To the Graduate Council:

I am submitting herewith a thesis written by B. Timothy Rhyne entitled "The Application of Directed Graphs to Data Modeling and Normalization." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Maria Zemankova

Maria Zemankova, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Vice Provost
and Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in her absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature B. Timothy Rhyne

Date July 24, 1988

THE APPLICATION OF DIRECTED GRAPHS TO DATA MODELING AND NORMALIZATION

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

B. Timothy Rhyne

August 1988

DEDICATION

This thesis is dedicated with love to my wife Marsha. Thanks, Tim.

ACKNOWLEDGMENTS

I would like to express gratitude to all the members of my committee.

Dr. Bill McClain, the first member of my committee, has been my instructor for many courses and displays an enthusiasm for teaching that is rare on the college level. Without the opportunity to take classes at the Oak Ridge Resident Graduate Center, including those in which Bill was the instructor, this degree would not be possible.

Dr. David Straight, whose so-called "creative test questions" always challenged me to apply material learned in class to new situations, rounded out my committee and brought to it an expertise in data structures and algorithms of which I hope that I have acquired a small fraction.

I would like to particularly thank my Chairperson, Dr. Maria Zemankova. Maria has faithfully and promptly read whatever crude drafts that I have given to her. Not only is she the best data base person whom I know in Tennessee, but she also made what has been to me a noticeable effort to communicate with me on my terms. I am not sure how much exposure Maria had previously with the topic of data modeling, which is a critical topic to the understanding of the thrust of the thesis; but if she did not previously have a great deal of exposure, she has obviously gained an understanding of the technique over the course of the

directing this thesis. Her guidance and suggestions have proved invaluable.

I appreciate the assistance of the University's Thesis Consultant--Ann Lacava. She not only knows the rules and tells you to follow them; she offers advice as to how to most easily change what you have into an acceptable form.

I would also like to thank Mr. Terry Futrell. Terry was formerly Data Resource Administrator at the Department of Energy's Y-12 Plant. I worked for Terry briefly, and it was he who introduced me to both my primary interests in computing--personal computers and data modeling. Terry has given me advice concerning the preferred course of resolving various problems which arose in the normalization process. His suggestions have proven more helpful than any book I have read on the subject.

Last, but not least, I would like to thank my wife, Marsha. Marsha has encouraged me, typed and edited the manuscript, and tolerated me when I have worked at home weekend after weekend in order to complete this thesis.

ABSTRACT

As the ratio of personal computers (PCs) to mini-computers and mainframes grows, more and more data processing applications will be run on PCs. Many of these applications will be designed using data and transaction modeling tools and normalization software. Data modeling and normalization are techniques not uncommon in many larger business enterprises today, but it is rare to find designers and programmers using data modeling tools for microcomputer applications. This is due to the lack of awareness of the benefits of data modeling and normalization and to the unavailability of automated tools for assisting the data base developer in the microcomputer environment.

This thesis describes an accompanying microcomputer software product, named the Data Modeling and Normalization System that provides such a tool for storing and updating data model user views, normalizing them, and designing a relational data base from the normalized scheme. The Data Modeling and Normalization System's data dictionary, data dependencies, and relational data base design data are stored in popular dBASE III data file format.

The normalization component of the Data Modeling and Normalization System is written in C and uses a directed graph data structure to represent the semantic data models used in the normalization process. Each step in the normalization process is described in Chapter VII. Such an

in-depth description of how an algorithm for computerized normalization works has previously been unavailable because normalization software systems currently available on mainframe computers are proprietary.

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES

LIST OF FIGURES

LIST OF FIGURES

I. OVERVIEW

Data base design tools have largely been ignored by the microcomputer community. Because "large-scale" data base applications have been beyond the capability of early personal computers, such tools were not essential. Today's state-of-the-art personal computers, however, have the power of minicomputers of five years ago. New chips like the Intel 80386 allow larger, more complex tasks to be performed on a personal computer than previously possible. The need to do more complex jobs in the Data Base Management System (DBMS) area has brought with it the need for better tools to assist the data base developer. This thesis presents such a new tool--the Data Modeling and Normalization System (DMNS).

In order to understand what data base design tools do, certain background material must be presented. In Section II an introduction to data base design tools is presented.

In Section III a method of data organization known as the relational data model is described.

In Section IV a technique for rearranging data to minimize unnecessary redundancy and possible inconsistencies, known as normalization, is discussed.

In Section V the two prior concepts are incorporated into a discussion of a discipline known as data modeling. The Data Modeling and Normalization System will facilitate the data modeling process and automate normalization.

Section VI examines the functions of the DMNS and briefly describes how the software is used. A more detailed explanation of the use of the DMNS is presented in a separate document--the Data Modeling and Normalization System User's Guide.[1]

Section VII introduces directed graphs--the data structure used to represent the data model during normalization--and provides a step-by-step discussion of the algorithms used to normalize the data model to fourth normal form. The section concludes with discussions of how the normalization algorithms act on specific instances of data previously in lower levels of normalization to transform those instances to fourth normal form.

Section VIII outlines the use of the DMNS on a sample moderate scale data modeling project. The project included was presented in a popular book dealing with data modeling. The differences in the two competing designs are discussed.

Section IX is the conclusion. The results of the development of the DMNS are discussed as are possible future avenues for research.

---

[1]B. Timothy Rhyne, User's Guide to the Data Modeling and Normalization System (unpublished manuscript).

II. INTRODUCTION TO DATA BASE DESIGN TOOLS

Data Base System Architecture

Before discussing specific data base design tools, it is worthwhile first to explain how a DBMS is organized and what the roles of data base design tools are in the overall development and implementation process. A data base is a computerized collection of data. Some authors use the spellings--database or data-base for the same concept. The DBMS is the software which allows data to be entered into and retrieved from the data base. A DBMS is a multipurpose software system. It is not directed toward a particular application when it is purchased from the DBMS vendor. In most cases, the DBMS is programmable so that specific applications may be custom tailored for the people who will perform the tasks required by the application. It is the development of these custom applications with which the Data Modeling and Normalization System is to assist.

Data bases may be thought of in three distinct ways. These ways are commonly referred to as the three-level data base architecture. The book Database System Concepts, by Korth and Silberschatz,[2] presents the following definitions of each of the three data base architecture levels.

> **Physical level.** This is the lowest level of abstraction, at which one describes how the data

---

[2]Henry F. Korth and Abraham Silberschatz, Database System Concepts (New York: McGraw Hill, 1986), p. 4.

are actually stored. At this level, complex, low-level data structures are described in detail.

Conceptual level. This is the next higher level of abstraction at which one describes what data are actually stored in the database, and the relationships that exist among data. This level describes the entire database in terms of a small number of relatively simple structures. Although the implementation of the simple structures of the conceptual level may involve complex physical-level structures, the user of the conceptual level need not be aware of this. The conceptual level of abstraction is used by database administrators, who must decide what information is to be kept in the database.

View level. This is the highest level of abstraction at which one describes only part of the entire database. Despite the use of simpler structures at the conceptual level, there remains a form of complexity resulting from the large size of the database. Many users of the database system will not be concerned with all of this information. Instead, such users need only a part of the database. To simplify the interaction of such users with the system, the view level of abstraction is defined. There may be many views provided by the system for the same database.

A data base management system is important largely because it provides the programmer with a property known as **data independence.** Data independence means that an application program does not need to know the record descriptions of the files which they need to access. Instead, the application program requests data by name from the data base management system and the DBMS retrieves the data and returns it to the application program. Some forms of modifications to the record structure of data base files, such as adding a new data field, can then be made without modifying the application programs which extract data from

them. Similarly, application programs can be changed without rearranging the data base. Data independence is a great time and cost saver to the application programmer.

A DBMS typically has embedded within it two sublanguages. These are the Data Definition Language (DDL) and Data Manipulation Language (DML). These languages are usually accessible to the user through an interactive dialogue manager within the DMNS or as calls from applications programming languages such as C, FORTRAN, and COBOL. These languages convert user requests from the user's conceptual view of the data in the data base to the operating system calls which do the actual data retrieval from disk. The user does not need to know where on the disk the data is stored, how it is organized, or how it is located. Data base design tools work on the view and conceptual levels of the DBMS architecture.

Types of Data Base Design Tools

The most common type of data base development tool for microcomputers is the so-called "application generator." Application generators create empty data base files and generate source code that allows users of the system to manipulate the data which will be stored in these files. Although they are often thought of as data base design tools, they are not truly such tools because they do not provide the designer with help in organizing the data base

files in an efficient arrangement. They only implement the design which the designer has specified.

Data modeling systems provide the data base designer with the tool needed to create an efficient arrangement of files for a data base application. They do not create the data base files themselves or write application programs.

Application Generators

Application generators typically work by first requiring the data base designer to input a data base design into the system by either describing the design in a syntactic form or by laying out screen formats to allow data to be entered into the data base. If the entry of the data base design is by laying out screen formats, then the application generator will create a data base schema matching the screens. This data base design will indicate which pieces of data (called data fields) will appear in which files, the type of data in each field (numeric, text, date, etc.), and sometimes other information such as valid acceptable ranges.

The application generator will also provide programs which display the data on the screen and allow common data manipulation functions, such as add, edit, delete, list, sort, etc. for manipulating the application data to be stored in the data base file associated with the screen.

It is the task of the data base designer, with no help from the application generator software, to decide which fields should be included in each file. Some such systems do not understand that data fields in two different screens with the same name are intended to refer to the same field in the data base.

For example, the application generator of dBASE III by Ashton-Tate works reasonably well for the novice computer user who wants to create a simple system, such as a mailing list, but the application generator quickly fails in more complex uses. This is due both to inadequacies in the software's capability and to inadequate data base designs which the application generator created from the screen formats prepared by the user.

Anyone using Ashton-Tate's application generator must function within the limitations of the software, but can improve the data base schema design by himself. Better data base design is achieved though a process known as **normalization.** Normalization is a method by which a data analyst creates a data base design with no unnecessary redundancy in the data, and which allows addition, revision, and deletion of data with a minimum of difficulty. The drawbacks of performing normalization by hand are its tendency to be both time consuming and error prone. Normalization will be discussed in further detail later.

The screens that users create for application generators are usually designed with one of two things in mind. The user either constructs multiple screens based on what he/she thinks is an intuitively logical grouping (which often resembles a normalized design), or he/she constructs a screen to resemble some business transaction which the user needs to perform. Often the two styles become mixed in usage and some screens are prepared with logical organization in mind and other screens are done around business transactions. Unfortunately, screen-driven application generators expect to work against normalized relations. Generally they do not function properly when given a task which requires that data from multiple tables be extracted or updated. The result of this method of operation is that there are a large number of real business transactions which these systems are not capable of performing. To handle complex systems, the need for processing business transactions and the need for creating stable third normal form data base schemas must both be satisfied but in different ways.

Data Modeling Software

Data modeling software allows for documentation of data dependencies, performs normalization, and helps with tracking where data enters and leaves a data base. Most of

these systems, such as "Designmanager" from Management Systems and Programming run only on mainframe computers.

A few data modeling systems are becoming available for microcomputers. One of these is called "Data Designer II," ((C) KnowledgeWare). Like other commercial data modeling software, it must interact with a mainframe to perform normalization.

The Data Modeling and Normalization System, which is a part of this thesis and whose normalization process is described herein, allows user views of data transactions to be represented in a data base and then merged and normalized. Again, normalization can be performed manually, but it is a tedious process.

III. THE RELATIONAL DATA MODEL

An Introduction to the Relational Data Model

Data in a relational data model is organized in special two-dimensional matrices called **relations.** The rows of a relation are called **tuples,** and the columns are known as **attributes.** Consider the relation in Figure 1:

PAYROLL		
DEPARTMENT	EMPLOYEE	WAGE
Shipping	J.E. Mills	5.40
Purchasing	K.W. Wright	7.25
Finance	O.C. Jackson	
Shipping	T.I. Howard	4.90

Figure 1. Sample Relation PAYROLL.

In relation PAYROLL, the column labels "DEPARTMENT," "EMPLOYEE," and "WAGE" are attribute names. The row containing the attribute values Purchasing, K.W. Wright, and 7.25 forms a tuple.

The number of columns in a relation is its **arity.** The number of rows is its **cardinality.** The arity of the relation PAYROLL is 3. The cardinality of relation PAYROLL is 4.

The structure of the PAYROLL relation can be denoted pictorially by the connected boxes shown in Figure 2. This is called the **relation schema** for the relation.

PAYROLL

DEPARTMENT	EMPLOYEE	WAGE

Figure 2. Relational schema for Relation PAYROLL.

Each attribute in a relation has an allowable range, or set, of possible values known as its **domain.** For every tuple, each attribute value must be a member of that attribute's domain. For example, the domains for the attributes in relation PAYROLL might be the set of all departments in a company for DEPARTMENT, the set of all employee names for EMPLOYEE, and all positive real numbers for WAGE. Since a relation consists of one or more attributes, the set of possible tuple values of relation PAYROLL, called the **intension** of the relation, is determined by finding the Cartesian product of the domains of each of its attributes. Considering again the relation PAYROLL, if the domain of attribute DEPARTMENT is represented by D_1, the domain of the attribute EMPLOYEE is D_2, and the domain of the attribute WAGE is D_3, then the domain of the relation formed by joining the attributes DEPARTMENT, EMPLOYEE, and WAGE must be $D_1 \times D_2 \times D_3$.

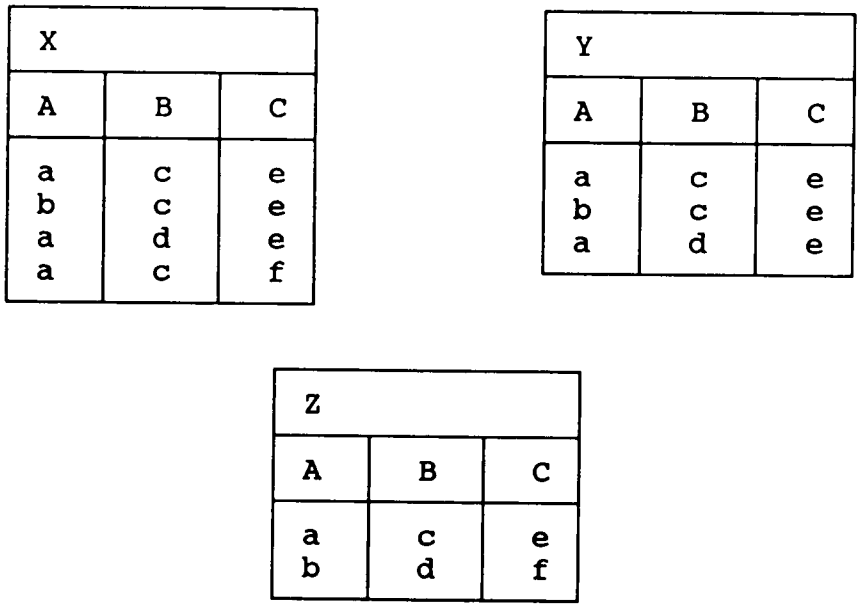
At any instance in time the actual data in the tuples of a relation form a set called the **extension.** The extension must always be a subset of the intension.

The domain of an attribute may be allowed to include the null value when the value of the attribute is not known. For example, in Figure 1 the WAGE of O.C. Jackson was unknown, so no value has been entered for WAGE in that tuple.

Keys

In a relation, some attributes have special roles. These roles are called **keys**. A key can be one attribute, all attributes, or a subset of attributes. Keys of more than a single attribute are called **composed keys**. Although, semantically, the ordering of attributes in a composed key is important, in the relational model the order of attributes in a composed key and in the relation is not significant. Any of the types of keys discussed in the remainder of this section can be either single attribute keys or composed keys.

The first type of key is called the **primary key** of a relation. Primary key values cannot be duplicated in more than one tuple of the relation. Every relation has one and only one primary key. Consider the two relations shown in Figure 3:



X		
A	B	C
a	c	e
b	c	e
a	d	e
a	c	f

Y		
A	B	C
a	c	e
b	c	e
a	d	e

Z		
A	B	C
a	c	e
b	d	f

Figure 3. Primary keys.

