are defined in the database section. A database declaration contains the name followed by the specification of the domains of its arguments. For example the functional dependencies will be stored in the database `fd(sym,list,list)`. This declaration means that we can store facts named 'fd' in this database and each fact will contain three arguments, the first a symbol followed by two other lists of symbols. For example, suppose we want to assert the fact that the attribute B is functionally dependent on the set of attributes {A,C} in a relation R, we will represent this fact in our program as follows :

`fd(r,[a,c],[b])`

Note that we had to change the attribute names etc. to lower case as in Prolog any upper case letter is taken to be a variable.

As another example, there is a database defined as `schema(sym,list)`. This means that if we wish to give the system, information about a relation R containing the set of attributes {R#,N,S,M,MN, FN} we will either put the fact directly by writing the following fact-statement in the clauses section :

`schema(r,[r#,n,s,m,mn,fn]).`

or by using assert statement in the right-hand-side of any

rule like :

```
goal1 :- assert(schema(r,[r#,n,s,m,mn,fn])).
```

Note that every statement in the clauses section must terminate in a period (.).

The database section is followed by the predicates section. This section is used to declare each predicate that will be used in the program to describe various facts. Thus the program knows in advance the structure of each predicate. The declarations in this section are similar to those in the database section i.e. the predicate name followed by the domains-specification of its arguments.

The final section of a Prolog program is the clauses section. The actual code of the program is contained in this section only and we shall discuss the various clauses in the rest of this chapter.

The two statements in the very beginning of the program viz. nowarnings and code=2000 are compiler directives. The statement 'nowarnings' tells the compiler not to give warnings like variable used only once and the statement 'code=2000' specifies the internal code array in terms of the number of paragraphs. The default code-array size of 1000 paragraphs i.e. 16 Kilo Bytes was found to be insufficient.

One very important feature of this program must be mentioned here viz. all the variables of the type 'list' are treated as if they are not simply lists but are 'ordered sets'.



This means that we will never allow any attribute to appear more than one in any list and also the ordering of the attributes will not be immaterial, the original ordering of the attributes of any relation being defined by the schema declaration of that relation or the schema declaration of some relation from which it has been derived.

### 5.2.2 Utility clauses

There are various clauses in our program that are used so often that we shall call them utility clauses. The list of the various utility clauses is as follows :

`equal(L1,L2)` succeeds when list L1 equals list L2. If either of L1 and L2 is unbound, it binds it equal to the other.

`equal2(LL1,LL2)` succeeds when listoflists L1 equals listoflists L2. If any of them is unbound, it binds it equal to the other.

`elem(E,L)` succeeds when E is an element of the ordered set L.

`listelem(L,LL)` succeeds when list L is an element of the listoflists LL.

`subset(A,B)` succeeds when list A is an ordered subset of list B.

`attr(A,REL)` succeeds when list A is an ordered subset of the attributes of the schema of REL. Domain-type of REL is 'sym'.

`union(U,A,B,REL)` succeeds by building the union U of two ordered sets A and B of attributes of the relation REL. It is important that U, A and B all are ordered in consistence with the ordering

in the set of attributes of the schema of the relation REL.

`minus1(D,A,N)` succeeds by building the difference D between the set A and a single element N. This means that this clause succeeds by achieving  $D = A - \{N\}$ .

`listminus1(LD,LA,LN)` succeeds by building the difference LD between the listoflists LA and a single list LN. This means that this clause succeeds by achieving  $LD = LA - \{LN\}$ .

`minus(D,A,B)` succeeds by building the difference D between the ordered sets A and B. Thus it succeeds after achieving  $D = A - B$ .

`append(L1,L2,A)` succeeds by appending list L2 to list L1 and then storing it in list A if A is not already bound. The result A will not be an ordered set as it will be a simple concatenation of L1 and L2 and thus may have repeated attributes also.

`append2(L1,L2,A)` succeeds by appending listoflists L2 to listoflists L1 and then storing it in listoflists A if A is not already bound.

Let us explain one of these clauses, say the union clause' as it is lengthiest of the lot. The rule goes as follows :

```
union(U,A,B,REL) :- schema(REL,S), subset(A,S), subset(B,S),
                    union1(U,A,B,S).
```

```
union1(A,A,[],_) :- !.
union1(B,[],B,_) :- !.
union1([H|TU],[H|TA],[H|TB],[H|TS]) :- !, union1(TU,TA,TB,TS).
union1([H|TU],[H|TA],B,[H|TS]) :- !, union1(TU,TA,B,TS).
union1([H|TU],A,[H|TB],[H|TS]) :- !, union1(TU,A,TB,TS).
union1(U,A,B,[_|TS]) :- union1(U,A,B,TS).
```

The first thing the union clause does is that it calls for the list of attributes S from the schema data base. Then it checks whether the lists A and B are ordered subsets of S. If it is so, it calls the rule union1(U,A,B,S). There are six clauses for the predicate union1. Prolog tries them one-by-one unless one is satisfied. The first clause says that if B is a null set then the union will be equal to A. The second does the same for the case when A is a null set. The third clause says that if the first element of A,B and S is same then the union of A and B also will have the same element as its first element and to find the tail of the union, the same rule union1 is to be applied to the tails of A, B and S. The fourth and the fifth clauses say that if the first element of S matches with any of the lists A and B, then the first element of U will also be the same element and to find the tail of U we will have to apply again the rule union1 to the tail of the list whose first element matched, the full of the other list and the tail of S. The sixth and the last rule says that if the first element of S does not match with that of any of A and B, then U is to be found by applying the rule union1 to A, B and the tail of S.

The symbol '!' used at many places in the clauses denotes cut. The 'cut' is a very special facility of Prolog and is used to prevent backtracking to go beyond a particular point. In the clauses for 'union1' the cuts are used to avoid multiple answers i.e. not to allow the program to look for another answer once it has found the value of U.

### 5.2.3 Closure of a set of attributes

The closure of a set of attributes  $X$  with respect to a set of functional dependencies is the set of all attributes  $A$  such that  $X \twoheadrightarrow A$  can be deduced by the FD axioms. This set is obtained in our Prolog program by a recursive rule. At each level, an FD is searched such that the left hand side LHS of it is a subset of  $X$  and the right hand side RHS is not a subset of  $X$ . When such a dependency is found, the RHS contributes its new attributes to the closure and the algorithm is recursively called on the new set built as the union of  $X$  and RHS. The clauses are :

```
closure(REL,X,RESULT) :- fd(REL,LHS,RHS), subset(LHS,X),
                          not(subset(RHS,X)),
                          union(U,X,RHS,REL), !,
                          closure(REL,U,RESULT).
```

```
closure(REL,X,RESULT) :- RESULT = X.
```

It is the first rule that goes in the recursive process. The second rule becomes active only when the modified set  $X$  has become so big that no FD satisfies the conditions of the first rule. The second rule then assigns the value of  $X$  to RESULT. Hence in RESULT the closure of the initial set  $X$  comes.

### 5.2.4 Elimination of Extraneous Attributes

An attribute is extraneous in an FD if it can be eliminated from the LHS so that the new dependency  $LHS - \{A\} \twoheadrightarrow RHS$  holds. The algorithm reducelhs does this job of eliminating extraneous attributes from the LHS of an FD. It

looks for an attribute A of the LHS, builds the difference Z between LHS and A, evaluates the closure of Z and tests whether RHS is a subset of this closure. If so, the attribute A is extraneous and the reducelhs rule is recursively evaluated on Z. When no further reduction is possible the second rule of reducelhs sets the value of the NEWLHS equal to Z. The rules are as follows :

```

reducelhs(REL,LHS,RHS,NEWLHS) :- elem(A,LHS), minus1(Z,LHS,A),
                                not (equal(Z,[])),
                                closure(REL,Z,ZCLO),
                                subset(RHS,ZCLO), !,
                                reducelhs(REL,Z,RHS,NEWLHS).

reducelhs(REL,LHS,RHS,NEWLHS) :- NEWLHS = LHS.

```

Another rule elimattr does the job of calling each FD one by one, applying reducelhs on it, checking whether the LHS of the FD has been changed after applying reducelhs and if so retracting the earlier FD from the database and asserting the new FD i.e. the FD with NEWLHS. The rule is as follows :

```

elimattr(REL) :- fd(REL,LHS,RHS),
                 reducelhs(REL,LHS,RHS,NEWLHS),
                 not (equal(LHS,NEWLHS)),
                 retract(fd(REL,LHS,RHS)),
                 asserta(fd(REL,NEWLHS,RHS)), fail.

elimattr(REL).

```

Note that while asserting the new FD, the predicate used is asserta instead of simply assert or assertz. This ensures that the newly asserted FD goes to the beginning of database so that it is not called again when the rule follows forced recursion due to the predicate fail.

### 5.2.5 Elimination of Redundant FDs

The rule elimredundfds does the job of eliminating all the redundant FDs i.e. the FDs whose elimination from the set of FDs 'F' does not alter the closure of F. The first rule calls each FD one by one and retracts it from the data base temporarily. Now it calculates the closure of it's LHS with respect to the new set of FDs. If it's RHS belongs to the closure of it's LHS with respect to this new set of FDs, this means that this particular FD is redundant. In this case, it does not put the FD back to the data base and calls for another FD. But if RHS does not belong to the closure of LHS, this means that this particular FD's removal has changed the FD-closure, and hence the rule first puts back the FD to the data base and then calls for another FD. When all the FDs are checked, the second rule makes it true and the process ends.

```
elimredundfds(REL) :- fd(REL,LHS,RHS),
                    retract(fd(REL,LHS,RHS)),
                    closure(REL,LHS,Z),
                    choice(REL,LHS,RHS,Z),
                    fail.
elimredundfds(REL).

choice(REL,LHS,RHS,Z) :- not (subset(RHS,Z)),
                        asserta(fd(REL,LHS,RHS)).

choice(REL,LHS,RHS,Z) :- subset(RHS,Z).
```

Note that the rule elimredundfds calls for another rule choice because the Turbo Prolog doesn't allow the use of the 'or' operator ';' in the clauses. The first clause of choice becomes operative when RHS is not a subset of Z and causes the FD to be

asserted back to the database. While the second clause of choice becomes operative only if RHS is found to be a subset of Z and this clause does nothing, just goes true.

#### 5.2.6 Bernstein's Algorithm

As we have discussed in the previous chapter, the Bernstein's Algorithm for converting an unnormalized relation to a set of 3NF relations, consists of six steps. In line with that, our Prolog implementation of the Bernstein's Algorithm also consists of six steps. They are as follows :

Step 1. The first steps consists of eliminating the extraneous attributes from the FDs in the functional dependencies set F. This simply means that all we have to do in the step1 is to call the predicate elimattr. Thus the step1 will be :

```
step1(REL) :- elimattr(REL).
```

Step 2. This step consists of finding a non-redundant cover of the FDs. This job will be done by the rule elimredundfds. Thus the step2 will be :

```
step2(REL) :- elimredundfds(REL).
```

step 3. This step consists of partitioning of the set of dependencies into groups with identical left hand sides. Two data bases viz. group(sym,listoflists) and clo(sym,list,list) are used here. The fact group stores the list of LHSs of a particular group formed for a relation REL and the fact clo stores the

closure of a particular list of attributes. The closure of each LHS is stored as this is going to be used in step4. The clauses for step3 are as follows :

```

step3(REL) :- fd(REL,LHS,_), not (group(REL,[LHS])),
              asserta(group(REL,[LHS])),
              closure(REL,LHS,CLO), asserta(clo(REL,LHS,CLO)),
              fail.

step3(REL).

```

The first rule of step3 looks at each FD and if no group already exists with same LHS it asserts a group for that. Also it finds the closure of this LHS and stores in the data base clo. When all the FDs are exhausted the second rule makes it true.