In Relation X none of the attributes can be an individual primary key because each has at least one duplicate entry. A and B cannot be a composed primary key because the first and last tuples have a duplicate combination A and B values. B and C cannot be a composed primary key because the first and second tuples have a duplicate combination of B and C values. A and C cannot be a composed primary key because the first and third tuples have a duplicate combination of A and C values. Therefore, the only possible primary key for Relation X is the combination of attributes A, B, and C.

In Relation Y none of the three attributes can be individual primary keys because each has at least one

duplicate entry. A and B can be the composed primary key because they have no matching attribute pairs across tuples. No other combination of two attributes in Relation Y could qualify as a composed primary key. The combination of all three attributes could also be the primary key.

In the relational schema, primary keys are shown in bold print. If the primary key is composed, then the components of the primary key are connected by a line beneath the schema. The relational schemas for Relations X and Y (assuming that A and B make up the composed key of relation Y) are shown in Figure 4:

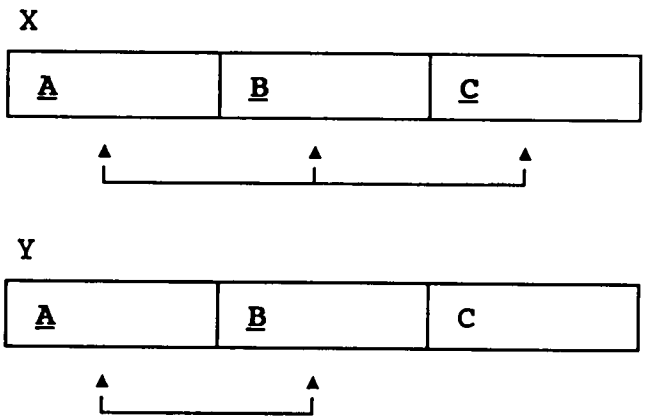


Figure 4. Composed primary keys in a relational schema.

Finding primary keys in a relation just by looking at data in the table is often not possible, as illustrated previously in the analysis of relation Y. Relation Z is a more extreme case of a relation in which many combinations of attributes can possibly be the primary key based only on

the data presented in the table itself (the extension). Relation Z has no duplication in any column. Is A the primary key? Could it be B or C? Could the primary key be A and B combined; B and C combined; A and C combined; or A, B, and C combined? Which is the correct primary key? The truth is that the primary key of relation Z can be any of the above choices, but will be only one of these choices.

Since relations by themselves have no interpretation, it is not possible to tell just by looking at data what the primary key is unless the data narrows the options to only one possible choice. The reason that the extension of a relation is not a reliable indicator of what is the primary key is for the relation, is that there is no proof that the next tuple added to the relation does not form a duplication of an attribute or set of attributes which was previously unique. Unless an attribute or set of attributes will be unique <u>at all times</u> it cannot be a primary key. When the topic of data modeling is discussed later, it will be shown that primary keys are identified <u>by data interpretation</u> and not <u>by data values</u>. Data can show what cannot be a primary key, but it is rarely proof of what the primary key for a relation actually is.

Tuples in a relation are often ordered by ascending or descending order of the primary key in each tuple. This often has semantic value but is not a requirement of the relational model.

A **foreign key** for a relation is a set of attributes which makes up the primary key in some other relation. Consider the two relations of Figure 5.

X

DEPTNO	DEPTNAME
83	Woodworking
62	Sewing
19	Administration

Y

EMPLOYEE	DEPTNO	SALARY
Jones	62	2015
Smith	83	2600
Rice	19	4000
Martin	83	2305
Date	62	2160
Codd	83	1900

Figure 5. Example of a foreign key.

The relational schema for relations X and Y are shown in Figure 6:

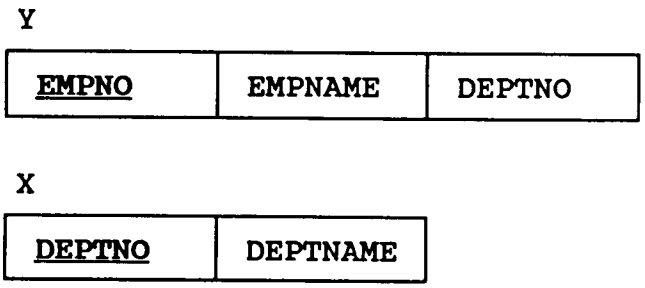


Figure 6. Relational schema for relations with a foreign key.

DEPTNO is a foreign key to Relation Y because it is the primary key for Relation X.

The Relation X above is also a good example of another concept. Assuming that Relation X is a list of departments in a company, then both the DEPTNO and DEPTNAME will always be unique in the company. This means that for Relation X, DEPTNAME would also have been an acceptable choice for the primary key. Keys which can serve as primary keys are called **candidate keys.** The choice of which candidate key to use as the primary key is up to the data base administrator. Candidate keys which are not chosen as the primary key are called **alternate keys.** Candidate keys are printed in bold and underlined, but separate individual candidate keys are not joined by a line underneath when their relational schema is drawn. The relational schema for Relation X from Figure 6 can be redrawn if DEPTNAME and DEPTNO are both candidate keys as below:

X

DEPTNO	**DEPTNAME**

Entities and Relationships

Data bases store information about people, places, things, concepts, etc. The user's understanding of such items and how they are interrelated is called the **conceptual schema.** Two common models for representation of the conceptual schema are the "entity-relationship model",

originated by P.P.S. Chen,[3] and "data modeling", popularized by James Martin.[4] Data modeling is a more detailed methodology and is discussed in detail in Section V. Both models use some similar terminology as described below.

A "real world" object, such as a person, is called an **entity**. A group of like entities, such as all persons who are federal employees, is known as an **entity set**. Every entity in an entity set can be distinguished from every other entity in that set.

In practice, when a data base is created, a relation stores data about all the members of an entity set. Each entity in the entity set has a corresponding tuple in the relation. Each entity of that entity set is distinguished from other entities by the value of the primary key of its associated tuple in the relation. Recall that primary keys are never duplicated, so all entities are uniquely distinguishable. For example, to identify members of the entity "all employees of Company XYZ," the attribute social security number (SSNO) could be used as the primary key of the relation PAYROLL as in Figure 7.

---

[3]P. P. Chen, "The Entity-Relationship Model: Toward a Unified View of Data," ACM Transactions on Database Systems, 1, No. 1, pp. 9-36.

[4]James Martin, An End-User's Guide to Data Base (Englewood Cliffs, N.J.: Prentiss-Hall, 1981), pp. 49-74.

PAYROLL

SSNO	NAME	SALARY	DEDUCTIONS

Figure 7. Relational schema for Relations PAYROLL.

Entity sets may be related to each other in some way. This may be in some form of ownership or other association. These associations are called **relationships.** For example, if a person owns several dogs, then there is said to be a one-to-many relationship between person and dogs. Relationships are enforced in a data base by using foreign keys. In this case the value of the foreign key in a tuple of the dog relation indicates to which person the dog belongs.

The use of primary keys to represent entities in an entity set gives rise to the need for two rules. The first rule is known as the entity integrity rule.

Entity Integrity: For every tuple in a relation, no attributes composing the primary key may be null.

The entity integrity rule is needed to force every tuple of a relation to represent an entity in an entity set. The second rule on keys of a relation deals with foreign keys and is called the referential integrity rule.

Referential Integrity: If an attribute is a foreign key in a relation and its value is not null, then its value

must be present in a tuple of the relation for which this attribute is the primary key.

The referential integrity rule prevents a foreign key from indicating an entity which does not exist in its entity set.

Relational Algebra

A relation is a useful way of grouping data; but in order to retrieve the desired data, there must be techniques for selecting, combining, and condensing the data needed to answer a user's specific questions. This technique is provided by the data manipulation language.

For relational data bases, two basic types of such systems exist--relational algebra and relational calculus. Relational algebra is a procedural language, and will be examined in greater detail in this section; that is, the user must define exactly what must be done and in what order. Relational calculus is nonprocedural. The user can specify what is needed and leave the data extraction procedure up to the query processing mechanism of the data manipulation language. This mechanism decides how to extract from the data base the information which the user has requested. The Structured English Query Language (SQL), which is probably the most popular query language in use for relational systems, is based on relational calculus.

Although relational calculus has greater use, this thesis will concentrate on relational algebra for two reasons. First, relational algebra, which is based on simple set theory, is easier for most people to understand. Second, it seems that if data model user views (described in the next section) are to be expanded into transaction views for the next generation of application generators, it will be by expanding relational algebra syntax.

Edgar F. Codd formulated the definition of relational algebra in 1972.[5] Several syntax variations have been published since the original work. This thesis will use the BNF syntax described in Chris Date.[6] The five "primitive" set operations are selection, projection, union, difference, and Cartesian product. These five primitive operations are said to be "complete" because any feasible data base query can be answered with a combination of the primitive operations alone. Conversely, if no query to answer the question can be formulated with the primitive operations, the question cannot be answered using any other technique.

Four additional operations, each of which can be composed from the primitive operations, are also introduced for convenience. These operations are intersection, join,

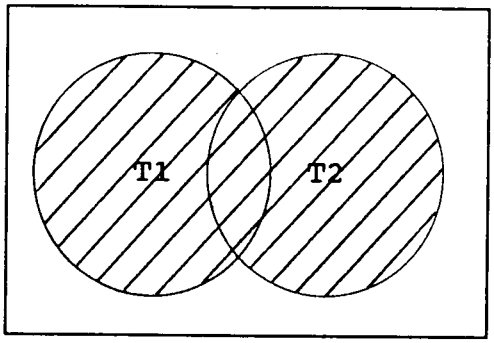
---

[5]Edgar F. Codd, "Further Normalization of the Data Base Relational Model," Data Base Systems, 6 (1972), pp. 33-64.

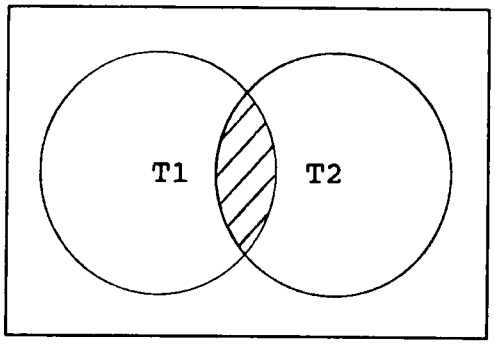
[6]C. J. Date, An Introduction to Database Systems (Reading, Mass.: Addison-Wesley, 1981) pp. 203-211.

natural join, and division. Each of these operations will now be examined in greater detail.

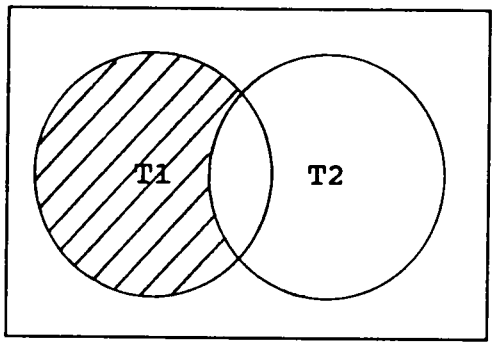
To perform union, intersection, and difference operations, the two relations must be **union-compatible.** For two relations to be union-compatible, they must have the same number of columns (the same arity), the same names for each corresponding column, and the same domains. Many people are familiar with union, intersection, and difference as set operations. In Figure 8(A) two sets, named T1 and T2, form a union. In Figure 8(B) the intersection of these two sets is shown. Figures 8(C) and 8(D) show the differences T1-T2 and T2-T1, respectively.



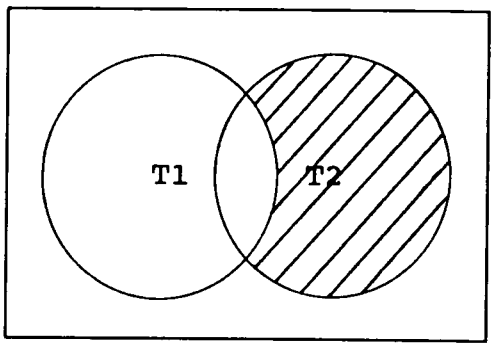
(A) T1 UNION T2



(B) T1 INTERSECTION T2



(C) T1 MINUS T2



(D) T2 MINUS T1

Figure 8. Venn diagrams of typical set operations.

These operations in relational algebra operate very similarly. Consider the two relations in Figure 9. Both T1 and T2 have arities of 2, have the same names for columns, and may be assumed to have the same domains for corresponding columns. Therefore, Relations T1 and T2 are union-compatible.

T1	
A1	A2
Jack	4
Bill	5
Marsha	2
Jack	8

T2	
A1	A2
Marsha	2
Jack	4
Marsha	8
Jackie	5

Figure 9. Example Relations T1 and T2.

A **union** is the combination of two relations with duplicate tuples eliminated. For T1 UNION T2, the resulting relation is shown in Figure 10. Relation T1 UNION T2 contains all unique tuples from Relations T1 and T2 and has only one occurrence each for the duplicated tuples--(Jack, 4) and (Marsha, 2).

T1 UNION T2	
A1	A2
Jack	4
Bill	5
Marsha	2
Jack	8
Marsha	8
Jackie	5

Figure 10. UNION of Relations T1 and T2.

The **intersection** of two relations is formed by taking the tuples of one relation that are also in the second

relation. Figure 11 shows the result of the intersection operation T1 INTERSECTION T2.

T1 INTERSECTION T2	
A1	A2
Jack Marsha	4 2

Figure 11. Example of the INTERSECTION operation.

The difference of two relations is formed by taking the tuples of the first relation which are not in the second relation. The resulting relations for the two differences between the Relations T1 and T2 (T1 MINUS T2 and T2 MINUS T1) are shown in Figure 12.

T1 MINUS T2	
A1	A2
Bill Jack	5 8

T2 MINUS T1	
A1	A2
Marsha Jackie	8 5

Figure 12. Example of the difference operation.

The Cartesian product of two relations, R1 and R2, is a relation with the attributes of R2 concatenated to the attributes of R1, and the tuples of the resulting relation equalling all possible combinations of the tuples from R1

and R2. The Cartesian product of two example Relations, named R1 and R2, is shown in Figure 13:

R1		
NAME	AGE	SEX
John Nancy	30 21	M F

R2	
PETNAME	SPECIES
Spot Fifi Blackie	Dog Cat Horse

R1 TIMES R2				
NAME	AGE	SEX	PETNAME	SPECIES
John	30	M	Spot	Dog
John	30	M	Fifi	Cat
John	30	M	Blackie	Horse
Nancy	21	F	Spot	Dog
Nancy	21	F	Fifi	Cat
Nancy	21	F	Blackie	Horse

Figure 13. Example of the Cartesian product operation.

The **selection** operation applied to an existing relation yields a subset of the tuples of the original relation. The criteria for choosing which tuples will be included in the resulting relation are expressed in terms of attribute name(s) and their selected values, combined by boolean connectives. The selection operation in relational algebra is analogous to the set operation of taking a subset.

Consider the following two examples: one from set theory and one from relational algebra. The Venn diagram of

Figure 14 is an example of a subset operation in set theory. Only particular elements of Set R are chosen to be in the subset of R, which is named R'.

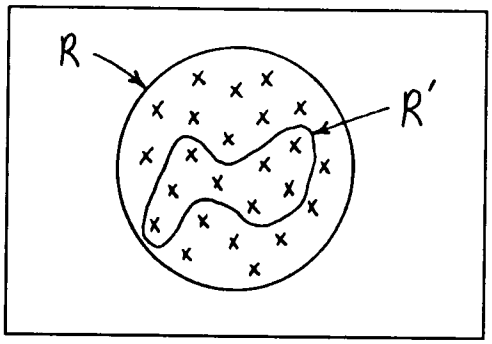


Figure 14. Venn diagram of a Set R and its subset R'.

Similarly in relational algebra, for some Relation R, another Relation R' is the subset of the tuples of R meeting a particular criteria. This can be thought of as a horizontal subset of a relation. In Figure 15 Relation R is a table of salespeople, their territory, and their gross sales. The Relation R' is a subset of a Relation R where R' contains only the tuples from R that meet the condition that the TERRITORY column must have the value of "Northwest."

R		
SALESPERSON	TERRITORY	GROSS
Johnson	Northwest	32000
Williams	Southeast	44000
Sanders	Midwest	37000
James	Northeast	29000
Miller	Northwest	35000
Smith	Southwest	52000
Raines	Southwest	37000
Alton	Northeast	33000

r = (R Where Territory = "Northwest")		
SALESPERSON	TERRITORY	GROSS
Johnson	Northwest	32000
Miller	Northwest	35000

Figure 15. Example of the selection operation.

The **projection** operation is a vertical subset of a relation with duplicate tuples eliminated. A projection can be made for any number of attributes in the relation and can be used to reorder the attributes in the resulting table. Consider the example in Figure 16.

PARTS			
PNO	PNAME	SNO	QOO
463A	Washer	S4	1000
628E	Bolt	S2	1500
463A	Washer	S4	500
628E	Bolt	S5	1000
183C	Screw	S1	1000
628E	Bolt	S2	500

P = PARTS[SNO,PNO,PNAME]		
SNO	PNO	PNAME
S4	463A	Washer
S2	628E	Bolt
S5	628E	Bolt
S1	183C	Screw

Figure 16. Example of the projection operation.

In Figure 16 the projection operation has eliminated the QOO column and reordered the other three columns in the output relation. The third and sixth tuples from relation PARTS are not in P because they would have been duplicates of tuples already in P.

A **join** operation allows tuples from two relations to be concatenated with each other if some logical condition specified by the user exists between attributes of the two original relations. The **natural join** (NJOIN) operation is a special case of the join. For a natural join, this condition must be an equality of attribute values between commonly named attributes from both relations. These common attributes appear only once in the output relation. An example of a natural join operation is shown in Figure 17.

X		
A	B	C
P4	K	4
P6	N	2
P1	L	3
P8	N	2
P2	M	6

Y		
B	C	D
N	2	JAN
L	3	AUG
M	5	MAR
L	3	OCT
K	4	MAR

X JOIN Y			
A	B	C	D
P4	K	4	MAR
P6	N	2	JAN
P1	L	3	AUG
P1	L	3	OCT
P8	N	2	JAN

Figure 17. Example of the natural join operation.

The last relational algebra operation considered here is the division operation. Given two relations, X and Y, where arity(X) > arity(Y), the quotient of X DIVIDEBY Y is formed by finding matching values of the entire Y table in liked named attributes in X and copying the remaining attributes of X to the output relation if the tuples agree in all the nonmatched attributes. Figure 18 shows an example of a division operation between two tables.

X		
NAME	CITY	ITEM
HOWARD	ATLANTA	CPU
MILLER	ATHENS	DISK
LOGAN	ATHENS	TAPE
LOGAN	ATLANTA	TAPE
TAFT	SAVANNAH	CPU
MILLER	ATLANTA	DISK
HOWARD	ATHENS	DISK

Y
CITY
ATHENS
ATLANTA

X / Y	
NAME	ITEM
MILLER	DISK
LOGAN	TAPE

Figure 18. Example of the division operation.

Before ending the topic of relational algebra, two additional points should be made. The first is that although many of the DBMSes being sold today claim to be relational systems, few actually are. To be worthy of the title "relational," a system must have all the primitive relational operations (or the alternative definition of primitive operations--union, difference, Cartesian product, projection, and selection). Ideally, a relational system will have all the operations described here for the convenience of the user.

The second point is that in addition to relational operations, relational systems usually have other convenient operations for maintaining the data base and performing statistical calculations on the data. Examples of maintenance operations would be **add** and **delete** for adding and removing tuples from the data base. Examples of statistical operators would be **sum**, **count**, and **average** for calculating values from a selection of tuples. Operators such as these are very useful in many practical situations.

IV. NORMALIZATION

Data Base Design Problems

When a data base designer sets out to create a data base to be used in an application, there will be many possible designs which can be used to achieve that goal. How does the designer select the best design? Perhaps the best way to answer this question is by examining some of the features of a poor design. From Figure 19 consider the relation PARTS.

PARTS				
PARTNO	PNAME	SUPPNO	SUPPNAME	PRICE
38A	Nut	13	ACME	0.12
38A	Nut	21	KMART	0.12
69D	Bolt	30	DIME STORE	0.40
43C	Screw	13	ACME	0.25
43C	Screw	21	KMART	0.25

PARTS

PARTNO	PNAME	**SUPPNO**	SUPPNAME	PRICE

Figure 19. Example design of a parts data base and its relational schema.

The intention of this data base is to keep track of the parts needed for a construction project, the part's supplier, and what each supplier charges for the part. The

design in Figure 19 is certainly the simplest that can be derived in that the design goals can be achieved with only one relation. Is this the best design though?

The first design consideration is: is this arrangement of fields in a single relation space efficient? Clearly, it is not space efficient. Even though we know that for every supplier number there is a single unique supplier name, the name of the supplier is repeated for every part which the supplier provides. Doing this wastes storage space. A condition such as this is called **data redundancy**. To save space, data redundancy must be minimized. The same problem occurs again in the PARTS relation with PARTNO and PNAME. Minimal data redundancy will be a critical theme in further data base design considerations as it also avoids the three other problems described below.

The relation above may be accurate and up to date for now, but what will happen as changes are made in the data base? With the current design, the potential exists for inconsistencies to occur and information to be lost.

An example of one problem which could arise is that someone could update the name of Part 38A in the first tuple but not update it in the second. The next person who needs data from the data base will have no way of knowing which part name is correct. This is called the **update anomaly**.

Next, suppose that a new part is needed for the project but that a supplier has not yet been identified for

the part. Since PARTNO and SUPPNO form the composed key for relation SP, a tuple for the new part cannot be added with a null SUPPNO without violating the entity integrity rule. This is known as the **insertion anomaly.**

Finally, what would happen if due to an engineering change, Part 69D is no longer needed for the project and its tuple is deleted? If the tuple for 69D is deleted, the name of Supplier 30 is also lost. This unintentional loss of information is called the **deletion anomaly.**

How can these pitfalls be avoided? An alternate design shown in Figure 20 does not suffer from the problems above.

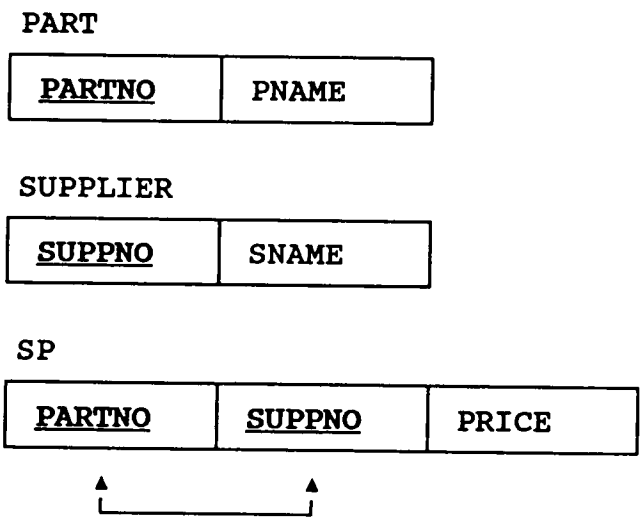


Figure 20. Alternative data base design for parts problem.

With the design of Figure 20, the names of parts and suppliers are listed only once, thereby eliminating the data redundancy for these fields. This saves storage space and also eliminates the three anomalies listed previously.

The gain in space economy and update consistency provided by the alternate design is not free, however. To list the names of suppliers, names of parts, and the price of each part charged by a supplier requires accessing three different files. The Data Base Administrator (DBA) must weigh the relative cost of these factors. A design such as the second one presented here, where unnecessary redundancy and anomalies are removed, is a good beginning target. Once developed, this design can then be refined to meet retrieval requirements as needed. How does one proceed to design a data base where the redundancy is reduced and the anomalies are eliminated? This is done through normalization. To begin the discussion of normalization, an understanding of data dependencies is necessary.

Data Dependencies

Attributes often have some "real world" inter-relationships with each other. These inter-relationships can be modeled by dependencies. There are three types of dependencies--functional, full functional, and multivalue.

Given two sets of attributes, A and B, B is said to be **functionally dependent** (FD) on A if each value of A has one and only one value of B that can possibly be associated with it. For example, consider the relational schema in Figure 21:

EMPLOYEES

EMPNO	NAME	DEPTNO

Figure 21. Sample Relation EMPLOYEES.

In the relation of Figure 21, NAME is functionally dependent on EMPNO. Other terminology sometimes used is to say that EMPNO is the **determinant** of NAME or that EMPNO **functionally defines** NAME.

EMPNO also functionally defines DEPTNO. Even though the same value of DEPTNO can appear in several tuples of relation EMPLOYEE, there is only one DEPTNO for an employee. Functional dependencies are often shown via an arrow from the attribute which functionally defines the other:

EMPNO ——▸ NAME,

EMPNO ——▸ DEPT, or

EMPNO ——▸ NAME DEPT.

Given two sets of attributes, A and B, where A ——▸ B, B is **fully functionally dependent** (FFD) on A if there is no subset, C, of A such that C ——▸ B. For example, in the following relational schema for automobiles

AUTOMOBILES

LICNO	MAKE	MODEL	OWNER	VIN

the following FD can be shown:

VIN OWNER ——▸ MAKE.

Where MAKE is the manufacturer of the vehicle, OWNER is the name of the owner, and VIN is the unique vehicle identification number of the car assigned at the factory. However, OWNER is superfluous to the left-hand side of the dependency because it is true that

VIN ——▸ MAKE.

Therefore, VIN fully functionally defines MAKE.

Sometimes functional dependencies hold only because there is some intermediary field which allows the dependency to be inferred. These dependencies are called **transitive dependencies.** As an example, suppose the following dependencies exist: A——▸B and B——▸C. It is therefore true that A——▸C; but this dependency is transitive because if we know a value of A, we can find the corresponding value of B, and with the value of B, find the appropriate value of C. Transitive dependencies are generally discarded as they provide no new information.

Given two sets of attributes, A and B, B is **multivalue dependent** (MVD) on A if for one value of A, there are zero, one, or many values of B associated with it. Consider the relation schema

EMPLOYEES

EMPNO	DEPT

.

The same value of DEPT can appear for different values of EMPNO; therefore, DEPT —▸▸ EMPNO. Alternative terminology

for this concept is to say that DEPT is the **multideterminant** of EMPNO, or that DEPT **multidefines** EMPNO.

Many texts use a different definition for multivalue dependencies that require reference of a third attribute from a relation. The DMNS will use the definition above.

Normal Forms

In normalization a data base design is repeatedly modified to achieve higher, "more normalized" forms. The transition from one level to the next is achieved by making different design changes at each level rather than by reapplying the same technique.

The steps from one form to the next are called **decompositions.** There are two rules which successful decompositions must follow. First, a decomposition should be nonloss: that is, the relation which is being decomposed must equal the NJOIN of the two relations to which it was decomposed. Another way of saying this is that the original relation can be rebuilt with the NJOIN operation. Second, decomposition should preserve all dependencies in the original relation. The attributes chosen in the output relations must still have all the dependencies intact. Obviously, if an FD exists from Attribute A to Attribute B in Relation X and if X is decomposed to Relations Y and Z, then A and B must <u>both</u> decompose to Y or <u>both</u> decompose to Z. Otherwise, there would be no way to continue to enforce

the FD between Attributes A and B if they were in different tables.

Recall that a relation is a subset of the Cartesian product of a set of domains. Then, to qualify as a relation a file must be made up of atomic (individual) values for each column. Such files are often referred to as **flat files.** Many computer systems allow files which do not fit within this definition to be created. Such systems allow a field defined within a single record to have a set of values. Such a field is called a **repeating field.** In some cases, two or more fields may repeat at once. Such groups of repeating fields are called **repeating groups.** Files with repeating fields or repeating groups in them are said to be **unnormalized.**

If a file qualifies as a relation by having only atomic values then it is in **first normal form** (1NF). 1NF is the lowest level normal form for a relation. To create 1NF relations from an unnormalized file, the file must be "flattened" by duplicating the nonrepeating fields such that a flat file is formed. An unnormalized file structure is shown in Figure 22.

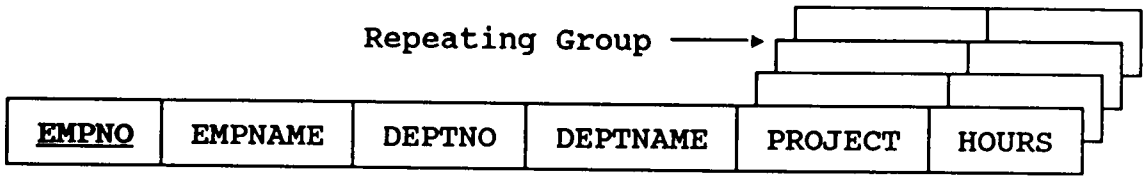


Figure 22. A file which is unnormalized.

EMPNO is the primary key of this file, and the fields EMPNAME, DEPTNO and DEPTNAME are functionally dependent on EMPNAME. The fields PROJECT and HOURS are, however, parts of a repeating group. This means that the file in Figure 22 is unnormalized.

To create a relation in the first normal form from an unnormalized file, any repeating fields in the unnormalized file must be eliminated. This is achieved by flattening the file into a relation where values of EMPNO, EMPNAME, DEPTNO, and DEPTNAME must be repeated for each pair of values of PROJECT and HOURS in the repeating group. Obviously, this 1NF relation is highly redundant. This 1NF relation is shown in Figure 23.

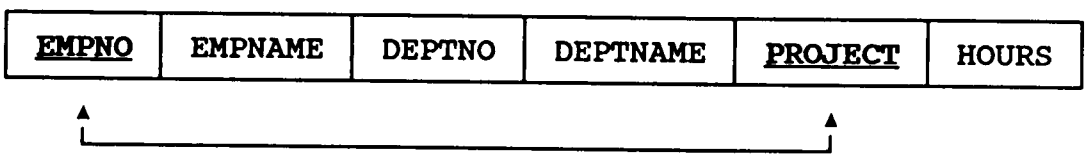


Figure 23. 1NF relation from an unnormalized file.

First normal form ensures that data in a file is in the form of a relation. It does not ensure that the attributes in the file are functionally dependent on the primary key. A relation in which all attributes are functionally dependent on the primary key is said to be in **second normal form** (2NF). Consider the relation created previously and shown in Figure 23. This relation is in 1NF but not in 2NF. The primary key of the relation is now the combination of the fields EMPNO and PROJECT. The field HOURS is functionally dependent on the entire primary key. However, the other fields are not functionally dependent on the entire key. The original key of the unnormalized file, EMPNO, is the determinant of the fields EMPNAME, DEPTNO, and DEPTNAME.

In order to transform this 1NF relation into 2NF relations, a decomposition is performed, based on projecting on the fields which are determinants of a functional dependency and their respective attributes. For the example relation in Figure 23, first, a projection on the fields EMPNO, PROJECT, and HOURS is taken. The combination of the fields EMPNO and PROJECT is the determinant of HOURS. Second, a projection on the fields EMPNO, EMPNAME, DEPTNO, and DEPTNAME is performed. The relations which result are shown in Figure 24. All attributes in every relation of Figure 24 are fully functionally dependent on the primary keys of the relations so these relations are all in 2NF.

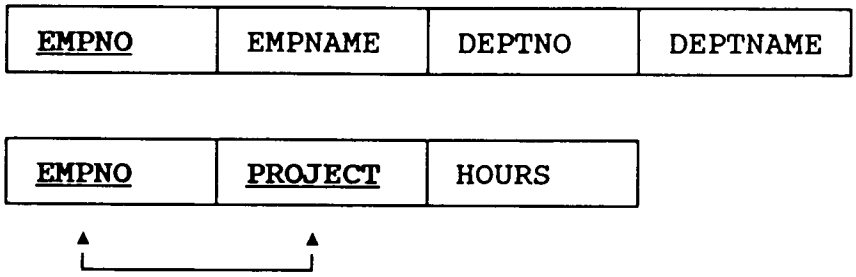


Figure 24. 2NF relations from a 1NF relation.