**Step 4.** This step consists of merging the groups with equivalent keys. Two keys X and Y are equivalent if each is functionally dependent on the other. As we have already explained, the merging is done by first assigning a dependencies set  $J := \emptyset$ . Then we look for equivalent keys. As soon as two equivalent keys X and Y are discovered, we merge the groups based on these keys and also add the FDs  $X \rightarrow Y$  and  $Y \rightarrow X$  to the set J while removing the same from the original dependencies set H. The Prolog code goes as follows :

```

step4(REL) :- clo(REL,L1,L1CLO), clo(REL,L2,L2CLO),
              not (L1=L2), subset(L1,L2CLO), subset(L2,L1CLO),
              not (alreadyexistsgroup(REL,L1,L2)),
              merge(REL,L1,L2),
              asserta(fdj(REL,L1,L2)), asserta(fdj(REL,L2,L1)),
              fail.

step4(REL) :- clo(REL,L,L1CLO), retract(clo(REL,L,L1CLO)), fail.

step4(REL) :- fdj(REL,L,R), fd(REL,L,A), subset(A,R),
              retract(fd(REL,L,R)), fail.

```



```

merge(REL,L1,L2) :- group(REL,G1), listelem(L1,G1),
                    group(REL,G2), listelem(L2,G2),
                    retract(group(REL,G1)),
                    retract(group(REL,G2)),
                    append2(G1,G2,NEWGROUP),
                    asserta(group(REL,NEWGROUP)), !.

```

```

alreadyexistsgroup(REL,L1,L2) :-
    group(REL,G),
    listelem(L1,G),listelem(L2,G)

```

A new database fdj(sym,list,list) has been introduced here and corresponds to the set of dependencies J discussed in the algorithm. Whenever the equivalent keys are discovered, groups are merged using the merge predicate which merges two groups with given LHSs. The rule 'merge(REL,L1,L2)', looks for groups G1 and G2 such that G1 is based on a set of keys of which L1 is one and similarly G2 on a set of keys containing L2, and retracts both the groups while asserting the group containing the union of the sets of keys of the groups G1 and G2. The alreadyexistsgroup predicate checks whether such a group already exists so that there is no need for merging the groups.

**Step 5.** This step eliminates the transitive dependencies, unwantedly introduced in the combined set of dependencies due to the step4 of merging groups. For this, we introduce temporarily the dependencies in the set J to the original set F. Then we take up, one by one, the original dependencies in the set F i.e. not introduced just now and check whether its removal has caused any difference in the closure of the dependencies. The closure is found with respect to all the dependencies now present in the set F except the one just now retracted. If the closure of

dependencies is found to be unaffected, this means that the FD was transitive and hence is not put back. At the same time the LHS of the FD is eliminated from the group in which it exists. The clauses are as follows :

```

step5(REL) :- fdj(REL,L,R), not (fd(REL,L,R)),
              assertz(fd(REL,L,R)), fail.

step5(REL) :- fd(REL,L,R), not (fdj(REL,L,R)),
              retr(REL,L,R),
              closure(REL,L,Z),
              choice2(REL,L,R,Z), fail.

step5(REL) :- fdj(REL,L,R), retract(fd(REL,L,R)), fail.

step5(REL).

retr(REL,L,R) :- retract(fd(REL,L,R)), !.

```

The first clause of the rule step5 stores back the FDs present in J to the set F. The second clause takes one FD from the set F at a time and checks that it is not present in the set J. Then it retracts the FD from F, finds the closure Z of its LHS and then call choice2. The third clause retracts the FDs temporarily put in the set F. It is to be noted that we have not applied simply the predicate retract but instead have defined another rule retr(REL,L,R) which performs a `retract(fd(REL,L,R))` followed by a cut i.e. '!'. This ensures that the program does not retract the same FD which it has just now reasserted and hence ruling out any possibility of the program into entering an infinite loop. The choice2 predicate has the following clauses :

```

choice2(REL,L,R,Z) :- not (subset(R,Z)), asserta(fd(REL,L,R)),
                      !.

choice2(REL,L,R,Z) :- subset(R,Z), elimin(REL,L),
                      !.

```

```

elimin(REL,L) :- not (fd(REL,L,_)), group(REL,G),
                  listelem(L,G), retract(group(REL,G)),
                  listminus1(A,G,L), not(equal2(A,[])),
                  asserta(group(REL,A)), !.
elimin(_,_).

```

The first clause of choice2 becomes active if the RHS of the retracted FD is not found to be the subset of the closure Z of its LHS and it asserts back the FD to the set F. The second clause becomes active if the RHS is found to be the subset of Z and calls the predicate elimin which in turn eliminates the LHS from the group which contains it.

step 6. The sixth and the final step consists of the construction of relations from each of the groups that we have after the elimination of transitive dependencies. The various steps involved in forming the relations are as follows : First we select a group, then make name for the required relation, then find the set of attributes i.e. the context for this relation and then assert some of the keys for this relation. The various rules to carry out these functions are makeiname, makeschema and assertsomekeys. The actual clauses are as follows :

```

step6(REL) :- step6b(REL,0).

step6b(REL,N) :- group(REL,G), NEWN = N + 1,
                 makeiname(REL,NEWN,NEWREL),
                 makeschema(REL,NEWREL,G),
                 assertsomekeys(NEWREL,G),
                 assertz(decomp(REL,NEWREL)),
                 assertz(in3nf(NEWREL)),
                 !, step6b(REL,NEWN).

step6b(REL,_) :- killmodifiedfds(REL),
                 reassertrememberedfds(REL).

```

The only work the predicate step6(REL) does is that it calls another rule step6b(REL,0). The predicate step6b(REL,N) first of all chooses a Group and then creates a name for the relation to be formed corresponding to this Group. The name of the relation is formed by the predicate make\_name and depends on the integer N. The predicate make\_schema creates the context for the new relation and asserts the same in the database schema. Some keys are asserted by the predicate assert\_some\_keys. Finally the current Group is retracted and the rule step6b is recursively called again with an incremented 'N'. The rules make\_name, make\_schema and assert\_some\_keys are as follows :

```

make_name(REL,N,NEWREL) :- appendchar(REL,'_',NREL),
                           Suffix = N + 96,
                           char_int(A,Suffix),
                           appendchar(NREL,A,NEWREL).

make_schema(REL,NEWREL,G) :- collect(REL,G,NEWSHEMA),
                              assertz(schema(NEWREL,NEWSHEMA)).

assert_some_keys(NEWREL,G) :- listelem(K,G),
                              assertz(key(NEWREL,K)),
                              fail.

assert_some_keys(_,_).

```

The rule make\_name makes use of another predicate appendchar and a built-in predicate char\_int. The predicate 'char\_int' is a type conversion predicate. appendchar(sym,char,newsym) performs the function of appending the character 'char' to 'sym' and assigning it to 'newsym' if newsym is not already bound. 'Suffix' is an integer equivalent of the suffix required for the NEWREL and is obtained by adding 96 to N because the ASCII code for 'a' is 97. char\_int(A,Suffix) assigns the character equivalent of Suffix to

the variable A which is finally appended to REL to give NEWREL by using appendchar.

Since the LHS common to all (or some) of the FDs in a group is a key to the relation formed for that group, the predicate assertsomekeys(NEWREL,G) looks for the LHSs associated with the group G and then asserts these as keys of NEWREL.

The predicate makeschema makes use of another predicate collect to find the context of the relation which is to be constructed for the group G and then asserts the same in the schema database. The clauses for the predicate collect are as follows :

```
collect(REL,G,RESULT) :- schema(REL,S),
                           collect2(REL,G,S,[],RESULT).
```

```
collect2(REL,G,TOTEST,ACCEPT,RES) :-
    elem(A,TOTEST),
    minus1(NEWTOTEST,TOTEST,A),
    choice3(REL,G,A,ACCEPT,NEWACCEPT),
    collect2(REL,G,NEWTOTEST,NEWACCEPT,RES).
```

```
collect2(_____,ACCEPT,RES) :- RES = ACCEPT.
```

The predicate collect calls another predicate collect2(REL,G,TOTEST,ACCEPT,RES) whose function is to store in RES the context of the relation corresponding to the Group G. TOTEST is the set of attributes from the context is to be found out and ACCEPT is a set of attributes which all form a part of the context. The set ACCEPT is assigned initially the null value i.e. 0 while the set TOTEST is assigned the value equal to the context of the relation REL. The set ACCEPT gradually builds up

as the rule collect recursively calls itself. Finally, when the set ACCEPT is fully built, the second clause of collect assigns its value to RES.

The third clause of the rule step6b calls the predicates killmodifiedfds and reassertrememberedfds which do the job of retracting all the FDs from the sets F and J and then storing back the original FDs to the set F.

### 5.2.7 Tsou & Fischer 's Algorithm

The Tsou & Fischer's algorithm for converting an unnormalized relation directly in a set of BCNF relations was explained in the previous chapter. The predicate bcnf(REL) does this job of converting the relation REL into BCNF relations. This predicate bcnf first of all makes a choice with the help of choice7a predicate. The choice is that if the schema of REL contains only two elements then no decomposition is needed since a relation with two elements is always in BCNF. But if the schema of REL contains more than two elements, choice7a calls for another rule bcnf2. In the rule bcnf2(REL,X,Y,DECOMP), REL is the relation to be decomposed, X is the active set AS of the algorithm, Y is a listoflists which contains the list of contexts of relations so far decomposed and DECOMP is a listoflists which will finally contain lists of contexts of all the relations finally obtained after decomposition. Thus the values of X and Y initially supplied to bcnf2 are S where S is the context of the original relation REL and the null set [] respectively. The clauses are as follows :

```
bcnf(REL) :- schema(REL,S), choice7a(REL,S,[],DECOMP),
            createnewrels(REL,DECOMP,0).
```

```
choice7a :- equal(S,[_,_]), !, equal2(DECOMP,[S]).
```

```
choice7a :- not (equal(S.[_,_])), !, bcnf2(REL,S,[],DECOMP).
```

```
bcnf2(REL,X,Y,DECOMP) :- equal(X,[_,_]), !,
                        append2(Y,[X],DECOMP).
```

```
bcnf2(REL,X,Y,DECOMP) :- not (check(REL,X)),
                        append2(Y,[X],DECOMP).
```

```
bcnf2(REL,X,Y,DECOMP) :- reduce1(REL,X,FINAL_Y,FINAL_A),
                        minus1(NEWX,X,FINAL_A),
                        append2([FINAL_Y],Y,NEWY), !,
                        bcnf2(REL,NEWX,NEWY,DECOMP).
```

There are three clauses for the bcnf2(REL,X,Y,DECOMP) rule. The first clause checks whether the active set X has been reduced to two elements. If so it adds the active set the listoflists Y to give the final DECOMP. The second clause calls for the predicate check which checks whether any element of X can be generated with the help of the other elements. If none can be so generated it adds X to the listoflists Y to give DECOMP. The third clause is called if such an element exists in X. First of all it calls the predicate reduce1(REL,X,FINAL\_Y,FINAL\_A). FINAL\_Y of reduce1 corresponds to the set B of the algorithm and FINAL\_A corresponds to the element E of the algorithm (see section 4.4). Thus for the active set X, the rule reduce1 calculates the set B and the element E which are here called FINAL\_Y and FINAL\_A respectively. The predicate reduce1 in turn calls another reduce2 which is recursive in nature and goes on reducing the active set temporarily and checks whether it can be reduced further. Ultimately it stores the final values in the,

variables FINAL\_Y and FINAL\_A.

The rule createnewrels called by the predicate bcnf does the job of extracting the lists of attributes from the listoflists DECOMP one by one and creating a relation for each. While creating the relations the predicates makebcnfname, addknownkeys and assertdecomp are used respectively to create the names for these relations, adding any keys to that relation using the result that in a BCNF relation every determinant is a key, and in the end asserting the knowledge decomposition by asserting facts to the database decomp. A fact decomp(REL,NEWREL) means that the relation NEWREL has been created after applying some normalization process to the relation REL.