Although second normal form requires that all attributes be functionally dependent on the primary key of the relation, it still allows the existence of a functional dependency among nonkey attributes to be a transitive FD. This implies that in a 2NF relation nonkey attributes do not have to be mutually independent of each other. In a **third normal form** (3NF) relation all attributes are fully functionally dependent on the primary key and are mutually independent (no transitive dependencies). If a relation is in 2NF but not in 3NF, it can be converted to 3NF relations using projections to eliminate any transitive dependencies. Reviewing the relations of Figure 24, the top relation which has the primary key EMPNO is not in 3NF as will be discussed further in the next paragraph. The bottom relation of Figure 24 is in 3NF since there are no transitive dependencies.

All the attributes in the top relation of Figure 24 are functionally dependent upon EMPNO. But, attribute DEPTNAME

is also functionally dependent upon DEPTNO. Since EMPNO is the determinant of DEPTNO and DEPTNO is a determinant of DEPTNAME, then DEPTNAME is transitively dependent upon EMPNO. To decompose this relation into two 3NF relations, all attributes must be mutually independent. To achieve this for the relation in Figure 24, a projection on the fields EMPNO, EMPNAME, and DEPTNO, and a second projection on the fields DEPTNO and DEPTNAME are taken. These projections yield the relations shown in Figure 25, both of which are in the 3NF.

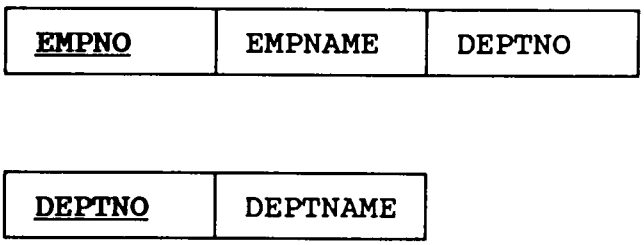


Figure 25. 3NF relations from a 2NF relation.

The next stronger level of normalization beyond third normal form is **Boyce-Codd Normal Form** (BCNF). All relations which are in 3NF are not necessarily in BCNF, but all relations in BCNF are in 3NF. While the definition of 3NF deals with only the attributes being FFD on the keys, BCNF also does not allow functional dependencies to exist between components of the candidate keys, such as occurs when

candidate keys have overlapping fields. An example of a 3NF relation which is not in BCNF is shown in Figure 26.

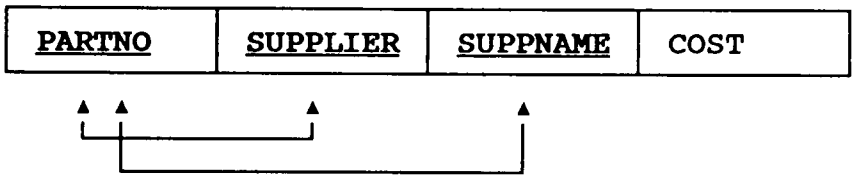


Figure 26. A 3NF relation which is not in BCNF.

The two candidate keys for the relation in Figure 26 are PARTNO+SUPPLIER and PARTNO+SUPPNAME. The file is in 3NF as COST is the only attribute in the relation and PARTNO+SUPPLIER——▸COST and PARTNO+SUPPNAME——▸COST. As candidate keys, PARTNO+SUPPLIER and PARTNO+SUPPNAME must be FFD with each other. Since PARTNO is always FFD with itself, it is obvious that SUPPLIER——▸SUPPNAME and SUPPNAME——▸SUPPLIER. Although these dependencies can exist within the keys in 3NF, they may not in BCNF. To create BCNF relations from 3NF relations, projections are taken on the 3NF relation to eliminate any FDs embedded within the keys. For the example relation in Figure 26, a projection is performed on SUPPLIER and SUPPNAME to create a new relation with the SUPPLIER/SUPPNAME cross-reference maintained there. As for the original relation, one of the candidate keys must be eliminated by leaving either SUPPLIER or SUPPNAME out of the project on the rest of the fields of the original relation. If PARTNO+SUPPLIER is chosen to be

the primary key (and now the only key) of the original relation, then SUPPNAME is left out of the projection. The two new relations in BCNF formed from these projections are shown in Figure 27.

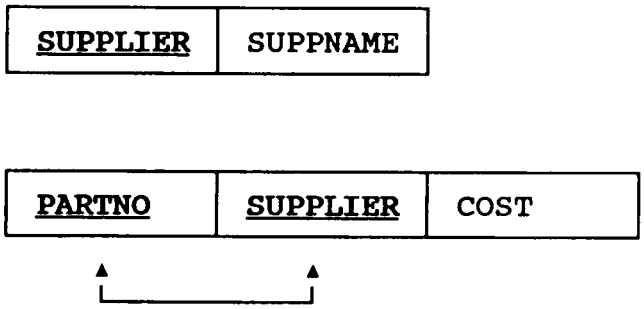


Figure 27. BCNF relations from a 3NF relation.

The last level of normalization which will be dealt with by the DMNS is the **fourth normal form** (4NF).[7] A relation in BCNF which is not in fourth normal form is an all-key relation (no functional dependencies) with multivalue dependencies existing between more than one pair of the components of the keys. As an example, a relation which is in BCNF but not in 4NF is shown in Figure 28.

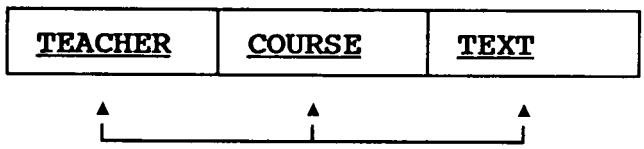


Figure 28. A BCNF relation which is not in 4NF.

---

[7]R. Fagin, "Multivalued Dependencies and a New Normal Form for Relational Databases," ACM Transactions on Database Systems, 2, No. 3, pp. 262-278.

Semantically, the data stored in this relation tracks which teachers teach which courses and which texts are required for which courses. A teacher may teach several courses, a course may be taught by different teachers, a text may be used in different classes, and a course may have multiple texts. The dependencies applying to the fields in this relation are then:

TEACHER—►►COURSE,
COURSE—►►TEACHER,
TEXT—►►COURSE, and
COURSE—►►TEXT.

The MVDs TEACHER—►►TEXT and TEXT—►►TEACHER have not been included because mention of such relationships were not included in the description above and, therefore, are not considered to be important semantically. Had the associations between TEACHER and TEXT been important the MVDs between them would have to be included and the normalization affected accordingly.

To transform this BCNF relation into fourth normal form relations, project on the multideterminant and its attribute in each of the multivalue dependencies (TEACHER—►►COURSE or COURSE—►►TEACHER) and (TEXT—►►COURSE or COURSE—►►TEXT). The resulting relations are shown in Figure 29.

COURSE	TEACHER

COURSE	TEXT

Figure 29. 4NF relations from a BCNF relation.

Although higher levels of normalization than 4NF have been defined, they are rarely encountered in real life. The DMNS prepares a relational data base design which is in fourth normal form.

V. DATA MODELING

This section describes the process around which the DMNS is built--data modeling. **Data Modeling** is a data base design methodology which involves identifying data items and the associations which exist between them.

An Overview of Data Modeling

Before going any further, data modeling terminology will be introduced. Unfortunately, data modeling terminology differs somewhat from that of the relational model. Rather than attempting to follow through with the relational model terminology, the author will switch to allow the reader to more readily follow readings from other data modeling sources. The following is a listing of definitions of data modeling terms to be used.[8]

Data items--"attributes" in relational terminology.

Data item groups--a convenient groupings of data items, the same as "tuples" in relational terminology.

Association--a dependency, either FD or MVD.

Identifies--"functionally defines" or "is the determinant of" in relational terminology, shown by a single-headed arrow from the identifier to the item defined.

Candidate Key--a data item which functionally defines another data item.

Primary Key--a key chosen from among one or more candidate keys to be the primary key of a relation and leaving

---

[8]James Martin, Managing the Data-Base Environment (Englewood Cliffs, N.J.: Prentiss-Hall, 1983), pp. 171-202.

any other candidate keys in the relation as alternate keys.

Attribute--any data item which does not functionally define any other data item(s).

Secondary key--an attribute (not really a key at all) which multidetermines one or more other data items. Secondary keys often have index files created for them in order to provide speedy random access to the values of the attribute in the file.

Root key--the primary key of some relation which does not identify the primary key of some other relation.

In addition to these terms, some definitions of new concepts relating to data modeling are also needed. These are:

User View--representations of a group of data items and the relationships between the data items in the grouping. A user view is analogous to the view level of the three-level data base architecture.

Canonical Synthesis--the process of combining user views into a normalized structure.

Logical Data Model--a diagram of data items and dependencies encompassing the data requirements from all the user views which apply to a particular application or area. The logical data model is analogous to the conceptual level of the three-level data base architecture.

The goal of the data modeler is first to develop the logical data model and later from the logical data model develop a physical data model. The sequence of tasks required to create a logical data model is:

1. Determine the bounds of the project.
2. Identify individual data inputs and outputs which need to be documented as user views.
3. Create user views of these data inputs and outputs.

4. Enter the user views into a canonical synthesis program, or perform synthesis manually.

5. Analyze the output of the synthesis to look for errors and semantic inconsistencies.

6. Correct original user views which led to errors or enter additional views required to resolve inconsistencies.

7. Redo steps 4, 5, and 6 until data model is error free.

At the end of this process the designer should have an accurate, stable logical data model which is independent of any particular DBMS. From this logical data model the physical data design would be created. Usually, the physical design differs from the logical design only when poor performance of some important transaction is too slow when a direct logical to physical mapping is used.

Pictorially, the steps of the data modeling process are shown in Figure 30.

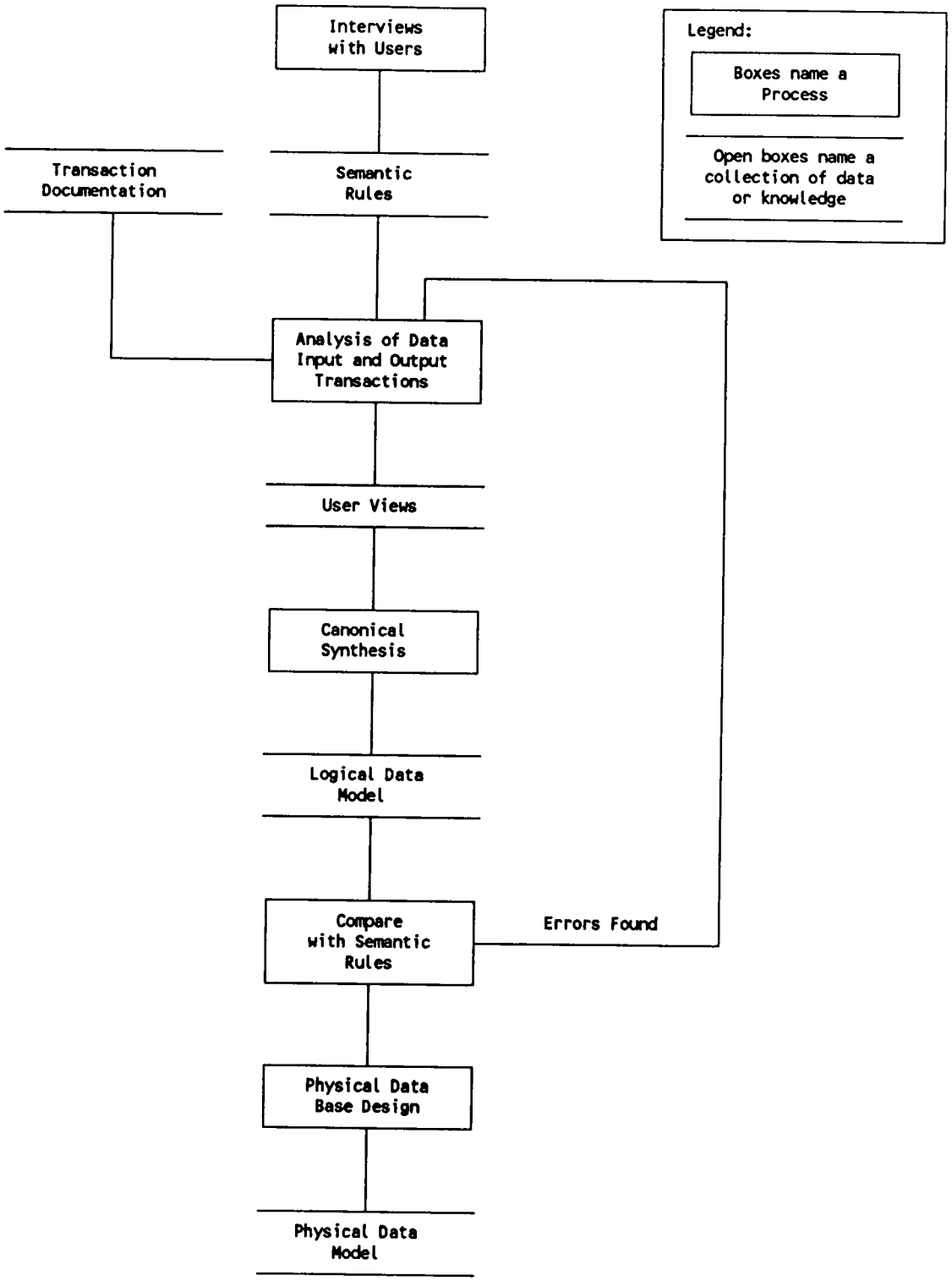


Figure 30. Flowchart of the data modeling process.

User Views

The user view is the basic building block of the logical data model. But how are user views created and where are they found?

Data modeling is based on semantic interpretation of data, not on the values of data in the fields. Data modeling depends on the modeler's own judgment and human understanding of the data. Looking at data values can prove that an association between two data items which the modeler thought to exist cannot actually exist. While looking at data values may suggest an association between two data items, it never proves the existence of an association.

The data modeler traces through the transactions required to perform business functions to acquire the knowledge necessary to design the data base to serve the business function. It is the investigation by the data modeler and resolving of such questions with the actual data users as, What identifies an employee?, What data items are needed to identify quantity sold?, and Are the same textbooks required for a class regardless of who is teaching the class? that provide the semantics for defining the associations between data items. The answers to the questions above might be Social Security Number; Product ID, Year, and Month; and Yes, respectively. From these answers, data item associations are formed (or not formed). The data modeler learns through experience how to quickly

fill in the gaps in a data model by asking the right questions.

Consider a user view which includes the following data elements: NAME, ADDRESS, PHONENO, and NAME_OF_PET. This indicates that NAME describes a person, each person has a single address and a single phone number (or no phone number), and that each person can have no pet at all, one pet, or several pets. These data dependencies can be shown as:

NAME ——▸ ADDRESS,

NAME ——▸ PHONENO, and

NAME —▸▸ NAME_OF_PET.

User views are often drawn with the data items in ovals, functional dependencies as single-headed arrows, and multivalue dependencies as double-headed arrows. These diagrams are commonly called **bubble charts.**[9] The user view is shown as a bubble chart in Figure 31.

---

[9]James Martin, Computer Data-Base Organization (Englewood Cliffs, N.J.: Prentiss-Hall, 1977), pp. 69-72.

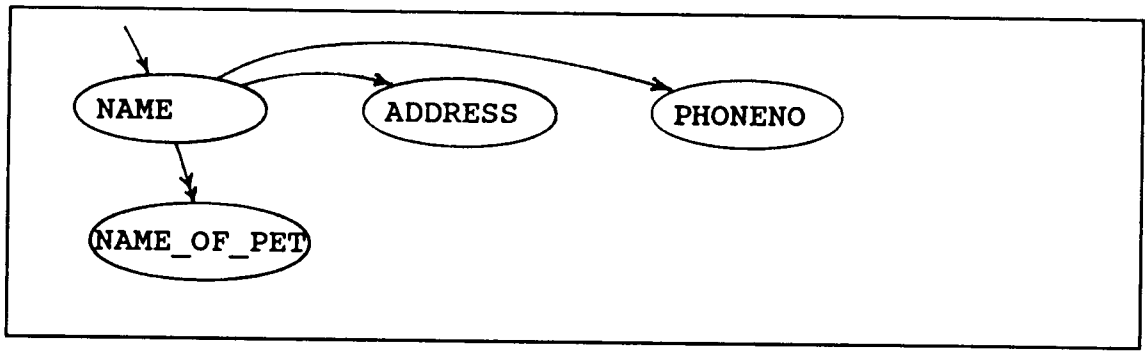


Figure 31. Bubble chart.

User views can be formed for every place that data will enter or leave a data base. Some examples of how data gets into and out of a data base are shown in Figure 32.

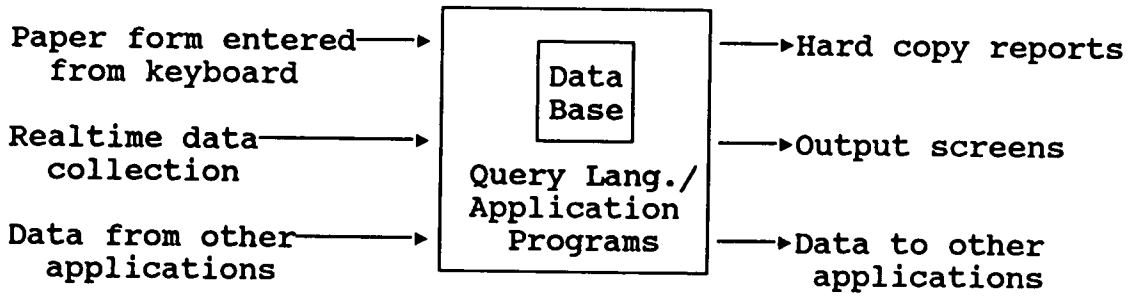


Figure 32. Data base application context diagram.

Because user views are collected from forms, screens, reports, etc., that are for some specific business transaction, the data items which are in each user view are usually closely associated with each other. These associations must be determined from investigations into the meaning of the data with the users of the data. It is critical to define each data item and to make sure that

different users are not using synonyms or using multiple definitions of the same data item. If the system has been computerized before, do not rely on the previous design or the programmer's definition of data items and associations. This leads to the perpetuation of the old errors in the data model.

The data modeler should attempt to represent the data associations in the user view in Boyce-Codd Normal Form--this is critical in developing a user view. If some of the data associations shown in the user view are not in BCNF, the canonical synthesis process will find inconsistencies between these associations and other associations for the same data items from other user views. This is a normal occurrence as the correct interpretation of data item associations may not be very clear in the beginning of the modeling process. The DMNS provides diagnostics if different user views do not agree with each other as to what data items are the determinants of other data items. After the data modeler determines which user view is in error, this user view should be corrected and fed back into the canonical synthesis tool. Eventually, through this cycle of error identification and correction, all user views will be in 4NF. It is during this process that the data modeler must demonstrate ability to recognize the true data dependencies and to resolve inconsistencies.

Drawing user views is as much an art as a science. An example of a user view and a discussion of how it was created from the transaction being modeled should prove helpful. Consider the sample report from Figure 33.

Date: 9/24/87

Employ. Number	Dept.	Employee Name	Telephone	Monthly Salary
2842	41	JK Lovin	304-8273	1,800
1629	26	EP Yount	304-7329	2,500
1947	41	OC Willard	304-3791	3,000
2718	41	MS Howard	304-8273	2,000

Figure 33. Employee listing.

To draw a user view for the example report in Figure 33, the data modeler could start by listing the data items of interest. The data modeler should know the data item names in the data dictionary. They might then appear as: EMPNO, DEPTNO, EMPNAME, EMPPHONE, and MOSAL. Notice that the report date in Figure 33 was not listed. This is because the data modeler decided that this data item is not significant to the data base design. If no one will ever need to know the date this report was printed, then the date can be disregarded in the user view.

Next, the user view's **entry point** (or points) is found. An entry point is a data item which is not identified by any other data items to be included in the user view. These

are often found in the upper left of reports and input forms because this organization scheme makes them more readable. For this example user view, this heuristic holds as the data item EMPNO is the user view's sole entry point. Starting from the entry point, the data associations between these data items are determined and drawn. Each employee belongs to a single department and has a name, a telephone, and a salary. Drawing these associations in a bubble chart format yields the chart of Figure 34.

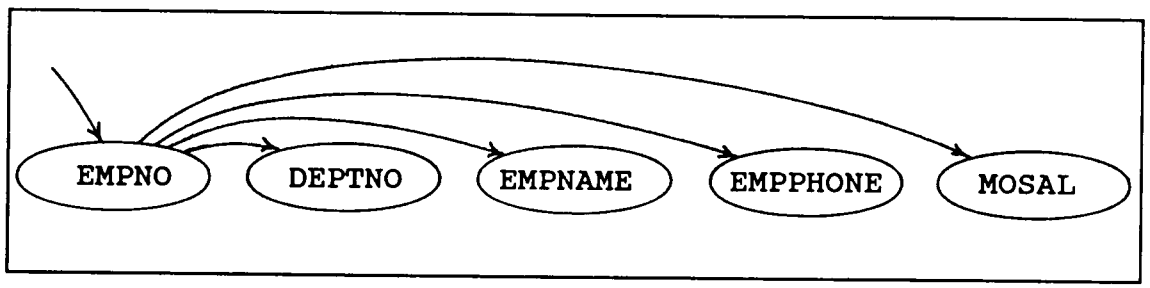


Figure 34. Bubble chart of employee listing.

It is important to note that on a single user view, all arrowheads normally point away from entry points. Reverse associations are not usually shown because, if they are important to the logical data model, they will appear on some other user view.

Now consider another report closely resembling the first. The primary difference in this report and the first report is that employees have been grouped by their

department. A sample page from this report is shown in Figure 35.

Dept.: 41 Dept. Name: Purchasing Date: 9/24/87

Employ. Number	Employee Name	Telephone	Monthly Salary
2842	JK Lovin	304-8273	1,800
1947	OC Willard	304-3791	3,000
2718	MS Howard	304-8273	2,000

Figure 35. Employee list by department.

The data items found in the report shown in Figure 35 are--DEPTNO, DEPTNAME, EMPNO, EMPNAME, EMPPHONE, and MOSAL. The entry point for this user view is not the same as before. Here the department number is the entry point, because the association of interest between employee number and department number is from department to employee. That is, this report shows the employees in a department, not what department an employee is in. Describing the dependencies from the entry point, a department number identifies a department name, a department number has many associated employee numbers, and an employee number has an associated employee name, telephone number, and monthly salary. The bubble chart can then be drawn as in Figure 36.

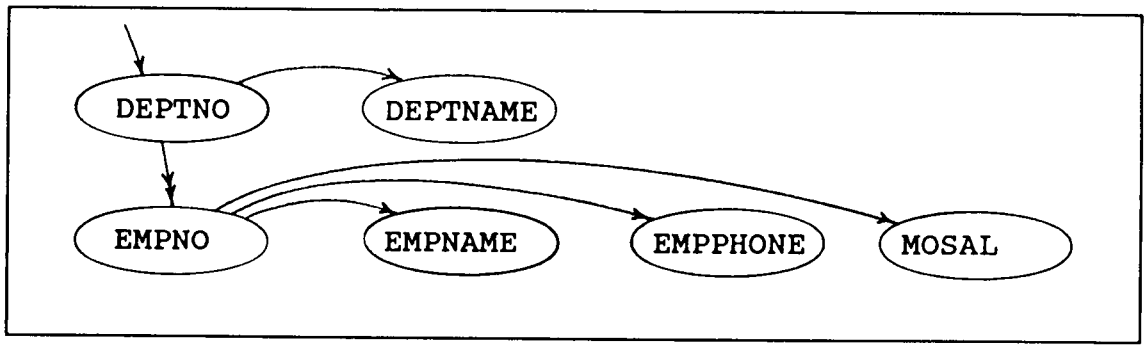


Figure 36. Bubble chart of employee list by department.

Between the two user views of Figure 34 and Figure 36, the associations in both directions between employees and departments have been completely defined.

Canonical Synthesis

Canonical synthesis is the process of combining user views into a normalized logical data model. This process aggregates data items into data item groups that are synonymous with normalized relations in the relational model. This section first presents a popular manual technique for canonical synthesis and then presents the algorithm used in the DMNS.

James Martin[10] lists steps for performing canonical synthesis by hand to generate a logical data model. Before listing these steps, it must be noted that there are some differences in the scope of Martin's steps, and that

---

[10]James Martin, Managing the Data-Base Environment, (Englewood Cliffs, N.J.: Prentiss-Hall, 1983), pp. 249-251.

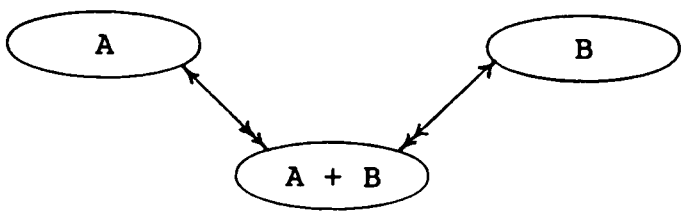
Martin's terminology has been modified slightly to be more consistent with that used in this thesis.

Martin's steps only normalize data to the third normal form, include considerations for other types of DBMSes other than those based on the relational model, and require resolution of inconsistencies and uncertainties as they appear. There are also additional steps for stability analysis and generating a physical design which are not included here. By comparison, the canonical synthesis performed by the DMNS will normalize to fourth normal form, consider only the relational model, and report errors for correction on later runs.

Presented now are Martin's steps for manual canonical synthesis:

1. Take the first user view of data and draw it in the form of a bubble chart--a graph with point-to-point directed links between single data items, representing associations of the two types: functional dependencies and multivalue dependencies.

   Where a concatenated key is used, draw this as one bubble, and draw the component data items of the concatenated key as separate bubbles, thus:



   Check that the representation avoids hidden transitive dependencies. Where a concatenated key data item has to be used, ensure that all single-arrow links from it go to data items which are dependent on the full concatenated key, not merely part of it. In other

words, ensure that the representation of the user's view is in third normal form.

Otherwise draw only the associations which concern this user.

2. Take the next user's view, representing it as above. Merge it into the graph. Check for any synonyms or homonyms in the attribute names. If any are found, resolve the naming inconsistency by revising the appropriate user views.

3. In the resulting graph distinguish between the attribute nodes and the primary key nodes. (A primary key node has one or more single-arrow links leaving it.) Mark the primary keys in some way, e.g., red color.

4. For each association between keys, add the inverse association if it is not already on the graph. If this results in bidirectional MVDs between keys, determine whether the inverse association would ever be used in reality. If it could be used at any time in the future, replace it by introducing an extra concatenated key incorporating the key data items that were linked.

5. Examine the associations and identify any that appear transitive. For any associations that are candidates for removal, check carefully that their meaning is genuinely transitive; if so, remove them.

6. Repeat the previous four steps until all user views are merged into the graph.

7. Identify the root keys. (A root key is a primary key with no functional dependencies leaving it to another primary key.)

   For pictorial clarity the diagram should be rearranged with the root keys at the top. The single-arrow links between keys should point upward where possible. The links between primary keys may be marked in color.

8. Observe whether the graph contains any isolated attributes. An isolated attribute could be treated in one of three ways:

   a. it may be implemented as a repeating attribute in a variable-length record;

   b. it may be treated as a unary relation; or

c. it may be the result of an error in interpretation of the user's data, in which case the error is corrected.

9. Adjust the graph to avoid any intersecting attributes. An intersecting attribute can be avoided by:

   a. replacing one or more links to it with equivalent links via an existing key;

   b. duplicating the data item in question; or

   c. treating it as a unary relation--a one-data-item record.

10. Redraw the data items arranged into groups (records, segments, tuples), each having one primary key and its associated attributes. A group may now be drawn as a box.

11. Identify all secondary keys. Draw the secondary-key links between the boxes.

As can be seen, Martin's approach requires that problems be fixed as soon as they are encountered. If the DMNS were to use this approach it would take a long time to find and fix all the problems in a data model as a new run of the DMNS's normalization workbench would be required each time a problem was encountered. Instead, it is more efficient for the DMNS to perform its normalization without human intervention. Whenever the DMNS encounters problems, it must do what most often works in the particular situation and continue as best it can. Anytime an uncertainty or a problem arises the DMNS must inform the data modeler what assumptions it made and what actions it took to resolve the problem. The steps performed by the DMNS are now presented below.

Entry Phase:

1. After drawing user views as described earlier in this section, enter these in the DMNS. Start the DMNS synthesis process from the DMNS menu. The following steps in the merge, normalization and output phases are then performed by the DMNS.

Merge Phase:

2. Add new vertices and edges found in each user view.

3. Expand any concatenated keys into the individual components of the concatenated key and generate MVDs from the individual components to the concatenated key.

4. If an association between data items is MVD in both directions, it will be replaced by introducing an extra concatenated key incorporating the key data items that were linked.

Normalization Phase:

5. Locate any cases where two vertices have both an FD and an MVD between them in the same direction. Delete the FD and warn the modeler.

6. Functional dependencies are examined for transitivity. For any FDs that are transitive, remove them and print a diagnostic message.

7. Find any overlapping candidate keys with functional dependencies within the keys (violating BCNF). Choose one key as the primary key and decompose the data item group. Print a diagnostic message.

8. Search for candidate keys. If candidate keys are found, choose one as a primary key.

9. Find any isolated attributes. Treat the attribute as a unary relation. Print a warning message because this may be the result of an error in interpretation of the user's data.

10. Locate any instances of keys with an association in one direction but an unknown association in the other. Assume that the association in the unknown direction is an MVD if the known association is FD and visa versa. Print a warning message.

11. Examine any unary relations which may exist to see if the modeler wishes to retain them. If not, remove the unary relation.

12. Locate any intersecting attributes. Duplicate the data item in all relations in which the attribute is functionally determined by relation's primary key. Print an error message.

13. Locate any undirected cycles among the functional dependencies and warn the modeler of their existence.

14. Find any all-key data item groups with multiple MVD associations between the data items in the data item group (violating 4NF). Split this data item group into data item groups each with a single MVD. Print a diagnostic message.

15. Match data item group keys with relation identifiers in order to label data item groups. Label any unmatched data item groups as "Unknown Relation."

Output Phase:

16. Ensure that each data item in the user view is in the data dictionary. If not, print an error message.

17. Print the data items arranged into data item groups, each having one primary key and the key's associated attributes. The groups are now printed as boxes.

Review Phase:

18. The data modeler now reviews the error and diagnostic messages generated during the synthesis. If any errors are found, correct the user views in which they appear and return to step 2.

Logical Data Model Considerations

A logical data model is a diagram of data items and the dependencies between them encompassing all the data requirements of the realm of interest. This "realm of interest" could include only a single application or the data base needs of an entire company. When writing a single

data base application for one aspect of a business, the important aspect is to create a data base which will not need major restructuring later when other applications are written. By no major restructuring, it is meant that attributes may be added to relations; but no existing relations can be split or a new relation inserted in such a way that the referential integrity and data independence of existing relations will be disturbed.

A data base design which does not need to be constantly altered to meet new applications is known as **stable.** Stability requires foresight in determining what other applications might impact the data base design. Any other applications which may require changes to the data base structure should be modeled at the same time as the first application. Then, a data base that will eventually serve all applications which require this data can be built when the first application is written.

For example, creating a logical data model for a purchasing department data base system without also modeling the needs of the receiving department would probably be a mistake. The data dependencies needed in logical data model for the receiving department may affect the structure of the logical data model designed for the purchasing department. When the differences are found later as the receiving department's application is written, reprogramming will be required for the purchasing system to accommodate the

changes required in the data structure of the data base they must share.

Once a logical data model has been completed, it would then be transformed into a physical data base design. Generally this is a direct mapping from the logical data model into the data definition language of the data base management system to be used. The DBMS then creates the actual data file matching the structure defined in the DDL syntax. Sometimes performance considerations will make it desirable to deviate from the logical design when creating the physical design, but most often the logical data model will be directly transformed into a physical data base.

Benefits of Data Modeling

Before leaving the topic of data modeling, it should be noted that the use of data modeling as a data base design tool has some beneficial properties that can be used to double check the accuracy of user views comprising the model. One of these properties is that user views provide traceability back to the transactions which are the sources of data items and to associations which have become part of the logical data base schema.

User views could also be used to help produce data flow charts that would be very beneficial to the application programmer. Such charts would help in designing locking

protocols for network or multiuser data base systems and in developing update sequencing.