#### 5.2.8 Minimizing a decomposition

As we have seen the decomposition algorithms many times create more relations than are needed. For example, in section 4.4 we showed that the Tsou & Fischer's algorithm produced an extra unwanted relation R\_5. In fact in this example, the relation R\_5 was a subset of the relation R\_2. In such cases when the decomposition produces a relation that is contained in another relation similarly produced, this unwanted relation can be easily removed using the minimize predicate. The minimize rule first takes up two of the decomposed relations and checks whether the context of one is contained in that of the other. If it is so it eliminates the former relation from the schema of the decomposition. The clauses for it are as follows :



```

minimize(REL) :- decomp(REL,REL1), decomp(REL,REL2),
                 not (REL1 = REL2),
                 schema(REL1,S1), schema(REL2,S2),
                 subset(S1,S2),
                 purge(REL1), retract(decomp(REL,REL1)),
                 fail.

```

```

minimize(REL).

```

```

purge(REL) :- retract(schema(REL,_)), fail.
purge(REL) :- retract(key(REL,_)), fail.
purge(REL) :- retract(fd(REL,_,_)), fail.
purge(REL) :- retract(in3nf(REL)), fail.
purge(REL) :- retract(inbcnf(REL)), fail.
purge(REL).

```

The job of eliminating a particular relation is done by the rule purge. purge(REL) retracts all the information about the relation REL from various data bases viz. schema, key, fd, in3nf and inbcnf. After all the initial purge rules have retracted all the facts related with REL from the various databases, the last clause of purge becomes active and makes it true.

#### 5.2.9 Tanaka's Algorithm for 4NF

The Tanaka's algorithm for converting an unnormalized relation containing both FDs and MVDs to a set of 4NF relations using the decomposition approach was explained in the previous chapter. As was mentioned there, this algorithm consists of four steps viz. calculating the set  $F'$ , then the sets  $M'$  and  $M''$  and finally decomposing the relation with the help of the dependencies set  $M''$ . The various steps are implemented as follows :

Step 1. This step calculates the FD set  $F' = FD( :D)$  where  $D = \{FD1-3, MVD0, FD-MVD3\}$ . The rule fm3(REL) applies the axiom FD-MVD3 repeatedly to the dependencies of the relation REL to generate more FDs and continues until no more FDs can be generated. The axiom FD-MVD3 is an FD-MVD interaction axiom and says that " if  $X \twoheadrightarrow Y, U \twoheadrightarrow \twoheadrightarrow V$  in  $W$  and  $X \subseteq W$ , then the following FD holds :  $U(V \cap X) \twoheadrightarrow Y \cap V$ ". The rule fm3 first chooses an FD and an MVD. Then the predicate getw gets the context of the MVD chosen. If the LHS of the FD belongs to the context of the MVD, the rule proceeds further. The predicate common finds the intersection of two sets. Before asserting the FD  $U(V \cap X) \twoheadrightarrow Y \cap V$ , the rule makes three checks using the predicates check1, check2 and check3. These check-predicates check whether the FD we want to assert is trivial or is contained in some other already existing FD. In the end the fm3 clause terminates in the predicate fail. This forces the rule to go on repeating itself until no more FDs can be asserted. Then the second clause of fm3 sets it true. The clauses for fm3 are given as follows :

```
fm3(REL) :- fd(REL,L1,R1), mvd(REL,L2,R2),
            getw(REL,L2,R2,W), subset(L1,W),
            common(A1,R2,L1), common(B1,R2,R1),
            not (equal(B1,[])), union(U,L2,A1,REL),
            not(fd(REL,U,B1)),
            not (check1(REL,U,B1)),
            not (check2(REL,U,B1)),
            not (check3(REL,U,B1)),
            assertz(fd(REL,U,B1)),
            fail.
```

fm3(REL).

The application of the rule fm3 to the set of the given

dependencies introduces some unwanted FDs in the dependencies set. The rules remextral and remextra do the job of removing such unwanted FDs. The rule remextral consists of three clauses. The first clause removes all the trivial FDs i.e. the FDs in which the RHS is a subset of the LHS. The next two clauses remove the FDs which are contained in some other FD. The rule remextra does two functions. It merges the FDs with identical LHSs and modifies the FDs whose LHS is a superset of that of some other FD while whose RHS is not. In the remextra calls remextral to remove any unwanted FD it might have introduced in the system.

The predicate f3a enforces the axiom FD3 in a modified form. It generates new FDs using the transitivity property of the FDs and goes on doing it until no more FDs can be enforced. The clauses for f3a are as follows :

```
f3a(REL) :- fd(REL,A,B), fd(REL,C,D), subset(C,B),
            union(U,B,D,REL), not (fd(REL,A,U)),
            assertz(fd(REL,A,U)), fail.
```

```
f3a(REL).
```

After applying the rule f3a to the set of dependencies it becomes once again necessary to call remextra to remove the unwanted FDs generated.

**Step 2.** The second step of the Tanaka's algorithm consists of finding the intermediate set  $M'$ . In this set, the dependencies are represented in the standard form, as discussed in the previous chapter, represented as follows :

$$X : [Y_0] \ Y_1 \mid Y_2 \mid \dots \mid Y_n ,$$

where  $X \twoheadrightarrow Y_0$  and  $X \twoheadrightarrow \twoheadrightarrow Y_i$  for any  $i > 0$ .

We introduce two databases mdf(sym,list,list) and mdm(sym,list,list) here. Corresponding to a sample dependency in the standard form as shown above, with relation say 'r', the following facts will have to be asserted to these new databases :

mdf(r,[x],[y0]),  
mdm(r,[x],[yi]), for each  $i > 0$ .

It should be noted that as usual, the attribute names had to be converted to lower case.

The process for obtaining M' is discussed in the section 4.4 of the last chapter. The rule mdash is used in the program to carry out this job. When the rule mdash is executed, the intermediate set M' is obtained with dependencies in it represented in the standard form i.e. represented with the help of facts in the data bases mdf and mdm.

Step 3. The step 3 of the Tanaka's algorithm consists of calculating the dependencies set  $M'' = M':E$  i.e. the closure of the intermediate set M' with respect to E where E is the axioms set {MVD0-5}. The MVD6 axiom has been neglected in this algorithm as it is usually neglectable in most of the practical applications. Also we have mentioned that the axiom MVD7 is equivalent to the axioms MVD3-5 and so the set E can be taken as {MVD0-2, MVD7} also. It was found, however, while writing the prolog code for step 3 that it becomes easier to find the set M'' if we use both MVD3 and MVD7, although theoretically MVD7 includes MVD3.

The rule m7 is used to apply the axiom MVD7 to the intermediate set M'. This rule undergoes forced recursion because its first clause terminates in a fail. In each iteration it asserts new facts to the databases mdf and mdm. It also asserts facts to the database mvd2 every time it asserts to mdm. In fact mvd2 keeps the knowledge of which new mdm facts are introduced by the rule m7. Only when no more dependencies are assertable, does this clause terminate and the second clause now makes the rule true. In the process of enforcing axiom MVD7, we introduce many unwanted dependencies. To clean the new set of dependencies, rules clean1 and clean are used.

The rule clean1 simply removes the duplicate mdm's and mvd2's introduced by the rule m7. The clauses for clean1 are as follows :

```

clean1(REL) :- mdm(REL,L,R), retr2(REL,L,R),
              not (mdm(REL,L,R)), asserta(mdm(REL,L,R)), fail.

clean1(REL) :- mvd2(REL,L,R), retr3(REL,L,R),
              not (mvd2(REL,L,R)), asserta(mvd2(REL,L,R)), fail.

clean1(REL).

retr2(REL,L,R) :- retract(mdm(REL,L,R)), !.
retr3(REL,L,R) :- retract(mvd2(REL,L,R)), !.

```

It can be seen that in the clean1 clauses, we have used predicates retr2 and retr3 instead of directly using the retract predicate. In the retr2 and retr3 rules, the retract command is followed by a cut i.e. '!'. This is done to ensure that the program doesn't retract a fact just asserted by it and thus ensuring an impossibility of it's entering into an infinite loop.

The first clause of the rule clean calls the rule clean1. The second clause of it retracts those mdm's whose LHSs are supersets of some other mdm. Its third clause partitions the RHS of those mdm's who have another mdm having the same LHS and an RHS which is a subset of their's.

After cleaning the dependencies generated by the rule m7, the rule m3 is used to enforce a particular case of axiom MVD3 on the dependencies set. This axiom is the Transitivity axiom for MVDs and states that : " if  $X \twoheadrightarrow \twoheadrightarrow Y$  in  $Z$  and  $Y \twoheadrightarrow \twoheadrightarrow W$  in  $Z$  then the following MVD also holds viz.  $X \twoheadrightarrow \twoheadrightarrow W - Y$  in  $Z$ ". The clauses for the rule m3 are as follows :

```
m3(REL) :- mdm(REL,L,R), context(REL,L,R,C),
            mvd2(REL,X,Y),
            subset(X,R), subset(Y,R),
            minus(M,R,Y), not (equal(M,[])),
            assertz(mdm(REL,L,Y)),
            assertz(mdm(REL,L,M)),
            fail.
m3(REL).
```

It is to be noted that while applying the axiom MVD3 to the set of dependencies, m3 considers only the newly asserted mdm's i.e. the mvd2's for generating new mdm's. context is a database which contains information about the contexts of various mdm's.

The rule m3 also introduces unwanted dependencies which are to be removed by using the rule clean again. In the end the rule cleanup retracts all the facts from the temporarily used databases viz. rememb, context and mvd2. cleanup also retracts those mdm's who do not have an mdf with the same LHS.

The rule putspecialmdfs puts in the set  $M''$ , the mdf's and mdm's corresponding to those MVDs whose context is equal to the schema of the original relation REL, and also which satisfy these two conditions : (i) No FD should have an LHS which is superset of the LHS of this MVD. (ii) No MVD should have an LHS which is superset of the LHS of this MVD.

**Step 4.** The final step in the Tanaka's algorithm consists of decomposing the initial relation REL into 4NF relations with the help of the dependencies set  $M''$ . The predicate make4nfrels does this job here. make4nfrels calls a similar predicate make4nfrels1 which in turn completes the process with the help of two predicates getlowest1 and use. The rule getlowest1 selects the minimal dependency satisfying all the conditions discussed in the algorithm in the last chapter. The rule use, after that, creates new relations with the help of that minimal dependency, retracts that dependency and recursively calls make4nfrels1 for each of the newly created relation. When all the newly created relations are converted to 4NF, the process ends.

After obtaining the decomposition, the rule minimize is used to eliminate any relation which is contained totally in some other decomposed relation. And finally, printdecomp prints the decomposition i.e. the details of all the newly created 4NF relations.

APPENDIX - I

```
/*.....RESEARCH PROJECT..... */
/*..... Program for Automatic Normalization ..... */
/*..... up to 4NF ..... */
/* by */
/* SANDEEP GOEL */
/* 1990 */
/*.....Turbo Prolog 2.0 version.....*/
```

```
nowarnings
code=3000
```

```
domains
```

```
file = resfile
sym = symbol
list = sym*
listoflists = list*
int = integer
```

```
database
```

```
schema(sym,list)          fd(sym,list,list)
group(sym,listoflists)    clo(sym,list,list)
rememberfd(sym,list,list) fdj(sym,list,list)

decomp(sym,sym)           in3nf(sym)
key(sym,list)             inbcnf(sym)

mvd(sym,list,list)        mdf(sym,list,list)
mdm(sym,list,list)        rememb(sym,list)
context(sym,list,list,list) mvd2(sym,list,list)
mdmtemp(sym,list,list)    in4nf(sym)
allkey(sym)               remembmdf(sym,list,list)
mvdtemp(sym,list,list)    store(list)
```



predicates

g1	goal1	g2	goal2
g3	goal3	g4	goal4
g5	goal5	g6	goal6
goal1a	goal2a		

equal(list, list)	equal2(listoflists, listoflists)
elem(sym, list)	listelem(list, listoflists)
subset(list, list)	attr(list, sym)
union1(list, list, list, list)	union(list, list, list, sym)
minus1(list, list, sym)	minus(list, list, list)
append(list, list, list)	appendchar(sym, char, sym)
listminus1(listoflists, listoflists, list)	
append2(listoflists, listoflists, listoflists)	
closure(sym, list, list)	reducelhs(sym, list, list, list)
elimattr(sym)	elimredundfds(sym)
choice1(sym, list, list, list)	choice2(sym, list, list, list)
make3nf(sym)	
remembercovering(sym)	step1(sym)
step2(sym)	step3(sym)
step4(sym)	step5(sym)
step6(sym)	step6b(sym, int)
printdecomp(sym)	merge(sym, list, list)
alreadyexistsgroup(sym, list, list)	elimin(sym, list)
retr(sym, list, list)	
writegivenrelation(sym)	writeallfds(sym)
writeallmvds(sym)	writeallrhs(sym, list)
makeiname(sym, int, sym)	reassertrememberedfds(sym)
makeschema(sym, sym, listoflists)	killmodifiedfds(sym)
assertsomekeys(sym, listoflists)	collect(sym, listoflists, list)
collect2(sym, listoflists, list, list, list)	
isvalidattribute(sym, listoflists, sym)	
choice3(sym, listoflists, sym, list, list)	
choice4(sym, list, list)	
choice4a(sym, sym, list)	
minimize(sym)	purge(sym)
makebcnf(sym)	bcnf(sym)
reduce1(sym, list, list, sym)	reduce2(sym, list, sym, list, sym)
check(sym, list)	choice5(sym, list, listoflists)
bcnf2(sym, list, listoflists, listoflists)	
createnewrels(sym, listoflists, integer)	
makebcnfname(sym, integer, sym)	assertdecomp(sym, sym)
addknownkeys(sym, sym)	printrelation(sym)
printallkeys(sym)	printallfds(sym)
choice8(sym)	choice9(sym)
choice10(sym)	

```

make4nf(sym)
fm3(sym)
common(list,list,list)
remextra01(sym)
f3a(sym)
check2(sym,list,list)

mdash(sym)
check4(sym,list,list)
check5(sym,list,list)
writemdash2(sym,list)
getallcontexts(sym)

m7(sym)
check6(sym,list,list,list)
clean1(sym)
retr2(sym,list,list)
cleanup1(sym)

choice15(sym,list,list,list,list,list,list,list)

make4nfrels(sym)
use(sym,sym,list,integer)
getlowest(sym,sym,list)
getlowest2(sym,sym,list)
reloadmdfs(sym)
choice17(sym,sym,list,list,list)

getcontext(sym,list,list)
check8(sym,list)
putspecialmdfs(sym)

getw(sym,list,list,list)
remextra(sym)
getasubset(list,list)
check1(sym,list,list)
check3(sym,list,list)

rtside(sym,list,list)
choice12(list,list,list,list)
writemdash(sym)
writefdash(sym)
writemdoubledash(sym)

getw2(sym,list,list,list)
clean(sym)
m3(sym)
retr3(sym,list,list)
cleanup(sym)

make4nfrels1(sym,sym)
use1(sym,sym,list,integer)
getlowest1(sym,sym,list)
check7(sym,list,list)
choice16(sym,sym,list,list,list)
enough(sym)

getcontext2(sym,list,list,list)
check9(sym,list)
assertmdms(sym,list)

```

clauses

```

/* ... equal(L1,L2) equals two lists L1 and L2 ... */

equal([A|B],[C|D]) :- A=C,equal(B,D).
equal([A],[B]) :- A=B.
equal([],[]).

/* ... equal(LL1,LL2) equals two listoflists LL1 and LL2 ... */

equal2([A|B],[C|D]) :- equal(A,C),equal2(B,D).
equal2([A],[B]) :- equal(A,B).
equal2([],[]).

elem(E,[E|_]).
elem(E,[_|T]) :- elem(E,T).

listelem(E,[E|_]).
listelem(E,[_|T]) :- listelem(E,T).

```

```

subset([],_) :- !.
subset([H|TA],[H|TB]) :- !,subset(TA,TB).
subset(A,[_|TB]) :- subset(A,TB).

attr(E,R) :- schema(R,S), subset(E,S).

union(U,A,B,REL) :- schema(REL,S), subset(A,S), subset(B,S),
    union1(U,A,B,S).

union1(A,A,[],_) :- !.
union1(B,[],B,_) :- !.
union1([H|TU],[H|TA],[H|TB],[H|TS]) :- !,union1(TU,TA,TB,TS).
union1([H|TU],[H|TA],B,[H|TS]) :- !,union1(TU,TA,B,TS).
union1([H|TU],A,[H|TB],[H|TS]) :- !,union1(TU,A,TB,TS).
union1(U,A,B,[_|TS]) :- union1(U,A,B,TS).

/* minus1(R,A,B) means R = A - [B] */

minus1([],[],_) :- !.
minus1(TA,[B|TA],B) :- !.
minus1([HA|TX],[HA|TA],B) :- minus1(TX,TA,B).

listminus1([],[],_) :- !.
listminus1(TA,[B|TA],B) :- !.
listminus1([HA|TX],[HA|TA],B) :- listminus1(TX,TA,B).

/* minus(R,A,B) means R = A - B */

minus(R,R,[]) :- !.
minus(Z,A,[HB|TB]) :- minus1(R,A,HB), minus(Z,R,TB).

/* append (X,Y,Z) means Y+X --> Z */

append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

append2([],L,L).
append2([X|L1],L2,[X|L3]) :- append2(L1,L2,L3).

/* appendchar(Str,Chr,Newstr) appends character Chr
to string Str and stores in string Newstr */

appendchar(SYM,CHR,NEWSYM) :- str_char(B,CHR),
    concat(SYM,B,NEWSYM).

```

```

closure(REL,X,RESULT) :- fd(REL,LHS,RHS), subset(LHS,X),
                        not (subset(RHS,X)),
                        union(U,X,RHS,REL), !,
                        closure(REL,U,RESULT).

closure(_,X,RESULT) :- RESULT = X.

elimattr(REL) :-
    write("Elimination of extraneous attributes from the cover of "),
    write(REL), write(" : "), nl,nl, fd(REL,LHS,RHS),
    reducelhs(REL,LHS,RHS,NEWLHS),
    not (LHS=NEWLHS), retract(fd(REL,LHS,RHS)),
    asserta(fd(REL,NEWLHS,RHS)),
    write(" Extraneous attributes found in the dependency "),
    write("      "),write(LHS), write(" --> "), write(RHS),
    nl,nl,write(" .. The new left hand side = "), write(NEWLHS),
    nl,fail.

elimattr(REL) :- nl,write("* all extraneous attributes eliminated"),nl,nl.

reducelhs(REL,LHS,RHS,NEWLHS) :-
    elem(A,LHS),minus1(Z,LHS,A),
    not (equal(Z,[])),
    closure(REL,Z,ZCLO), subset(RHS,ZCLO),
    !, reducelhs(REL,Z,RHS,NEWLHS).

reducelhs(_,LHS,_,NEWLHS) :- NEWLHS=LHS.

elimredundfds(REL) :-
    write("Elimination of redundant FDs from the cover of "),
    write(REL), write(" : "), nl, fd(REL,LHS,RHS),
    retr(REL,LHS,RHS),
    closure(REL,LHS,Z),
    choice1(REL,LHS,RHS,Z),
    fail.

elimredundfds(_) :- nl,write("* all redundant fds eliminated"),nl,nl.

choice1(REL,LHS,RHS,Z) :- not (subset(RHS,Z)),
                        asserta(fd(REL,LHS,RHS)).

choice1(_,LHS,RHS,Z) :- subset(RHS,Z), nl, write("redundant fd "),
                        write(LHS), write(" --> "),
                        write(RHS), write(" eliminated "),nl.

```

```

/* ----- */
/* Decomposition into 3NF */
/* ----- */

/*..... BERNSTEIN'S ALGORITHM.....*/
/* ----- */

make3nf(REL) :- nl,nl,
    write(" Applying BERNSTEIN'S ALGORITHM for conversion"),
    write(" of ",REL," into 3NF"),nl,
    write(" -----"),
    write(" --      ---- ----"),nl,nl,nl,
    step1(REL), step2(REL), step3(REL),
    step4(REL), step5(REL), step6(REL), printdecomp(REL).

/* ---- step 1 : Eliminating extraneous attributes ---- */
/* ----- */

step1(REL) :- write(" Step 1"),nl, write(" ____ "), nl,nl,
    nl, elimattr(REL).

/* ---- step 2 : Eliminating redundant FDs ---- */
/* ----- */

step2(REL) :- nl,nl,write(" Step 2"),nl, write(" ____ "), nl,nl,
    nl, elimredundfds(REL), remembercovering(REL).

remembercovering(REL) :- fd(REL,LHS,RHS),
    assertz(rememberfd(REL,LHS,RHS)), fail.
remembercovering(_).

/* ---- step 3 : Partitioning into groups with ---- */
/* ----- identical LHSs ----- */

step3(REL) :- nl,nl,write(" Step 3"),nl, write(" ____ "), nl,nl,
    nl, write(" Partitioning of the cover of ",REL),
    write(" into groups with identical LHSs"), nl,nl,
    nl, fd(REL,LHS,_),
    /* equal(A,LHS), not (group(REL,[A])), */
    not(group(REL,[LHS])),
    asserta(group(REL,[LHS])),
    write("Group formed, based on lhs : ",LHS), nl,
    closure(REL,LHS,CLO), asserta(clo(REL,LHS,CLO)),
    fail.

step3(_):- nl, write(" * partition into groups completed "),nl,nl.

```

```

/* ----- step 4 : Merging groups with identical keys ----- */
/* ----- */

step4(REL) :- nl,nl,write(" Step 4"),nl, write(" ____ "), nl,nl,
              nl, write(" Merging groups with equivalent keys :"),
              nl, nl, clo(REL,L1,L1CLO),clo(REL,L2,L2CLO),
              not (L1=L2),
              subset(L2,L1CLO),subset(L1,L2CLO),
              not (alreadyexistsgroup(REL,L1,L2)),
              write("Equivalent keys discovered : ",L1," <--> ",L2),
              nl, merge(REL,L1,L2),
              asserta(fdj(REL,L1,L2)),asserta(fdj(REL,L2,L1)),
              fail.

step4(REL) :- clo(REL,L,L1CLO), retract(clo(REL,L,L1CLO)),
              fail.

step4(REL) :- fdj(REL,L,R), fd(REL,L,A), subset(A,R),
              retract(fd(REL,L,A)), fail.

step4(REL) :- nl,write(" * all equivalent keys are discovered "),nl,
              write(" and the groups are merged"), nl,nl,
              write(" The groups after merging are "), nl,
              group(REL,G), write(" Group : ",G), nl, fail.

step4(_).

merge(REL,L1,L2) :- group(REL,G1), listelem(L1,G1),
                   group(REL,G2), listelem(L2,G2),
                   retract(group(REL,G1)),retract(group(REL,G2)),
                   append2(G1,G2,NEWGROUP),
                   asserta(group(REL,NEWGROUP)), !.

alreadyexistsgroup(REL,L1,L2) :- group(REL,G), listelem(L1,G),
                                 listelem(L2,G).

/* ----- step 5 : Eliminating Transitive Dependencies ----- */
/* ----- */

step5(REL) :- nl,nl,write(" Step 5"),nl, write(" ____ "), nl,nl,
              nl, write(" Elimination of transitive dependencies :"),
              nl, fdj(REL,L,R),not (fd(REL,L,R)),
              assertz(fd(REL,L,R)),
              fail.

step5(REL) :- fd(REL,L,R), not (fdj(REL,L,R)),
              retr(REL,L,R),
              closure(REL,L,Z),

```

```

        choice2(REL,L,R,Z),
        fail.

step5(REL) :- fdj(REL,L,R), retract(fd(REL,L,R)), fail.

step5(REL) :- write(" * transitive dependencies eliminated ."),
              nl,nl,write("      Now finally the groups are :"),
              nl,group(REL,G), write("      group : ",G),nl,fail.

step5(_) :- nl.

retr(REL,L,R) :- retract(fd(REL,L,R)), !.

choice2(REL,L,R,Z) :- not (subset(R,Z)), asserta(fd(REL,L,R)), !.

choice2(REL,L,R,Z) :- subset(R,Z), elimin(REL,L),
                    write("      ",L,"--> ",R," eliminated"),nl, !.

elimin(REL,L) :- not (fd(REL,L,_)), group(REL,G),
                listelem(L,G), retract(group(REL,G)),
                listminus1(Z,G,L),
                not (equal2(Z,[])),
                asserta(group(REL,Z)),!.

elimin(_,_).

/* ----- step 6 : Constructing Relations from the Groups ----- */
/* ----- */

step6(REL) :- nl,nl,write(" Step 6"),nl, write("      ____ "), nl,nl,
              nl, write("      Construction of relations"),nl,nl,
              step6b(REL,0).

step6b(REL,N) :- group(REL,G),NEWN = N + 1,
                makeName(REL,NEWN,NEWREL),
                makeschema(REL,NEWREL,G), assertsSomekeys(NEWREL,G),
                assertz(decomp(REL,NEWREL)), assertz(in3nf(NEWREL)),
                retract(group(REL,G)),
                !, step6b(REL,NEWN).

step6b(REL,_) :- killmodifiedfds(REL), reassertrememberedfds(REL).

makeName(REL,NR,NEWREL) :- appendchar(REL,'_',NREL),
                           SUFFIX = NR + 96,
                           char_int(A,SUFFIX),
                           appendchar(NREL,A,NEWREL).