Another property of data modeling is that one can perform a "data balance" on data items to help locate user views which need to be collected but have been missed. This is done by simply checking to ensure that every data item in the schema which has appeared in an input or edit transaction also appears in an output transaction, and that every data item which appears in an output transaction also appears in an input or editing transaction. The rationale for making this check is that you cannot print what you have not stored and that there is probably no reason to store what you do not want to print, indicating that either a user view is missing or some data items in the data base are not needed.

VI. DATA MODELING AND NORMALIZATION SYSTEM OVERVIEW

The software accompanying this thesis was created to enable the storage and organization of user views, data items, and relations and to automate the process of normalization and the design of relation data base applications.

This section is an overview of the capabilities provided by the software. It is not intended to be a software user's manual for the system. Such a document, User's Guide to the Data Modeling and Normalization System, is available from the author.

The DMNS operates on personal computers which run the MicroSoft (R) Disk Operating System ((C) MicroSoft) (MS-DOS). The DMNS requires 512kB of free memory to operate. For hardcopy output a printer is required.

The DMNS is divisible into two major component programs written in different languages. These two components are (1) the menus and data management component and (2) the normalization component. The menus and data management component of the system is written in dBASE III Plus ((C) Ashton-Tate) and use the Saywhat! ((C) The Research Group) screen management package. The normalization component is written in Microsoft (R) C Optimizing Compiler ((C) MicroSoft) and uses the dBC III Library ((C) Lattice (R)), a dBASE III file access utility.

Application Schema File Organization

The DMNS stores data in separate subdirectories for each application which has been modeled. In each application subdirectory there are several files. The result of a DOS directory command of a typical application subdirectory is shown in Figure 37. There will be nine files with a file name extension of ".DBF," eight with ".NDX," and one with ".MEM." This is the dBASE file naming convention with .DBF indicating "data base file," .NDX indicating "index file," and .MEM indicating "memory file."

```
 Volume in drive C is SEAGATE4051
 Directory of  C:\DMNS\MARTIN.APP

.            <DIR>       4-09-88   8:50a
..           <DIR>       4-09-88   8:50a
DEP      DBF     5024    3-19-88  12:15p
DEPTMP   DBF      929    4-16-88   4:59p
FIELDS   DBF     1024    4-16-88   4:25p
RELATION DBF      512    4-16-88   6:43p
UV       DBF     2175    4-11-88   9:47a
UVB      DBF      953    4-17-88  10:13a
VS       DBF      212    4-16-88   9:37a
VSSTRUCT DBF      623    4-16-88   9:37a
WBENCH   DBF      154    4-16-88   5:01p
DMNS     MEM       82    4-16-88   6:44p
DEP      NDX     2048    4-17-88  10:09a
FIELDS   NDX     1024    4-16-88   4:25p
UV       NDX     1024    4-09-88  10:14a
UVBBYDEP NDX     2048    4-17-88  10:13a
UVBBYUV  NDX     2560    4-17-88  10:13a
VS       NDX     1024    4-16-88   9:37a
VSPARENT NDX     1024    4-16-88   9:37a
VSSTRUCT NDX     2048    4-16-88   9:37a
       20 File(s)   2752512 bytes free
```

Figure 37. Sample directory of an application subdirectory.

The memory file, DMNS.MEM, contains the name of the application.

The .DBF files contain the data modeler's information about the application. In the VS.DBF file are the names of all view sets in the application. This file contains the fields VSNAME and VSMODDATE. VSNAME is a ten-character wide field for the view set name. VSMODDATE is a data field for the latest date when the view set was modified.

The VSSTRUCT.DBF file is a cross-reference indicating which user views are in each view set. VSSTRUCT.DBF includes the fields VSNAME and UVNAME. Once again, VSNAME is the name of the view set. UVNAME is the name of the user view which is included in the view set. Both fields are character fields which are ten places wide. For a view set composed of three user views there will be three records in the VSSTRUCT.DBF file with the same view set name in each and the name of each user view in one of the three records.

The UV.DBF file holds attribute data for user views. UV.DBF's fields are named UVNAME, UVSRCTYPE, UVSCRDESC, UVCREDATE, and UVMODDATE. UVNAME is the ten-character wide field for the user view name. UVSRCTYPE is a two-character wide field for coding the source type of the user view. These codes are left up to the data modeler using the system, but representative of the use of this field would be to let "RP" stand for report, "FM" stand for form, and "PG"

stand for a computer program. UVSRCDESC is a 254-character field to further describe the user view. For example, the contents of this field might be something like "Financial Report for Operating Divisions," which would be the document from which the user view was taken. Fields UVCREDATE and UVMODDATE are date fields for the dates the user view was created and last modified, respectively.

UVB.DBF is the file which cross-references user views and dependencies. Its two fields are named UVNAME and DEPNO. UVNAME is the ten-character name for the user view and DEPNO is a numeric field identifying a dependency. The DEPNO's value is assigned by the DMNS and is unique for every unique dependency. If two dependencies from different user views agree in their left-hand side, right-hand side, dependency type, and role name, then they are considered to be the same dependency and will have the same DEPNO. In the UVB.DBF file, two such matching dependencies would appear twice, once for each user view in which the dependency was found. This file is maintained by the DMNS and is never edited directly by the data modeler.

DEP.DBF is the file for dependencies. Its fields are DEPNO, LHSEQUAT, DEPTYPE, RHSEQUAT, and ROLENAME. DEPNO is again a numeric field identifying a dependency. In the DEP.DBF file, a specific DEPNO can appear only once. The other fields describe the dependency. LHSEQUAT and RHSEQUAT are the data item expressions on the left-hand side and

right-hand side, respectively. DEPTYPE is the dependency type (either FD or MVD) of the dependency. ROLENAME is the role name of the dependency assigned by the data modeler. ROLENAME is normally left blank, but can be used to show that two-data item expressions have multiple FDs or multiple MVDs between them which play different roles.

The file FIELDS.DBF is a list of data items (not data item expressions) to which the modeler wishes to give a description. The data stored in this file are not used in the normalization process, but are reproduced on the data base design report generated by the DMNS. The fields in this file are FLDNAME, FLDDESC, FLDTYPE, FLDTOTLEN, FLDDECLEN, and FLDMASK. FLDNAME is the ten-character name of a field (it is a field in the data base design where it is printed but is a data item in user views). FLDDESC is a 25-character field used to describe the data item. FLDTYPE is a one-character field used to indicate the type of field which FLDNAME will become in the application data base. If the data modeler is using dBASE as the data base system for the application, then the modeler can use "C" for character fields, "N" for numeric fields, "D" for date fields, and "L" for logical fields. FLDTOTLEN and FLDDECLEN are the total length of the field and number of decimal places (if applicable), respectively. FLDMASK is a 20-character field for a data validation mask for the field. Masks are used to

force entry of such conditions as digits only, alphabetic characters only, dashes in specific positions, etc.

The file RELATION.DBF allows the data modeler to describe relations in the data base design. If the modeler does not provide any further information about relations by including it in the RELATION.DBF file, then relations are identified by only their primary keys. The RELATION.DBF file allows descriptive names to be matched with these primary keys and be printed on the data base design report. The fields of the RELATION.DBF file are RELNAME, RELIDFLDS, and RELDESC. RELNAME is an eight-character field for assigning a unique short name to the relation which will also be suitable for use as a file name under MS-DOS. RELIDFLDS is the data item expression which is the primary key of the relation to be named. RELDESC is a 25-character field to assign the relation a descriptive long name.

WBENCH.DBF is the list of user views and view sets to be merged into the normalization workbench. Its single field is named DEPGRP, meaning dependency group. A dependency group may be either a user view or a view set.

The file DEFTMP.DBF is a temporary work file never accessed directly by the modeler.

The files with extension .NDX will not be described individually. They are index files for keeping track of the alphabetical or numerical order of the data in the .DBF files. They also allow random access into the .DBF files

with the indexed fields from the .DBF files duplicated in the .NDX files.

Features of the Data Modeling and Normalization System

The DMNS is easy to use. Its vertical and horizontal moving bar menus are helpful in selecting items. Data entry uses full screen mode with formatted screens or uses dBase III's full screen browse command. The DMNS' introductory screen and main menu are shown in Figures 38 and 39, respectively.

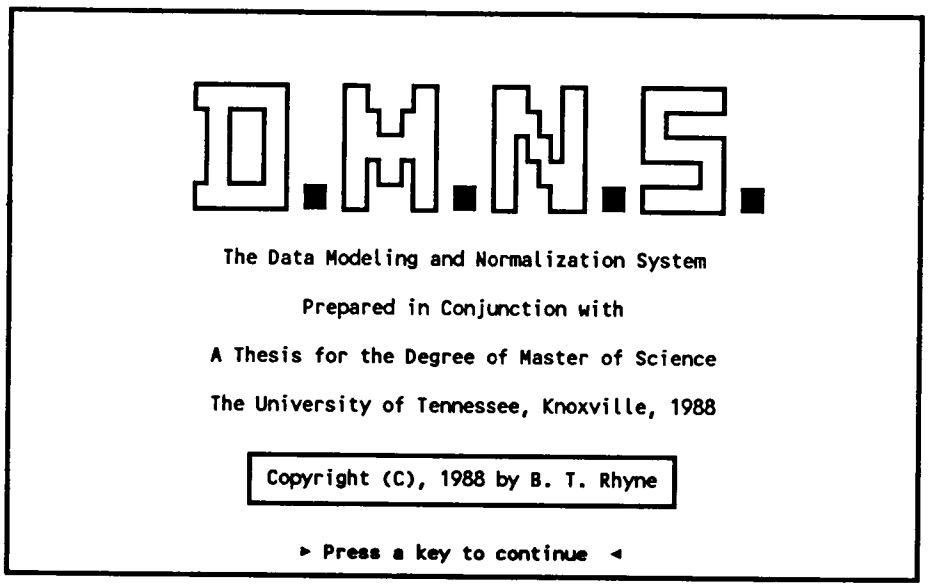


Figure 38. DMNS introductory screen.

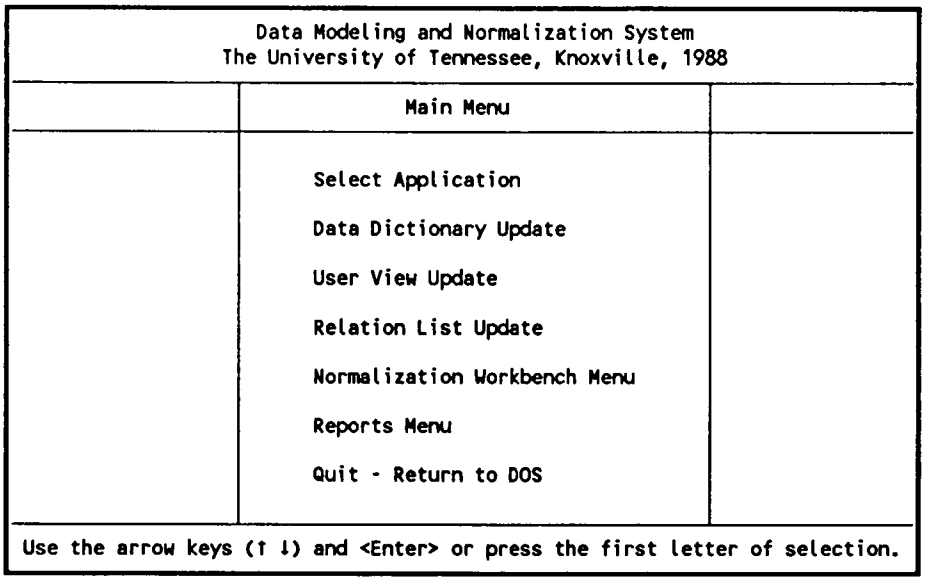


Figure 39. DMNS main menu.

The main menu makes available several options. Following is a discussion of each option.

Application Selection Menu

The first option from the main menu is to go to the application selection menu. This menu is shown in Figure 40.

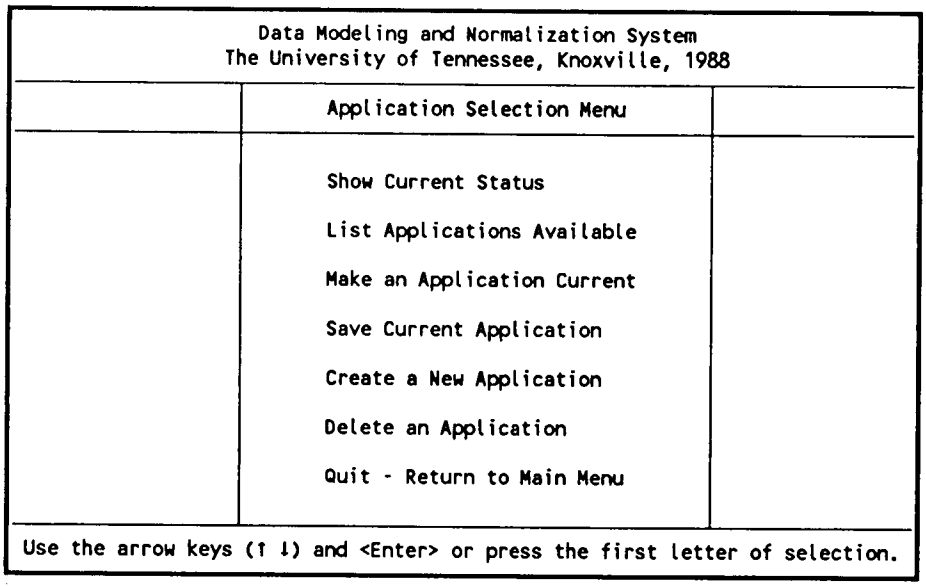


Figure 40. Application selection menu.

This menu allows the data modeler to work on several different applications at once and keep separate data models for each application. Data files for each application are kept in separate subdirectories beneath the directory in which the DMNS is installed. Each application and its associated subdirectory have a name given by the data modeler in addition to a file name extension of ".APP" for the subdirectory.

The "Show Current Status" option of the application selection menu displays the name of the application currently loaded in the DMNS directory, the amount of free space on the current drive, and the contents of the "RAM" DOS environment variable. If the user's system has a ram

drive (also known as a virtual disk, or vdisk), the DMNS will write its results to the ram drive if the user wishes. The DMNS will search the DOS environment for a variable named "RAM." RAM should be set to the appropriate drive identifier which corresponds to the ram drive; for example, D:, E:, F:, etc. This informs the DMNS as to which drive is the ram drive.

The "List Applications Available" option performs a directory of all files with a file name extension of ".APP," thus identifying the applications subdirectories by name.

The "Make an Application Current" option copies data files from the requested application subdirectory into the DMNS directory. Data files for the application in the DMNS subdirectory are used during normalization.

The "Save Current Application" option can be thought of as the converse of the "Make an Application Current" function. Choosing the save option causes the DMNS to copy the data files in the DMNS directory back to the appropriate application subdirectory. The files are still in the DMNS directory after the operation is complete; but, until this option is invoked, the application's subdirectory will not be updated.

The "Create a New Application" option creates a new subdirectory with a user-specified name and creates new files in the directory to store the data dictionary, user views, and relation data for this application.

The "Delete an Application" option deletes all data files in the application subdirectory specified by the user and removes the subdirectory from the drive.

Data Dictionary Update Option

The second option on the main menu is titled "Data Dictionary Update." This option uses the dBASE III browse command to allow the data modeler to update the list of data items in the schema. The keys used for modifying data in the browse mode are documented in dBASE III documentation[11] and in the User's Guide to the Data Modeling and Normalization System.[12] When a set of user views is normalized, the data items listed in the user views are cross-referenced with the data dictionary, and the information associated with the data item is printed on the report of the relational data base design.

User View Update Menu

The third option of the main menu is labeled "User View Update." Selecting this option brings up the menu shown in Figure 41.

The top selection from the "User View Update Menu" is

---

[11]Ashton-Tate, Learning and Using dBase III Plus (unpublished work).

[12]B. Timothy Rhyne, User's Guide to the Data Modeling and Normalization System.

tagged "Add a New User View." When this option is chosen, the screen shown in Figure 42 is displayed.