```

```

makeschema(REL,NEWREL,G) :- collect(REL,G,NEWSHEMA),
                             assertz(schema(NEWREL,NEWSHEMA)).

/* Collecting into RESULT the schema of the synthesized
   relation associated to group G. An attribute A
   belongs to the schema if it belongs to the LHS or
   RHS of some fd whose LHS is in G */

collect(REL,G,RESULT) :- schema(REL,S),
                          collect2(REL,G,S,[],RESULT).

collect2(REL,G,TOTEST,ACCEPT,RES) :-
    elem(A,TOTEST),
    minus1(NEWTOTEST,TOTEST,A),
    choice3(REL,G,A,ACCEPT,NEWACCEPT),
    collect2(REL,G,NEWTOTEST,NEWACCEPT,RES).

collect2(_,_,_,_ACCEPT,RES) :- RES = ACCEPT.

choice3(REL,G,A,ACCEPT,NEWACCEPT) :-
    isvalidattribute(REL,G,A),!,
    union(NEWACCEPT,ACCEPT,[A],REL).
choice3(_,_,_,_ACCEPT,NEWACCEPT) :-
    !,NEWACCEPT = ACCEPT.

isvalidattribute(REL,G,A) :- !, listelem(L,G),          /*****/
                          choice4a(REL,A,L).

choice4a(REL,A,L) :- fd(REL,L,R), choice4(A,L,R).
choice4a(REL,A,L) :- fdj(REL,L,R), choice4(A,L,R).

choice4(A,L,_) :- elem(A,L).
choice4(A,_,R) :- elem(A,R).

assertsomekeys(NEWREL,G) :- listelem(K,G),
                             assertz(key(NEWREL,K)),fail.
assertsomekeys(_,_).

killmodifiedfds(REL) :- fd(REL,L,R), retract(fd(REL,L,R)),fail.
killmodifiedfds(REL) :- fdj(REL,L,R), retract(fdj(REL,L,R)),fail.
killmodifiedfds(_).

reassertrememberedfds(REL) :- rememberfd(REL,L,R),
                              assertz(fd(REL,L,R)),
                              retract(rememberfd(REL,L,R)),
                              fail.
reassertrememberedfds(_).

```



```

/* ----- step1 to step6 of Bernstein's
algorithm over ----- */

/* minimization of a decomposition :
relations whose schema is a subset of another relation
are eliminated from the decomposition */

minimize(REL) :- decomp(REL,REL1), decomp(REL,REL2),
not ( REL1 = REL2 ), schema(REL1,S1),
schema(REL2,S2), subset(S1,S2),
purge(REL1), retract(decomp(REL,REL1)),
nl,nl, write("The decomposed relation "), write(REL1),
write(" is eliminated"), nl,nl,nl, fail.

minimize(_).

purge(REL) :- retract(schema(REL,_)), retract(key(REL,_)), fail.
purge(REL) :- retract(fd(REL,_,_)) , fail.
purge(REL) :- retract(in3nf(REL)) , fail.
purge(REL) :- retract(inbcnf(REL)) , fail.
purge(REL).

/* ----- */
/* Decomposition into BCNF */
/* ----- */

/*..... TSOU & FISCHER 'S ALGORITHM.....*/
/* ----- */

makebcnf(REL) :- nl,nl,
write(" Applying TSOU & FISCHER 'S ALGORITHM for conversion"),
write(" of ",REL," into BCNF"),nl,
write(" ----- ===== --- -----"),
write(" -- ---- ---"),nl,nl,nl,
bcnf(REL), printdecomp(REL).

bcnf(REL) :- schema(REL,S),
choice5(REL,S,DECOMP),
nl, nl, write(" * decomposition completed"), nl,nl,nl,nl,
createnewrels(REL, DECOMP, 0).

choice5(REL,S,DECOMP) :- equal(S,[_,_]), !, equal2(DECOMP,[S]),
write(" Relation ",REL," already in BCNF"),
nl,nl,write("relation decomposed : "),
write(S),nl.

choice5(REL,S,DECOMP) :- not(equal(S,[_,_])),!,
bcnf2(REL,S,[],DECOMP).

```

```

bcnf2(REL,X,Y,DECOMP) :- equal(X,[_,_]), !, append2(Y,[X],DECOMP),
    write("relation decomposed : "),
    write(X),nl.

bcnf2(REL,X,Y,DECOMP) :- not (check(REL,X)),
    append2([X],Y,DECOMP),
    write("relation decomposed : "),
    write(X), nl.

bcnf2(REL,X,Y,DECOMP) :- reduce1(REL,X,FINAL_Y,FINAL_A),
    minus1(NEWX,X,FINAL_A),
    append2([FINAL_Y],Y,NEWY),!,
    write("relation decomposed : "),
    write(FINAL_Y), nl,
    bcnf2(REL,NEWX,NEWY,DECOMP).

check(REL,X) :- elem(A,X), elem(B,X), not (A=B),
    minus(TESTSET,X,[A,B]),
    closure(REL,TESTSET,CLO), elem(A,CLO).

reduce1(REL,X,FINAL_Y,FINAL_A) :-
    elem(A,X), elem(B,X), not (A=B),
    minus(TESTSET,X,[A,B]),
    closure(REL,TESTSET,CLO), elem(A,CLO),
    minus1(NEW_X,X,B), !,
    reduce2(REL,NEW_X,A,FINAL_Y,FINAL_A).

reduce2(REL,X,PREVIOUS_A,FINAL_Y,FINAL_A) :-
    elem(A,X), elem(B,X), not (A=B),
    minus(TESTSET,X,[A,B]),
    closure(REL,TESTSET,CLO), elem(A,CLO),
    minus1(NEW_X,X,B), !,
    reduce2(REL,NEW_X,A,FINAL_Y,FINAL_A).

reduce2(REL,X,PREVIOUS_A,FINAL_Y,FINAL_A) :-
    FINAL_Y = X, FINAL_A = PREVIOUS_A.

createnewrels(REL,DECOMP,NR) :-
    NEWNR = NR + 1, listelem(SCHEMA,DECOMP),
    makebcnfname(REL,NEWNR,NEWREL),
    assertdecomp(REL,NEWREL),
    assertz(schema(NEWREL,SCHEMA)),
    assertz(inbcnf(NEWREL)),
    addknownkeys(REL,NEWREL),
    listminus1(NEWDEC,DECOMP,SCHEMA),
    !,
    createnewrels(REL,NEWDEC,NEWNR).

createnewrels(REL,DECOMP,NR) :-
    decomp(FREL,REL),
    retract(decomp(FREL,REL)).

createnewrels(REL,DECOMP,NR).

```

```

makebcnfname(REL,NR,NEWREL) :- appendchar(REL,'_',NREL),
                               SUFFIX = NR + 48,
                               char_int(A,SUFFIX),
                               appendchar(NREL,A,NEWREL).

addknownkeys(REL,NEWREL) :- fd(REL,LHS,RHS), schema(NEWREL,S),
                              subset(LHS,S), subset(RHS,S),
                              assertz(key(NEWREL,LHS)), fail.
addknownkeys(REL,NEWREL).

assertdecomp(REL,NEWREL) :- decomp(OLDREL,REL),
                              assertz(decomp(OLDREL,NEWREL)),
                              assertz(decomp(REL,NEWREL)).

assertdecomp(REL,NEWREL) :- assertz(decomp(REL,NEWREL)).

/* ----- */
/*           printing a decomposition           */
/* ----- */

printdecomp(REL) :- decomp(REL,REL1), printrelation(REL1), fail.
printdecomp(REL) :- not (decomp(REL,_)), printrelation(REL).
printdecomp(REL).

printrelation(REL) :- schema(REL,S),
                      write("Relation : "), write(REL," "), write(S),
                      choice8(REL), nl,nl,
                      choice9(REL),
                      printallkeys(REL), nl, nl,
                      choice10(REL), nl.

choice8(REL) :- in4nf(REL), !, write(" in 4NF ").
choice8(REL) :- inbcnf(REL), !, write(" in BCNF ").
choice8(REL) :- in3nf(REL), !, write(" in 3NF ").
choice8(REL).

choice9(REL) :- in4nf(REL), write("  KEY: "), !.
choice9(REL) :- write(" Some KEYS : ").

choice10(REL) :- fd(REL,_,_),
                 write("Functional dependencies :"),
                 nl, printallfds(REL),!.

choice10(REL).

printallkeys(REL) :- allkey(REL), write(" an all-key relation"),!.
printallkeys(REL) :- key(REL,K), write(" "), write(K), fail.
printallkeys(REL).

```

```

printallfds(REL) :- fd(REL,LHS,RHS), write(" "),
                  write(LHS), write(" --> "),
                  write(RHS), nl, fail.
printallfds(REL).

/* ----- */
/*      Writing a given relation      */
/* ----- */

writegivenrelation(REL) :- nl,nl,nl, schema(REL,S),
                          write(" The given relation is : "),nl,nl,nl,
                          write("      ",REL," : ",S),nl,nl,nl,
                          write(" F :"),nl,nl, writeallfds(REL),
                          nl,nl,nl,
                          mvd(REL,_,_), write(" M :"), nl,nl,
                          writeallmvds(REL),nl,nl,nl.

writegivenrelation(REL).

writeallfds(REL) :- fd(REL,L,R), write("      ",L," : ",R), nl, fail.
writeallfds(REL).

writeallmvds(REL) :- mvd(REL,L,R), assert(mvdtemp(REL,L,R)),
                    not (store(L)), assert(store(L)),fail.
writeallmvds(REL) :- store(L), write("      ",L," : "),
                    writeallrhs(REL,L), retract(store(L)),nl,
                    fail.
writeallmvds(REL) :- retract(mvdtemp(REL,_,_)), fail.
writeallmvds(REL).

writeallrhs(REL,L) :- mvdtemp(REL,L,R1), mvdtemp(REL,L,R2),
                    not(equal(R1,R2)),write(R1," | "),
                    retract(mvdtemp(REL,L,R1)),fail.
writeallrhs(REL,L) :- mvdtemp(REL,L,R), write(R),
                    retract(mvdtemp(REL,L,R)).

/* ----- */
/*      Decomposition into 4NF      */
/* ----- */

/*..... TANAKA 'S ALGORITHM .....*/
/*      ----- */

make4nf(REL) :- nl,nl,
               write(" Applying TANAKA 'S ALGORITHM for conversion"),
               write(" of ",REL," into 4NF"),nl,
               write(" ----- ===== -----"),
               write(" --      ---- ----"),nl,nl,nl,
               fm3(REL), remextra(REL),
               f3a(REL), remextra(REL), writefdash(REL),