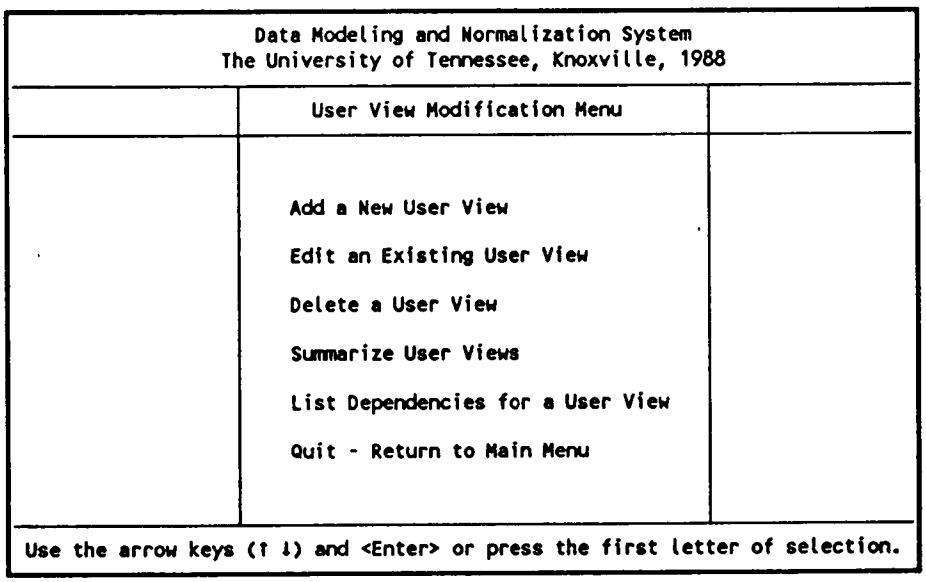


Figure 41. User view modification menu.

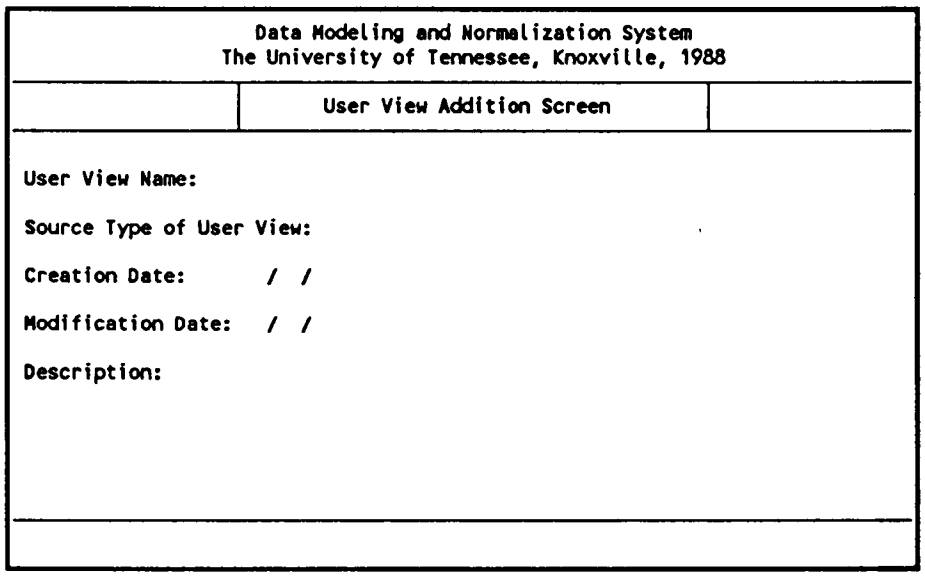


Figure 42. User view addition screen.

After data have been entered in the user view addition screen, a dBASE III browse screen will appear for the data modeler to enter the user-view dependencies. Before describing this screen, let us define a **data item expression** to be the data items which are on either one side or the other of a dependency. If this is a single data item, then it is a **simple data item expression.** If there is more than one data item, then the data items are separated by a plus sign (+) and it is referred to as a **compound data item expression.**

The data item expression for the left-hand side of the dependency, the type of dependency (FD or MVD), the data item expression for the right-hand side of the dependency, and a role name for the dependency are entered into this screen. All these fields are visible simultaneously. If the left-hand side data item expression or right-hand side data item expression of the dependency is too long to fit in the visible portion of their fields, then these fields scroll to allow longer expressions to be entered.

The second option on the User View Modification Menu is listed as "Edit an Existing User View." Selection of this option displays a new type of menu--a horizontal moving bar menu shown in Figure 43.

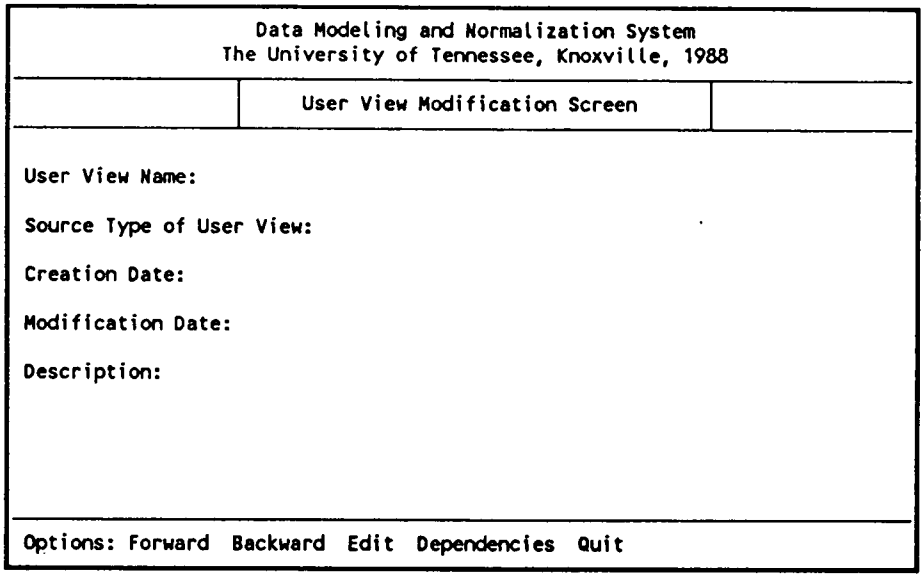


Figure 43. User view modification screen.

Data for individual user views are shown above the menu on the screen. Initially the screen shows the first user view (alphabetically). To see the next user view, the "Forward" option of the menu is used. To see the prior user view, the "Backward" option of the menu is employed. When the user tries to go forward past the end or backward before the beginning, this will cause the DMNS to sound a beep and wrap around to the user view on the opposite end of the list. The "Edit" option allows one to change information displayed on the screen. The last option (other than "Quit") is the "Dependencies" option. Choosing this option brings up a dBASE III browse screen for the user view's

dependencies. An example of such a screen is shown in Figure 44.

```
LHSEQUAT---------- DEPTYPE RHSEQUAT---------- ROLENAME---
VESSELNO           FD      VESSELNAME
VESSELNO           FD      TONNAGE
VESSELNO           FD      DETAILS
VESSELNO           FD      COUNTRY
VESSELNO           FD      VESOWNER
VESSELNO           MVD     VOYAGENO

BROWSE      |<H:>|DEPTMP          |Rec: 1/6      |    |
                    View and edit fields.
```

Figure 44. Example of dependency edit screen using browse.

In the dependency edit screen of Figure 44, dependencies can be modified, deleted, or added to the user view. The data shown on the screen are in a temporary buffer, and the changes are not applied to the application schema data base until the screen is exited.

The third option of the User View Modification Menu of Figure 41 is used to "Delete a User View." The user needs to supply the name of the user view to be deleted, and the DMNS will complete the operation. User view dependencies will be deleted if no other user view contains the same

dependency. Otherwise only the reference of this dependency to the deleted user view is removed.

The fourth option of the User View Modification Menu is used to "Summarize User Views." This option will list on the screen the name, source type, creation date, and modification date of all user views in the application schema data base.

The fifth option, "List Dependencies for a User View," lists on the screen all dependencies for a specified user view.

Relation List Update

The fourth option on the main menu is titled "Relation List Update." This option uses the dBASE III browse command to allow the data modeler to update the names in RELATION.DBF for relations formed during the normalization and relational data base design.

When a set of user views is normalized, relations are created with unique primary keys. The relation list allows these relations to be named and described. The primary keys of the relations from the normalization process are matched with the field RELIDFLDS in this file. When the primary key of a relation matches a RELIDFLDS value in RELATION.DBF the name and description found in the relation list are used to label the relation. If no match is found, the relation is named "Unknown."

Normalization Workbench Menu

Normalization is performed in the normalization workbench. The menu for the workbench is shown in Figure 45.

The workbench maintains a current list of "views" which are to be normalized. These views may either be user views or view sets (collections of user views). The first option on the menu is "Choose Views for Work Bench." This selection allows the list to be revised. The list is then saved in a file and, therefore, is nonvolatile. This file is also considered a part of an application; so when an application is saved or loaded, this file accompanies it.

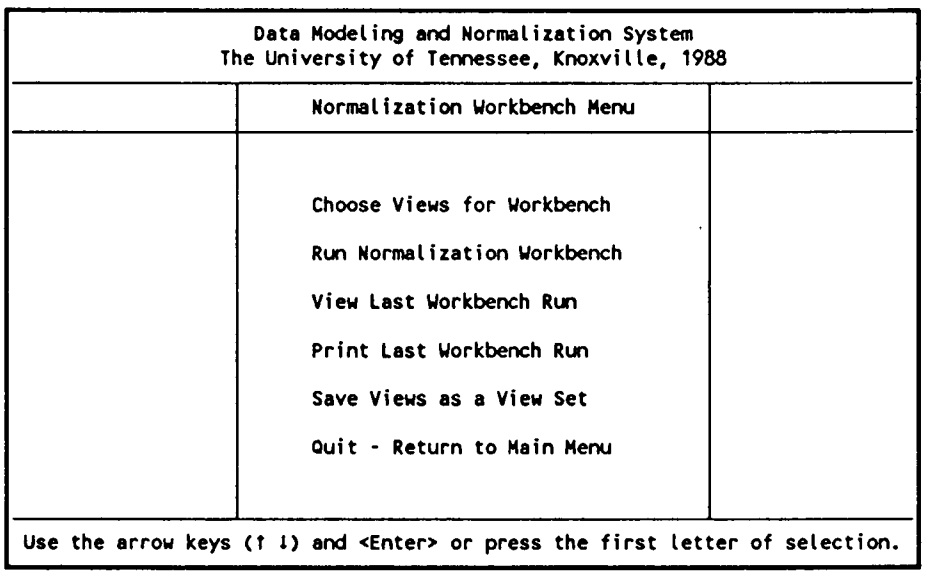


Figure 45. Normalization workbench menu.

The second choice on the workbench menu is "Run Normalization Workbench." It executes the C language

program for normalization and relational data base design using the user views and view sets requested above. As the workbench runs, it informs the user which phase it is executing. These phases are the selection, merge, normalization, and output phases. After the workbench is finished, the results, which have been saved in a file named WBENCH.MSG, are displayed on the screen. More detail concerning the normalization work bench will be presented in the next section.

The third option on the normalization workbench menu is "View Last Workbench Run." This option allows the output of the last run of the workbench (in file WBENCH.MSG) to be redisplayed on the screen if it is available.

The output file is written in the ram drive if the RAM environmental variable has been set as described earlier in this section. If RAM has not been initialized, WBENCH.MSG is written to the current drive. If the output file was created in ram drive it will be lost when the computer is turned off. If this occurs this file will have to be recreated by running the workbench again in order to review the results of the last run.

The fourth option on the workbench menu "Print Last Workbench Run" complements the third option by providing a hardcopy of the output file. Embedded within the output file are form-feed characters which cause most printers to advance the paper to the top of a new page.

The last option on the Normalization Workbench Menu (other than "Quit") is "Save Views as a View Set." This option allows the user to assign a view set name to the group of user views and view sets being used in the workbench. This new view set name can then be used as a shorthand way of identifying all users views which are now included in the workbench views list.

Reports Menu

All hardcopy reports with the exception of the workbench output file are accessed through the reports menu shown in Figure 46.

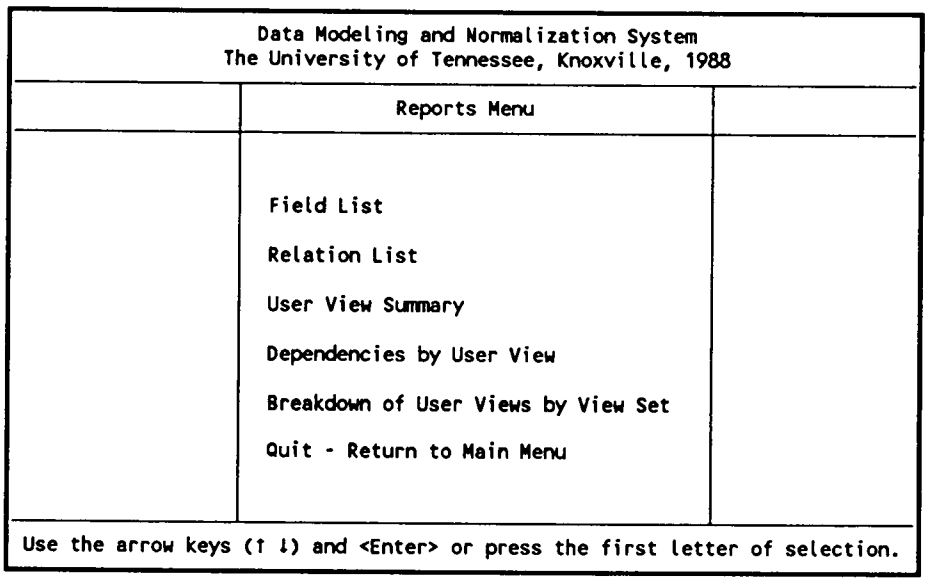


Figure 46. Reports menu.

As can be seen from the reports menu, there are five reports available. The first report is a list of the fields

(data items) for the application. The second report is a list of the relation names to be used after normalization. The third report is a listing of the summary information of all user views for the application. It does not include the dependencies for these user views. The fourth report lists not only user views but also the dependencies associated with each user view. The fifth report shows which user views make up a view set.

VII. NORMALIZATION WORKBENCH OPERATION

This section describes how the normalization workbench of the DMNS operates. A method of visualizing the dependencies, the directed graph, is introduced; graph manipulation functions are described; and a step-by-step synopsis of the normalization process used in the DMNS is provided.

Directed Graphs

A **graph** is a set of nodes (**vertices**) and connections between nodes (**edges**). In a directed graph (or digraph for short), each edge must have a direction from one of the vertices it connects to the other vertex. This direction is usually designated by an arrowhead at the end of the edge touching the vertex to which the edge is directed. The DMNS normalization workbench uses directed graphs to represent data dependencies. Each vertex represents a data item expression which is on either the left-hand side or right-hand side of a dependency. There are two sets of edges, one for functional dependencies and one for multivalue dependencies, which are then superimposed over the vertices to form two different directed graphs with the same set of vertices.

Two edges are **parallel** if they are between the same pair of vertices and in the same direction. An edge is **self-looping** if it comes from and goes to the same edge. A

simple directed graph is a graph which has no parallel edges and no self-looping edges.

In the DMNS, parallel edges between a pair of vertices are allowed as long as they are given different role names (so they can be differentiated from each other). Parallel edges indicate that two different relationships exist between a pair of vertices. Since parallel edges are permitted, the directed graphs in the DMNS are not simple.

Self-looping is obviously not incorrect from a data modeling standpoint since any vertex is FFD with itself, but these trivial self-loops serve no purpose as they do not provide any new information to the model.

DMNS Data Structures

Typically, the vertices in a directed graph are stored in an array or in a linked list. Edges are usually preserved in one of two different structures--an adjacency matrix or adjacency lists. An adjacency matrix is a two-dimensional array in which a specific value in cell (i,j) indicates that there is an edge from vertex i to vertex j. Adjacency lists are linked lists in which each vertex has an index or pointer to a list of edges originating from the vertex.