```

```

mdash(REL), writemdash(REL),
m7(REL), clean(REL),
m3(REL), clean(REL),
cleanup(REL), putspecialmdfs(REL),
writemdoubledash(REL),
make4nfrels(REL), minimize(REL),
printdecomp(REL).

common(I,L1,L2) :- minus(A,L1,L2), minus(I,L1,A).

fm3(REL) :- fd(REL,L1,R1), mvd(REL,L2,R2),
            getw(REL,L2,R2,W), subset(L1,W),
            common(A1,R2,L1), common(B1,R2,R1),
            not(equal(B1,[])), union(U,L2,A1,REL),
            not(fd(REL,U,B1)),
            not(check1(REL,U,B1)),
            not(check2(REL,U,B1)),
            not(check3(REL,U,B1)),
            assertz(fd(REL,U,B1)),
            fail.
fm3(_).

getw(REL,L,R,W) :- mvd(REL,L,R1), not(equal(R,R1)),
                  union(U,R,R1,REL), union(W,L,U,REL).

check1(REL,U,B1) :- subset(B1,U).

check2(REL,U,B1) :- fd(REL,U,R), subset(B1,R).

check3(REL,U,B1) :- fd(REL,L,R), not(equal(L,U)), subset(L,U),
                  subset(B1,R).

remextra1(REL) :- fd(REL,L,R), subset(R,L), retract(fd(REL,L,R)), fail.

remextra1(REL) :- fd(REL,L1,R1), fd(REL,L1,R2), not(equal(R1,R2)),
                  subset(R1,R2), retract(fd(REL,L1,R1)), fail.

remextra1(REL) :- fd(REL,L1,R1), fd(REL,L2,R2), not(equal(L1,L2)),
                  subset(L1,L2), subset(R2,R1), retract(fd(REL,L2,R2)),
                  fail.

remextra(REL) :- remextra1(REL).

remextra(REL) :- fd(REL,L1,R1), fd(REL,L1,R2), not(equal(R1,R2)),
                  union(U,R1,R2,REL),
                  retract(fd(REL,L1,R1)), retract(fd(REL,L1,R2)),
                  assertz(fd(REL,L1,U)),
                  fail.

```

```

remextra(REL) :- fd(REL,L1,R1), fd(REL,L2,R2), not(equal(L1,L2)),
subset(L1,L2), union(U,R1,R2,REL),
not(fd(REL,L2,U)), assertz(fd(REL,L2,U)),
fail.

remextra(REL) :- remextra1(REL).

remextra(REL).

getasubset(A,[_|TB]) :- getasubset(A,TB).
getasubset([H|TA],[H|TB]) :- !, getasubset(TA,TB).
getasubset([],_) :- !.

f3a(REL) :- fd(REL,A,B), fd(REL,C,D), subset(C,B),
union(U,B,D,REL), not(fd(REL,A,U)),
assertz(fd(REL,A,U)), fail.
f3a(_).

putspecialmdfs(REL) :- schema(REL,S), mvd(REL,L,_),
not(fd(REL,L,_)),
not(check8(REL,L)),
not(check9(REL,L)),
getcontext(REL,L,CONT),
equal(CONT,S),
not(mdf(REL,L,L)),
assertz(mdf(REL,L,L)),
assertmdms(REL,L),
fail.
putspecialmdfs(REL).

check8(REL,L) :- fd(REL,L1,_), subset(L,L1).

check9(REL,L) :- mvd(REL,L1,R), not(equal(L,L1)),
subset(L1,L).

assertmdms(REL,L) :- mvd(REL,L,R), not(mdm(REL,L,R)),
assert(mdm(REL,L,R)), fail.
assertmdms(REL,L).

getcontext(REL,L,CONT) :- mvd(REL,L,R), assert(mvdtemp(REL,L,R)), fail.
getcontext(REL,L,CONT) :- getcontext2(REL,L,L,CONT).

getcontext2(REL,L,P,CONT) :- mvdtemp(REL,L,R), union(U,P,R,REL),
retract(mvdtemp(REL,L,R)),
!, getcontext2(REL,L,U,CONT).
getcontext2(REL,L,P,CONT) :- CONT = P.

```

```

/*.....Calculating M ' from F ' .....*/

mdash(REL) :- mvd(REL,L,R), getasubset(L1,L),fd(REL,L1,R1),
              subset(L,R1),minus(M,L,L1),not(equal(M,[])),
              assertz(mdf(REL,L1,M)), rtside(REL,L,L1),
              fail.

mdash(REL) :- fd(REL,L,R), mvd(REL,L,R1), union(U,L,R,REL),
              minus(M,R1,U), not(check5(REL,L,M)),
              assertz(mdm(REL,L,M)),
              fail.

mdash(REL) :- rememb(REL,L), retract(mdm(REL,L,_)), fail.

mdash(REL) :- fd(REL,L,R), schema(REL,S),
              not(check4(REL,L,R)),
              assertz(mdf(REL,L,R)),
              union(U,L,R,REL), minus(M,S,U),
              assertz(mdm(REL,L,M)),
              fail.

mdash(REL) :- mdm(REL,L,R1), mdm(REL,L,R2), mdm(REL,L,R3),
              not(equal(R1,R2)), not(equal(R2,R3)),
              not(equal(R1,R3)), union(U,R2,R3,REL),
              equal(U,R1), retract(mdm(REL,L,R1)),
              fail.

mdash(REL) :- mdf(REL,L,R), minus(M,R,L), not(equal(M,R)),
              retract(mdf(REL,L,R)), assertz(mdf(REL,L,M)),
              fail.

mdash(_).

check4(REL,L,R) :- fd(REL,X,Y), choice12(L,R,X,Y),
                  union(U,L,R,REL), union(UX,X,Y,REL),
                  subset(U,UX), subset(X,L).

choice12(L,R,X,Y) :- not(equal(L,X)).

choice12(L,R,X,Y) :- not(equal(R,Y)).

check5(REL,L,M) :- equal(M,[]), assert(rememb(REL,L)).

rtside(REL,L,L1) :- mvd(REL,L,R), fd(REL,L1,R1), common(C,R,R1),
                   mdf(REL,L1,M), union(M1,M,C,REL),
                   minus(N,R,C), assertz(mdm(REL,L1,N)), fail.
rtside(REL,L,L1).

```

```

writefdash(REL) :- nl,nl,write(" The set F' is found to be :"),
                  nl,nl,fail.
writefdash(REL) :- fd(REL,L,R), write(" ",L," : ",R), nl, fail.
writefdash(REL).

writemdash(REL) :- nl,nl,write(" The set M' is found to be :"),
                  nl,nl,fail.
writemdash(REL) :- mdf(REL,L,R),
                  write(" ",L," : |*|",R,"|*|"),
                  writemdash2(REL,L),nl,fail.
writemdash(REL).

writemdash2(REL,L) :- mdm(REL,L,R1), write("| ",R1," "), fail.
writemdash2(REL,L).

writemdoubledash(REL) :- nl,nl,write(" The set M\" is found to be :"),
                        nl,nl,fail.
writemdoubledash(REL) :- mdf(REL,L,R),
                        write(" ",L," : |*|",R,"|*|"),
                        writemdash2(REL,L),nl,fail.
writemdoubledash(REL) :- nl,nl,nl.

/*.....Calculating M" from M' .....*/

getw2(REL,L,R,W) :- check6(REL,L,R,R1),
                  union(U,R,R1,REL), union(W,L,U,REL),I.

getw2(REL,L,R,W) :- union(W,L,R,REL),I.

check6(REL,L,R,R1) :- mdm(REL,L,R1),
                    common(C,R,R1), equal(C,[]).

getallcontexts(REL) :- mdm(REL,L,R), getw2(REL,L,R,W),
                    not(context(REL,L,R,W)),
                    assertz(context(REL,L,R,W)),
                    fail.

getallcontexts(REL) :- context(REL,L,R,W), mdf(REL,L,R1),
                    union(U,W,R1,REL), retract(context(REL,L,R,W)),
                    asserta(context(REL,L,R,U)),fail.

getallcontexts(REL).

m7(REL) :- getallcontexts(REL), fail.

m7(REL) :- mdm(REL,X,Y),assertz(mdmtemp(REL,X,Y)), fail.

```



```

m7(REL) :- mdm(REL,X,Y),
           mdmtemp(REL,X,Y),
           retract(mdmtemp(REL,X,Y)),
           mdmtemp(REL,U,V),
           not(equal(X,U)),
           common(D,Y,V), not(equal(D,[])),
           context(REL,X,Y,Z),context(REL,U,V,W),
           subset(X,W), subset(U,Z),
           minus(A,Y,W), minus(B,Z,A),
           common(C1,Y,U),union(U1,X,C1,REL),
           common(C2,V,X),union(U2,U,C2,REL),
           not(mdm(REL,U1,D)),not(mvd2(REL,U1,D)),
           assertz(mvd2(REL,U1,D)),assertz(mdm(REL,U1,D)),
           not(mdm(REL,U2,D)),not(mvd2(REL,U2,D)),
           assertz(mvd2(REL,U2,D)),assertz(mdm(REL,U2,D)),
           not(mdmtemp(REL,U1,D)),not(mdmtemp(REL,U2,D)),
           assertz(mdmtemp(REL,U1,D)),assertz(mdmtemp(REL,U2,D)),
           choice15(REL,U1,U2,X,U,D,Z,W),
           fail.

```

```

m7(REL).

```

```

choice15(REL,U1,U2,X,U,D,Z,W) :-
           equal(X,U1),not(context(REL,U1,D,Z)),
           assertz(context(REL,U1,D,Z)),fail.

```

```

choice15(REL,U1,U2,X,U,D,Z,W) :-
           equal(U,U2),not(context(REL,U2,D,W)),
           assertz(context(REL,U2,D,W)),fail.

```

```

choice15(REL,U1,U2,X,U,D,Z,W).

```

```

clean(REL) :- clean1(REL),fail.

```

```

clean(REL) :- mdm(REL,L,R), mdm(REL,L1,R), not(equal(L,L1)),
           subset(L,L1), retract(mdm(REL,L1,R)),
           retract(mvd2(REL,L1,R)),fail.

```

```

clean(REL) :- mdm(REL,L,R), mdm(REL,L,R1),
           not(equal(R,R1)),
           subset(R,R1), minus(M,R1,R),
           retract(mdm(REL,L,R1)),
           not(mdm(REL,L,M)),assertz(mdm(REL,L,M)),
           fail.

```

```

clean(_).

```

```

clean1(REL) :- mdm(REL,L,R), retr2(REL,L,R),
           not(mdm(REL,L,R)),asserta(mdm(REL,L,R)),fail.

```

```

clean1(REL) :- mvd2(REL,L,R), retr3(REL,L,R),
               not(mvd2(REL,L,R)),asserta(mvd2(REL,L,R)),fail.

clean1(_).

retr2(REL,L,R) :- retract(mdm(REL,L,R)),!

retr3(REL,L,R) :- retract(mvd2(REL,L,R)),!

m3(REL) :- mdm(REL,L,R), context(REL,L,R,C),
           mvd2(REL,X,Y),
           subset(X,R),subset(Y,R), minus(M,R,Y), not(equal(Y,[ ])),
           assertz(mdm(REL,L,Y)),
           assertz(mdm(REL,L,M)),
           fail.

m3(_).

cleanup1(REL) :- mdm(REL,L,R), not(mdf(REL,L,_)),
                retract(mdm(REL,L,R)), fail.

cleanup1(REL).

cleanup(REL) :- cleanup1(REL), fail.
cleanup(REL) :- retract(rememb(_,_)), fail.
cleanup(REL) :- retract(context(_,_,_)), fail.
cleanup(REL) :- retract(mvd2(_,_,_)), fail.
cleanup(REL).

/* ..... making decomposed relations from M" .....*/

make4nfrels(REL) :- appendchar(REL,'_',NEWREL),
                   schema(REL,S), assert(schema(NEWREL,S)),
                   make4nfrels1(REL,NEWREL),fail.

make4nfrels(REL) :- decomp(REL,REL1), not(in4nf(REL1)),
                   assert(in4nf(REL1)), assert(allkey(REL1)),
                   fail.

make4nfrels(REL).

make4nfrels1(REL,REL1) :- getlowest1(REL,REL1,L), !,
                          not(equal(L,[ ])),use(REL,REL1,L,1).

make4nfrels1(REL,REL1).

```

```

use(REL,REL1,L,N) :-
    mdf(REL,L,R), schema(REL1,S), common(C,R,S),
    not(equal(C,[])), union(U,C,L,REL),
    makeName(REL1,N,REL2), assertz(schema(REL2,U)),
    assertz(decomp(REL,REL2)),
    M=N+1, retract(mdf(REL,L,R)),
    assert(in4nf(REL2)), assert(key(REL2,L)),
    use1(REL,REL1,L,M).

use1(REL,REL1,L,N) :-
    mdm(REL,L,R), schema(REL1,S), common(C,R,S),
    not(equal(C,[])), union(U,C,L,REL),
    makeName(REL1,N,REL2), assertz(schema(REL2,U)),
    assertz(decomp(REL,REL2)),
    assertz(decomp(REL1,REL2)),
    M=N+1, retract(mdm(REL,L,R)),
    use1(REL,REL1,L,M), fail.

use1(REL,REL1,L,N) :- decomp(REL1,REL2),
    not(in4nf(REL2)),
    make4nfrels1(REL,REL2),
    retract(decomp(REL,REL2)),
    retract(decomp(REL1,REL2)),
    fail.

use1(REL,REL1,L,N).

getlowest1(REL,REL1,L1) :- getlowest(REL,REL1,L1), reloadmdfs(REL).
getlowest1(REL,REL1,L1) :- equal(L1,[]), reloadmdfs(REL).

getlowest(REL,REL1,L1) :- enough(REL),
    getlowest2(REL,REL1,L1),
    reloadmdfs(REL), !.

getlowest(REL,REL1,L1) :- reloadmdfs(REL),
    enough(REL),
    equal(L1,[]),
    assert(in4nf(REL1)), assert(allkey(REL1)).

getlowest(REL,REL1,L1) :- reloadmdfs(REL),
    not(enough(REL)),
    mdf(REL,L,R), schema(REL1,S1),
    subset(L,S1), L1=L,!.

getlowest(REL,REL1,L1) :- equal(L1,[]),
    assert(in4nf(REL1)), assert(allkey(REL1)).

enough(REL) :- mdf(REL,L,R), mdf(REL,X,Y), not(equal(L,X)), !.

getlowest2(REL,REL1,L1) :- mdf(REL,L,R),
    not(check7(REL,L,R)),
    schema(REL1,S1),
    choice16(REL,REL1,L,S1,L1).

```

```

choice16(REL,REL1,L,S1,L1) :- subset(L,S1), L1=L.

choice16(REL,REL1,L,S1,L1) :- not(subset(L,S1)),
                                retract(mdf(REL,L,R)),
                                choice17(REL,REL1,L,R,L1).

choice17(REL,REL1,L,R,L1) :- not(remembmdf(REL,L,R)),
                                assert(remembmdf(REL,L,R)),
                                getlowest2(REL,REL1,L1).

choice17(REL,REL1,L,R,L1) :- getlowest2(REL,REL1,L1).

check7(REL,L,R) :- mdf(REL,X,Y), not(equal(L,X)), subset(Y,R).

reloadmdfs(REL) :- remembmdf(REL,L,R), retract(remembmdf(REL,L,R)),
                  assert(mdf(REL,L,R)), fail.

reloadmdfs(REL).

g1 :- assert(schema(r1,[r,n,s,m,mn,fn])),
      assert(fd(r1,[r],[n])),
      assert(fd(r1,[n],[r])),
      assert(fd(r1,[n],[mn])),
      assert(fd(r1,[n],[fn])),
      assert(fd(r1,[n,s],[m])),
      assert(fd(r1,[fn],[mn])),
      assert(fd(r1,[mn],[fn])).

g2 :- assert(schema(r2,[a,b,c,d,e,f])),
      assert(fd(r2,[a,b],[c])),
      assert(fd(r2,[c],[a])),
      assert(fd(r2,[d],[e])),
      assert(fd(r2,[d,e],[f])),
      assert(fd(r2,[e],[d])),
      assert(fd(r2,[e],[f])).

g3 :- assert(schema(r3,[a,b,c,d,e])),
      assert(fd(r3,[a,b],[c])),
      assert(fd(r3,[d],[b,e])),
      assert(mvd(r3,[b],[a,c])),
      assert(mvd(r3,[b],[d,e])).

g4 :- assert(schema(r4,[a,b,c,d,e])),
      assert(fd(r4,[b],[c])),
      assert(mvd(r4,[a,b],[c,d])),
      assert(mvd(r4,[a,b],[e])).

```

```

g5 :- assert(schema(r5,[a,b,c,d])),
      assert(mvd(r5,[a,b],[c])),
      assert(mvd(r5,[a,b],[d])),
      assert(mvd(r5,[a],[b])),
      assert(mvd(r5,[a],[c])),
      assert(mvd(r5,[a],[d])).

g6 :- assert(schema(r6,[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q])),
      assert(fd(r6,[g],[d,k,l,m])),
      assert(fd(r6,[a,c],[o,p,q])),
      assert(fd(r6,[h],[a,b,n])),
      assert(mvd(r6,[a,b],[c,d,e,f,k,l,m])),
      assert(mvd(r6,[a,b],[g,h,i,j,n,o,p,q])),
      assert(mvd(r6,[c],[a,e,l,m])),
      assert(mvd(r6,[c],[b,f,o,p])),
      assert(mvd(r6,[d],[a,h,l])),
      assert(mvd(r6,[d],[b,i,j,m,n,o,p])),
      assert(mvd(r6,[f],[a,b,g])),
      assert(mvd(r6,[f],[h,i,j,l,m])),
      assert(mvd(r6,[c,h],[a,d])),
      assert(mvd(r6,[c,h],[b,e,f])),
      assert(mvd(r6,[k],[l,m])),
      assert(mvd(r6,[k],[a,b,p,q])),
      assert(mvd(r6,[l],[p,q])),
      assert(mvd(r6,[l],[c])),
      assert(mvd(r6,[m],[n,o])),
      assert(mvd(r6,[m],[c])).

goal1 :- g1,
         openwrite(resfile,"result.dat"),
         writedeviceresfile),
         writegivenrelation(r1),
         make3nf(r1),
         flush(resfile).

goal1a :- g1,
         openappend(resfile,"result.dat"),
         writedeviceresfile),
         makebcnf(r1),
         flush(resfile).

goal2 :- g2,
         openappend(resfile,"result.dat"),
         writedeviceresfile),
         writegivenrelation(r2),
         make3nf(r2),
         flush(resfile).

goal2a :- g2,
         openappend(resfile,"result.dat"),
         writedeviceresfile),
         makebcnf(r2),
         flush(resfile).

```

```
goal3 :- g3,
        openappend(resfile,"result.dat"),
        writedeviceresfile),
        writegivenrelation(r3),
        make4nf(r3),
        flush(resfile).

goal4 :- g4,
        openappend(resfile,"result.dat"),
        writedeviceresfile),
        writegivenrelation(r4),
        make4nf(r4),
        flush(resfile).

goal5 :- g5,
        openappend(resfile,"result.dat"),
        writedeviceresfile),
        writegivenrelation(r5),
        make4nf(r5),
        flush(resfile).

goal6 :- g6,
        openappend(resfile,"result.dat"),
        writedeviceresfile),
        writegivenrelation(r6),
        make4nf(r6),
        flush(resfile).
```

APPENDIX - II

EXAMPLE - 1

=====

goal : goal1  
=====

The given relation is :

r1 : ["r","n","s","m","mn","fn"]

F :

["r"] : ["n"]  
["n"] : ["r"]  
["n"] : ["mn"]  
["n"] :- ["fn"]  
["n","s"] : ["m"]  
["fn"] : ["mn"]  
["mn"] : ["fn"]

Applying BERNSTEIN'S ALGORITHM for conversion of r1 into 3NF

-----

Step 1

-----

Elimination of extraneous attributes from the cover of r1 :

\* all extraneous attributes eliminated

Step 2

-----

Elimination of redundant FDs from the cover of r1 :

redundant fd ["n"] --> ["mn"] eliminated

\* all redundant fds eliminated

Step 3

-----

Partitioning of the cover of r1 into groups with identical LHSs

Group formed, based on lhs : ["mn"]  
Group formed, based on lhs : ["fn"]  
Group formed, based on lhs : ["n","s"]  
Group formed, based on lhs : ["n"]  
Group formed, based on lhs : ["r"]

\* partition into groups completed

Step 4

-----

Merging groups with equivalent keys :

Equivalent keys discovered : ["r"] <--> ["n"]  
Equivalent keys discovered : ["fn"] <--> ["mn"]

\* all equivalent keys are discovered  
and the groups are merged

The groups after merging are

Group : [{"fn"}, {"mn"}]  
Group : [{"r"}, {"n"}]  
Group : [{"n"}, {"s"}]

Step 5

-----

Elimination of transitive dependencies :

\* transitive dependencies eliminated .

Now finally the groups are :

group : [{"fn"}, {"mn"}]  
group : [{"r"}, {"n"}]  
group : [{"n"}, {"s"}]

Step 6

-----

Construction of relations



Relation : r1\_a ["mn","fn"] in 3NF

Some KEYS : ["fn"] ["mn"]

Relation : r1\_b ["r","n","fn"] in 3NF

Some KEYS : ["r"] ["n"]

Relation : r1\_c ["n","s","m"] in 3NF

Some KEYS : ["n","s"]

goal : goal1a

=====

Applying TSOU & FISCHER 'S ALGORITHM for conversion of r1 into BCNF

-----

relation decomposed : ["r","n"]

relation decomposed : ["n","s","m"]

relation decomposed : ["mn","fn"]

relation decomposed : ["n","fn"]

relation decomposed : ["n","s"]

\* decomposition completed

Relation : r1\_1 ["n","fn"] in BCNF

Some KEYS : ["n"]

Relation : r1\_2 ["mn","fn"] in BCNF

Some KEYS : ["fn"] ["mn"]

Relation : r1\_3 ["n","s","m"] in BCNF

Some KEYS : ["n","s"]

Relation : r1\_4 ["r","n"] in BCNF

Some KEYS : ["r"] ["n"]

Relation : r1\_5 ["n","s"] in BCNF

Some KEYS :

EXAMPLE - 2

=====

goal : goal2  
=====

The given relation is :

r2 : ["a","b","c","d","e","f"]

F :

["a","b"] : ["c"]  
["c"] : ["a"]  
["d"] : ["e"]  
["d","e"] : ["f"]  
["e"] : ["d"]  
["e"] : ["f"]

Applying BERNSTEIN'S ALGORITHM for conversion of r2 into 3NF

-----

Step 1

-----

Elimination of extraneous attributes from the cover of r2 :

Extraneous attributes found in the dependency ["d","e"] --> ["f"]

.. The new left hand side = ["e"]

\* all extraneous attributes eliminated

Step 2

-----

Elimination of redundant FDs from the cover of r2 :

redundant fd ["e"] --> ["f"] eliminated

\* all redundant fds eliminated

Step 3

----

Partitioning of the cover of r2 into groups with ic

Group formed, based on lhs : ["e"]  
Group formed, based on lhs : ["d"]  
Group formed, based on lhs : ["c"]  
Group formed, based on lhs : ["a","b"]

\* partition into groups completed

Step 4

----

Merging groups with equivalent keys :

Equivalent keys discovered : ["d"] <--> ["e"]

\* all equivalent keys are discovered  
and the groups are merged

The groups after merging are

Group : [{"d"}, {"e"}]  
Group : [{"a"}, {"b"}]  
Group : [{"c"}]

Step 5

----

Elimination of transitive dependencies :

\* transitive dependencies eliminated .

Now finally the groups are :

group : [{"d"}, {"e"}]  
group : [{"a"}, {"b"}]  
group : [{"c"}]

Step 6

----

Construction of relations

Relation : r2\_a [{"d"}, {"e"}, {"f"}] in 3NF

Some KEYS : [{"d"}] [{"e"}]

Relation : r2\_b ["a","b","c"] in 3NF

Some KEYS : ["a","b"]

Relation : r2\_c ["a","c"] in 3NF

Some KEYS : ["c"]

goal : goal2a

=====

Applying TSOU & FISCHER 'S ALGORITHM for conversion of r2 into BCNF

-----

relation decomposed : ["a","c"]

relation decomposed : ["d","e"]

relation decomposed : ["e","f"]

relation decomposed : ["b","c","e"]

\* decomposition completed

Relation : r2\_1 ["b","c","e"] in BCNF

Some KEYS :

Relation : r2\_2 ["e","f"] in BCNF

Some KEYS : ["e"]

Relation : r2\_3 ["d","e"] in BCNF

Some KEYS : ["d"] ["e"]

Relation : r2\_4 ["a","c"] in BCNF

Some KEYS : ["c"]

EXAMPLE - 3

=====

goal : goal3  
=====

The given relation is :

r3 : ["a","b","c","d","e"]

F :

["a","b"] : ["c"]  
["d"] : ["b","e"]

M :

["b"] : ["a","c"] | ["d","e"]

Applying TANAKA 'S ALGORITHM for conversion of r3 into 4NF

-----

The set F' is found to be :

["a","b"] : ["c"]  
["d"] : ["b","e"]

The set M' is found to be :

["a","b"] : |\*|["c"]|\*| ["d","e"]  
["d"] : |\*|["b","e"]|\*| ["a","c"]

The set M'' is found to be :

["a","b"] : |\*|["c"]|\*| ["d","e"]  
["d"] : |\*|["b","e"]|\*| ["a","c"]

Relation : r3\_a ["a","b","c"] in 4NF

KEY : ["a","b"]

Relation : r3\_\_b\_a ["b","d","e"] in 4NF

KEY : ["d"]

Relation : r3\_\_b\_b ["a","d"] in 4NF

KEY : an all-key relation

EXAMPLE - 4

=====

goal : goal4.

=====

The given relation is :

r4 : ["a","b","c","d","e"]

F :

["b"] : ["c"]

M :

["a","b"] : ["c","d"] | ["e"]

Applying TANAKA 'S ALGORITHM for conversion of r4 into 4NF

----- =====

The set F' is found to be :

["b"] : ["c"]

The set M' is found to be :

["b"] : [\*|["c"]|\*] ["a","d","e"]

The set M'' is found to be :

["b"] : [\*|["c"]|\*] ["a","d","e"]

["a","b"] : [\*|["a","b"]|\*] ["c","d"] | ["e"]

The decomposed relation r4\_\_b\_a is eliminated

Relation : r4\_\_a ["b","c"] in 4NF

KEY : ["b"]

Relation : r4\_\_b\_b ["a","b","d"] in 4NF

KEY : an all-key relation

Relation : r4\_\_b\_c ["a","b","e"] in 4NF

KEY : an all-key relation

EXAMPLE - 5

=====

goal :: goal5

=====

The given relation is :

r5 : ["a","b","c","d"]

F :

M :

["a","b"] : ["c"] | ["d"]

["a"] : ["b"] | ["b"] | ["c"] | ["d"]

Applying TANAKA 'S ALGORITHM for conversion of r5 into 4NF

-----

The set F' is found to be :

The set M' is found to be :

The set M" is found to be :

["a"] : [\*|["a"]|\*] ["b"] | ["c"] | ["d"]

The decomposed relation r5\_\_a is eliminated

Relation : r5\_\_b ["a","b"] in 4NF

KEY : an all-key relation

Relation : r5\_\_c ["a","c"] in 4NF

KEY : an all-key relation

Relation : r5\_\_d ["a","d"] in 4NF

KEY : an all-key relation

#### EXAMPLE - 6

=====

goal : goal6

=====

The given relation is :

r6 : ["a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q"]

F :

["g"] : ["d","k","l","m"]

["a","c"] : ["o","p","q"]

["h"] : ["a","b","n"]

M :

["a","b"] : ["c","d","e","f","k","l","m"] |

["g","h","i","j","n","o","p","q"]

["c"] : ["a","e","l","m"] | ["b","f","o","p"]

["d"] : ["a","h","l"] | ["b","i","j","m","n","o","p"]

["f"] : ["a","b","g"] | ["h","i","j","l","m"]

["c","h"] : ["a","d"] | ["b","e","f"]

["k"] : ["l","m"] | ["a","b","p","q"]

["l"] : ["p","q"] | ["c"]

["m"] : ["n","o"] | ["c"]



The set F' is found to be :

- ["l"] : ["p"]
- ["m"] : ["o"]
- ["d"] : ["b","n","p"]
- ["c"] : ["a","l","m","o","p"]
- ["a","d"] : ["b","l","n","p"]
- ["b","d"] : ["b","m","n","o","p"]
- ["a","c"] : ["a","l","m","o","p","q"]
- ["h"] : ["a","b","d","k","l","m","n","o"]
- ["f"] : ["a","b","d","k","l","m","n","o"]
- ["k"] : ["l","m","o","p"]
- ["a","b"] : ["b","d","k","l","m","n","o"]
- ["g"] : ["b","d","k","l","m","n","o","p"]

The set M' is found to be :

- ["l"] : [\*|["p"]|\*| ["q"] | ["c"] |  
["a","b","c","d","e","f","g","h","i"]
- ["m"] : [\*|["o"]|\*| ["n"] | ["c"] |  
["a","b","c","d","e","f","g","h","i"]
- ["d"] : [\*|["b","n","p"]|\*| ["a","h","l"]  
["a","c","e","f","g","h","i","j","k","m","o","q"]
- ["c"] : [\*|["a","l","m","o","p"]|\*| ["e"] | ["b","f"] |  
["b","d","e","f","g","h","i","j","k","n","q"]
- ["a","d"] : [\*|["b","l","n","p"]|\*|  
["c","e","f","g","h","i","j","k","m","o","q"]
- ["h"] : [\*|["a","b","d","k","l","m","n","o","p","q"]|\*|  
["c","e","f","g","i","j"]
- ["f"] : [\*|["a","b","d","k","l","m","n","o","p","q"]|\*| ["g"] |  
["h","i","j"] | ["c","e","g","h","i","j"]
- ["k"] : [\*|["l","m","o","p"]|\*|  
["a","b","c","d","e","f","g","h","i","j","n","q"]
- ["g"] : [\*|["b","d","k","l","m","n","o","p"]|\*|  
["a","c","e","f","h","i","j","q"]
- ["b","d"] : [\*|["m","n","o","p"]|\*|  
["a","c","e","f","g","h","i","j","k","l","q"]
- ["a","c"] : [\*|["l","m","o","p","q"]|\*|  
["b","d","e","f","g","h","i","j","k","n"]
- ["a","b"] : [\*|["d","k","l","m","n","o","p","q"]|\*|  
["c","e","f"] | ["g","h","i","j"]

The set  $M^*$  is found to be :

```

["l"] : [*|["p"]|*| ["a","b","d","e","f","g","h","i","j","k","m","n","o"] | ["
["m"] : [*|["o"]|*| ["a","b","d","e","f","g","h","i","j","k","l","p","q"] | ["
["d"] : [*|["b","n","p"]|*| ["c","e","f","g","k","q"] | ["m","o"] |
        ["a","h","l"] | ["i","j"]
["c"] : [*|["a","l","m","o","p"]|*| ["d","g","h","i","j","k","n"] |
        ["q"] | ["e"] | ["b","f"]
["a","d"] : [*|["b","l","n","p"]|*| ["c","e","f","g","i","j","k","q"] |
        ["h"]
["h"] : [*|["a","b","d","k","l","m","n","o","p","q"]|*| ["c","e","f"] |
        ["g"] | ["i","j"]
["f"] : [*|["a","b","d","k","l","m","n","o","p","q"]|*| ["g"] |
        ["h"] | ["i","j"] | ["c","e"]
["k"] : [*|["l","m","o","p"]|*| ["h"] | ["i","j"] | ["f"] |
        ["g"] | ["e"] | ["q"] | ["c"] | ["a","b","d","n"]
["g"] : [*|["b","d","k","l","m","n","o","p"]|*| ["h"] | ["i","j"] |
        ["c","e"] | ["a","f","q"]
["b","d"] : [*|["m","n","o","p"]|*
["a","c"] : [*|["l","m","o","p","q"]|*
["a","b"] : [*|["d","k","l","m","n","o","p","q"]|*| ["f"] |
        ["c","e"] | ["g"] | ["h"] | ["i","j"]

```

Relation :  $r6\_a$  ["l","p"] in 4NF

KEY : ["l"]

Relation :  $r6\_c$  ["l","q"] in 4NF

KEY : an all-key relation

Relation :  $r6\_b\_a$  ["m","o"] in 4NF

KEY : ["m"]

Relation :  $r6\_b\_c$  ["m","n"] in 4NF

KEY : an all-key relation

Relation :  $r6\_b\_b\_a$  ["b","d"] in 4NF

KEY : ["d"]

Relation :  $r6\_b\_b\_b$  ["d","e","f","g","k"] in 4NF

KEY : an all-key relation

Relation : r6\_\_b\_b\_c ["d","m"] in 4NF

KEY : an all-key relation

Relation : r6\_\_b\_b\_e ["d","i","j"] in 4NF

KEY : an all-key relation

Relation : r6\_\_b\_b\_d\_a ["a","d","l"] in 4NF

KEY : ["a","d"]

Relation : r6\_\_b\_b\_d\_b ["a","d","h"] in 4NF

KEY : an all-key relation

Relation : r6\_\_d\_a ["c","l"] in 4NF

KEY : ["c"]

## REFERENCES

- [1]. Kent, William. "A simple guide to five normal forms in Relational Data Base theory". Communications of the ACM, Feb. 1983, vol.26, No.2, pp 120-125.
- [2]. Ram, Sudha and Curran, S.M. "An automated tool for relational data base design". Information systems Vol.14, No. 3, pp 247-259, 1989.
- [3]. Bernstein, Philip A. "Synthesizing Third Normal Form relations from Functional Dependencies". ACM Transactions on Database Systems, Vol.1, No.4, December 1976, pp 277-298.
- [4]. Ceri, S. and Gottlob, G. "Normalization of relations and Prolog". Communications of the ACM, June 1986, Vol.29, No.6, pp 524-544.
- [5]. Beeri, Catriel; Bernstein, Philip A. and Goodman, Nathan. "A sophisticate's introduction to database normalization theory". Proceedings of the 4th International Conference on Very Large Databases (West Berlin). 1978, pp 113-124.
- [6]. Tanaka, Yuzuru. "Logical design of a relational schema and integrity of a Data base". Data base architecture, edited by Bracchi/Nijssen, North-Holland Publishing Company, 1979.
- [7]. Martin, James. "Principles of Data-Base Management". Prentice Hall of India, 1982.

- [8]. Date, C.J. "An introduction to Database systems". Third ed. 1985, Addison-wesley/Narosa.
- [9]. Townsend, Carl. "Introduction to Turbo Prolog". BPB Publication, 1988.
- [10]. Nath, Sanjeeva. "Turbo Prolog, features for programmers". Galgotia Publications, 1988.
- [11]. WordStar Professional , release 4.
- [12]. Borland's "Turbo Prolog 2.0 User's guide".
- [13]. Borland's "Turbo Prolog 2.0 Reference guide".