The DMNS stores vertices in an array and uses an adjacency list for each of the two directed graphs (for FDs and MVDs) it maintains. A structure named VERTEX is

composed of the following elements--FIELDEXPR, NOFIELDS, FDSIN, FDSOUT, VISITED, KEY, FDNDX, and MVDNDX. The array VERTEX is a 200-element array of the structure VERTEX. Components of an array of a structure are identified by the array name followed by a period and then followed by the name of the component of the structure. VERTEX.FIELDEXPR is a character array which holds the data item expression identifying the vertex. VERTEX.NOFIELDS is an integer for the number of data items in the data item expression. VERTEX.FDSIN and VERTEX.FDSOUT are integers for the number of functional dependencies directed into and out of the vertex, respectively. VERTEX.VISITED is an integer field used for marking the vertex in various searches. VERTEX.KEY is a character used to indicate the role a vertex plays in a relation. VERTEX.KEY is set to "P" if the vertex is a primary key, "C" if the vertex is a candidate key, and "S" if the vertex is a secondary key. If the vertex is on the left-hand side of one or more FDs, then the integer VERTEX.FDNDX indicates the first entry in the FD adjacency list of the FDs leaving this vertex. The integer VERTEX.MVDNDX serves the analogous purpose for MVDs leaving the vertex.

The adjacency lists for FDs and MVDs are arrays of a structure named EDGE. FD and MVD are arrays of structure EDGE with dimension 250. The components of the EDGE structure are LHS, RHS, NAME, DEPNO, and NEXT. FD.LHS and

FD.RHS are integers indicating the indexes to array vertex for the vertices on the left-hand side and right-hand side, respectively, of the functional dependency. FD.NAME is the role name of the functional dependency. FD.DEPNO is the unique DMNS internal dependency number of the functional dependency. FD.NEXT is the link to the next functional dependency which has the same left-hand side. If this is the last functional dependency in the linked list, FD.NEXT is set to NONE (defined as -1). Listings of the structures VERTEX and EDGE are included among the program listings in Appendix A (p. 161).

Directed Graph Manipulation Primitives

Directed graph primitives are functions for manipulating the vertices and edges on a graph and provide answers to questions about the same. The primitive functions deal with the counters, indexes, and pointers which may require examination or maintenance when the graph is changed. At a point in the normalization process when a new edge needs to be added to the graph, the primitive for this operation is invoked with the proper arguments. These primitives allow the normalization process to work at a higher conceptual level with features of the graph rather than being cluttered with the details of the graph data structure.

When discussing an element of the vertex array, an element may be identified in one of two ways. The element can be referred to by its index within the array or it can be referred to by the contents of the vertex.fieldexpr variable since these entries (the data item expressions for the vertex of the graph) are unique.

The first two primitives are used to convert between the two possible methods for identifying a vertex. These primitives are called **index(tagarg)** and **tag(indexarg)**.[13] The index(tagarg) primitive returns the index of the vertex for a specified data item expression (tagarg). The tag(indexarg) primitive returns a character pointer (address of a string in the C language) to the data item expression of the vertex for a specified index (indexarg).

The next four primitives operate on vertices. The first, **addvertex(vertexid)**, inserts a new vertex into the graph. The data item expression with which this vertex is to be labeled is passed as argument vertexid. If the vertex already exists, the call to the function is ignored. Initially, added vertices have no connecting edges. The second primitive of the set, **explodekey(key, keyflds)**, breaks a compound data item expression (key) into the individual data items that compose the expression. These individual data items are placed in an array whose address

---

[13]Wayne Amsbury, Data Structures from Arrays to Priority Queues (Belmont, Calif.: Wadsworth Publishing, 1985), pp. 295-298.

is specified by the structure pointer keyflds. The third primitive, **mergekeys(key1, key2, newkey)**, forms a new data item expression containing all unique data items in the data item expressions key1 and key2 and places it in the character array space for the new data item--newkey. The fourth primitive which operates on vertices is **subset(key1, key2)**. Subset(key1, key2) returns a TRUE or FALSE value to indicate whether the first data item expression (key1) is a subset of the second data item expression (key2).

The primitives in the subsequent group operate on the FD directed graph. One of these, **addfd(lhsid, rhsid, depname, depno)**, draws an edge between a pair of vertices on the graph. The vertices on either side of the new edge are identified by the data item expressions lhsid and rhsid. The role name for the new dependency is depname and the dependency number for the new dependency is passed in as argument depno. The function addfd(lhsid, rhsid, depname, depno) is shown as an example of a directed graph primitive in Appendix A (p. 163). Two other primitives, **anyfd(lhsid, rhsid)** and **namedfd(lhsid, rhsid, depname)**, return either the value TRUE or the value FALSE to indicate whether edges exist between a pair of vertices on the graph. Anyfd(lhsid, rhsid) indicates whether any FDs at all exist between the vertices identified by data item expressions lhsid and rhsid. Namedfd(lhsid, rhsid, depname) indicates whether an FD with a specified role name (depname) exists

between the vertices labeled as lhsid and rhsid. The last two primitives, **delallfd(lhsid, rhsid)** and **delfd(lhsid, rhsid, depname)**, remove edges from the graph. Delallfd(lhsid, rhsid) deletes all edges between a pair of vertices labeled as lhsid and rhsid. Delfd(lhsid, rhsid, depname) deletes one edge with a specific role name (depname) between the pair of vertices labeled lhsid and rhsid, respectively.

The last group of primitives operates on the MVD directed graph. These functions are analogous to the functions for FDs. **Addmvd(lhsid, rhsid, depname, depno)** draws an edge between a pair of vertices labeled as lhsid and rhsid on the graph. **Anymvd(lhsid, rhsid)** and **namedmvd(lhsid, rhsid, depname)** return a TRUE or FALSE value to indicate whether edges exist between a pair of vertices labeled as lhsid and rhsid on the graph. Anymvd(lhsid, rhsid) indicates the existence of any MVD between lhsid and rhsid. Namedmvd(lhsid, rhsid, depname) indicates an edge with a specified role name (given by depname) exists between lhsid and rhsid. Primitives **delallmvd(lhsid, rhsid)** and **delmvd(lhsid, rhsid, depname)** remove edges from the graph. Delallmvd(lhsid, rhsid) deletes all MVDs from between a pair of vertices, while delmvd(lhsid, rhsid, depname) deletes a specific MVD with

the role name depname from between a pair of vertices labeled as lhsid and rhsid.

All function names used in the DMNS workbench and the data types of their argument lists are enumerated in Appendix A (p.162). In further discussions involving the directed graph primitives, the function arguments will be left off unless they are needed to clarify what function is being performed.

DMNS Normalization Synopsis

Construction of the directed graphs begins in the merge phase of the workbench. When each dependency is read in, the addvertex() function is called for both its left- and right-hand side data item expressions. If addvertex() finds that either of these vertices already exist in the graph, then the present vertex is left unchanged. After the vertices have been added, either addfd() or addmvd() is called as appropriate. Here again, if the edge already exists in the graph, nothing is done.

Merge also looks for two situations which require it to add some missing information to the model. The first situation is a vertex with a compound data item expression. Under the assumption that each individual data item in a compound data item expression must stand by itself somewhere in the data model, these expressions are broken apart with explodekey() and added as individual vertices. Then

between each of the component vertices and the original compound vertex, an FD to the component is added and an MVD to the compound vertex is added. The role name given to each of the added dependencies is "DMNS added." If the vertices and/or dependencies already existed, nothing is changed. For example, if a vertex whose tag() is A+B is added to the workbench, then the following steps must be taken:

1. add vertex A,
2. add vertex B,
3. add A+B ⟶ A (DMNS added),
4. add A+B ⟶ B (DMNS added),
5. add A ⟶▸ A+B (DMNS added), and
6. add B ⟶▸ A+B (DMNS added).

These modifications are illustrated in Figure 47.

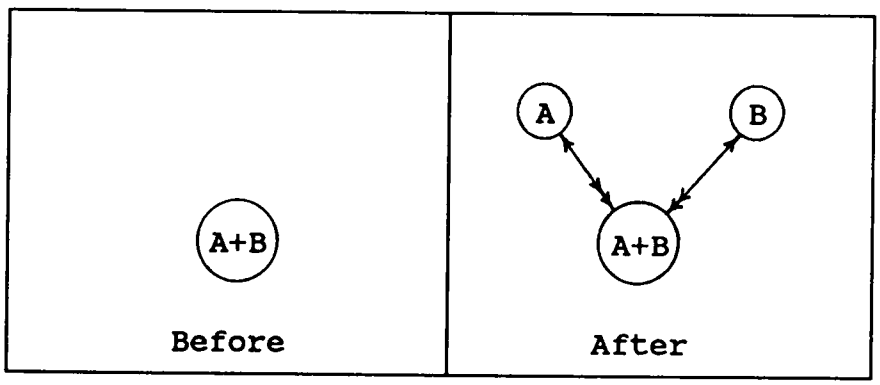


Figure 47. Explosion of a compound data item expression.

If a later attempt is made to add a dependency between two vertices where a "DMNS added" dependency currently exists, the role name of the new dependency replaces the old DMNS generated one.

The second situation occurs when a multivalue dependency is added from some Vertex A to another Vertex B and there is already a multivalue dependency from Vertex B to Vertex A. Such bidirectional multivalue dependencies can only be represented in the relational model by a new relation whose key is the merger of the data elements in Vertices A and B. This new vertex will be an intersection between Vertices A and B. The multivalue nature of the association between A and B is maintained because there will be multivalue dependencies from A to A+B and from B to A+B. In summary, the following steps would be taken:

1. delete A $\longrightarrow\!\!\!\!\blacktriangleright\!\!\blacktriangleright$ B,
2. delete B $\longrightarrow\!\!\!\!\blacktriangleright\!\!\blacktriangleright$ A,
3. add vertex A+B,
4. add A+B $\longrightarrow\!\!\!\!\blacktriangleright$ A (DMNS added),
5. add A+B $\longrightarrow\!\!\!\!\blacktriangleright$ B (DMNS added),
6. add A $\longrightarrow\!\!\!\!\blacktriangleright\!\!\blacktriangleright$ A+B (DMNS added), and
7. add B $\longrightarrow\!\!\!\!\blacktriangleright\!\!\blacktriangleright$ A+B (DMNS added).

These modifications are illustrated in Figure 48.

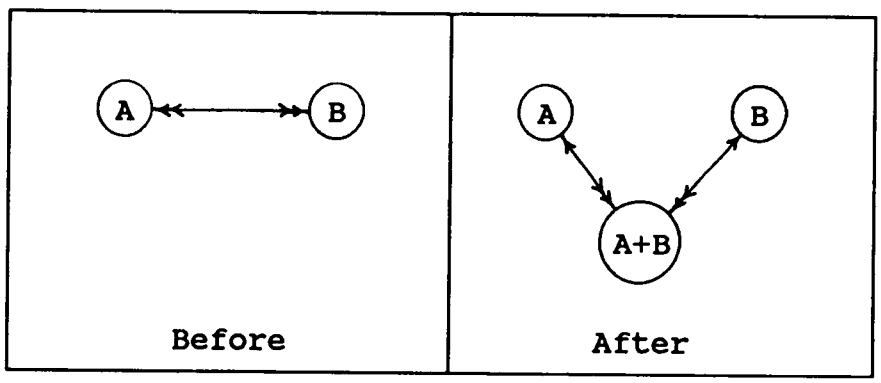


Figure 48. Elimination of two-way MVDs.

As with the previous situation, the role name for the added dependencies is "DMNS added." If a dependency from a user view matches one of those added, the role name of the "DBMS added" dependency will be replaced with the one from the user view.

The steps just described, which take place in the merge phase, are performed for all dependencies in all user views to be synthesized. After the last user view has been merged, the normalization phase of the workbench begins. The normalization phase is actually a series of short steps which either modify the workbench contents or check for some condition to which the data modeler should be alert. These steps are listed in Figure 49.

1. Find parallel edges of dissimilar types.
2. Remove transitive dependencies.
3. Match vertices with RELATION.DBF file.
4. Locate candidate keys.
5. Assign vertices which functionally define another vertex as primary keys.
6. Locate isolated attribute vertices and assign them to be primary keys.
7. Find and assign unknown primary key to primary key reverse associations.
8. Remove individual components of exploded keys which are not determinants of any other vertex.
9. Warn of intersecting attributes.
10. Warn of parallel edges.
11. Warn of cycles in functional dependencies.
12. Check secondary key accesses for feasibility.

Figure 49. Directed graph normalization summary.

Each of these steps will be described in the order in which they take place.

Before the workbench can continue, any parallel edges of dissimilar dependency types must be eliminated; that is, a pair of vertices for which there exists a functional dependency and a multivalue dependency in the same direction. If this condition exists, then the functional dependency between these two vertices is deleted.

The second step in the normalization process is the location and removal of transitive functional dependencies. Transitive dependencies are those which can be inferred from other dependencies. Removal of these dependencies results in change to the data model as shown in the example of Figure 50.

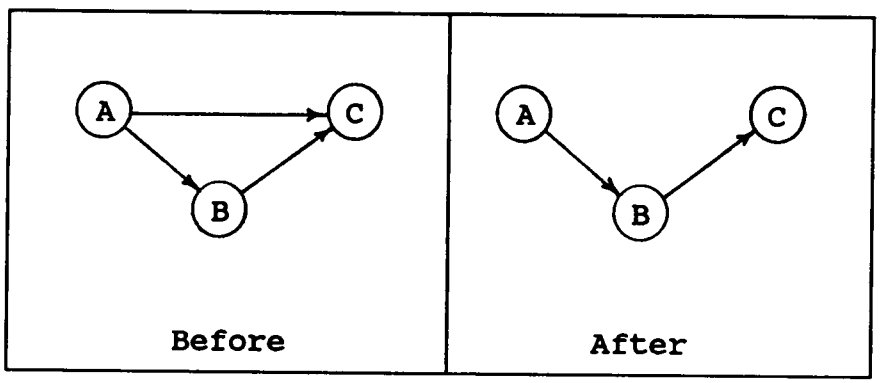


Figure 50. Removal of transitive functional dependencies.

Transitive dependencies are found through the use of a breadth first search. Selecting as a source the vertices which have more than one functional dependency directed into

them, the functional dependencies are backtracked. Each vertex is marked as it is visited. If a vertex which has already been visited is encountered, then a transitive dependency exists. The transitive dependency is directed from the vertex now being visited back to the source vertex. This source code for this part of the normalization process is shown as an example in Appendix A (p. 164).

The third step in the normalization process is the matching of user-defined relations with vertices. This is the first of three steps in which primary keys, and hence relations, are identified from among the vertices in the graph. The field RELIDFLDS from the RELATION.DBF file is compared with the data item expression tags for the vertices in the workbench. If the tag of a vertex is found to match an entry in the RELATION.DBF file it will be considered to be the primary key of a relation, regardless of the vertex's placement within the graph. This feature of the DMNS is not truly a step in normalization; it is provided for the convenience of the data modeler, so that descriptive labels and user-defined file names may be assigned to the data base design's relations. Notice that variations in the use of this option are also discussed later when describing candidate key assignment and in the removal of individual components of exploded keys which are not determinants of any other vertex.

Also, note that this option can be used as an override mechanism to force the workbench to create a relation where the DMNS would not otherwise decide that one existed. Used in this way, deviations from fourth normal form can be forced on the data base design, but it is a powerful feature which the modeler must use with care.

The fourth step from Figure 49 is the location of candidate keys. Candidate keys are two or more data item expressions which could each be the primary key of the relation in which they are located. If all dependencies have been entered in the workbench, then true candidate keys will functionally define each other and will also be determinants of all the same attributes. In the DMNS, since some dependencies may be missing from a data model, it must be taken for granted that if two vertices functionally define each other that they also are determinants of the same attributes. For example, in Figure 51, vertices A and B functionally determine each other, but do not have exactly the same attributes. The DMNS will assume that A and B are candidate keys of the same relation and that they DO have the same attributes.

When a pair of candidate keys is found, one must be chosen as the primary key of the relation. If the workbench has no indication from the modeler as to which should be the primary key, then it will arbitrarily select one of the candidate keys to be the primary key. If it does so, it

will notify the modeler which candidate key it has chosen to be the primary key. The modeler can force the workbench to choose the candidate key desired to be the primary key by adding a record in the RELATION.DBF for that vertex.

When a primary key is selected, either by relation matching or arbitrarily, the dependency from the alternate key to the primary key is removed; and the primary key is made the determinant of all the attributes which the alternate key previously defined. An example of this is shown in Figure 51.

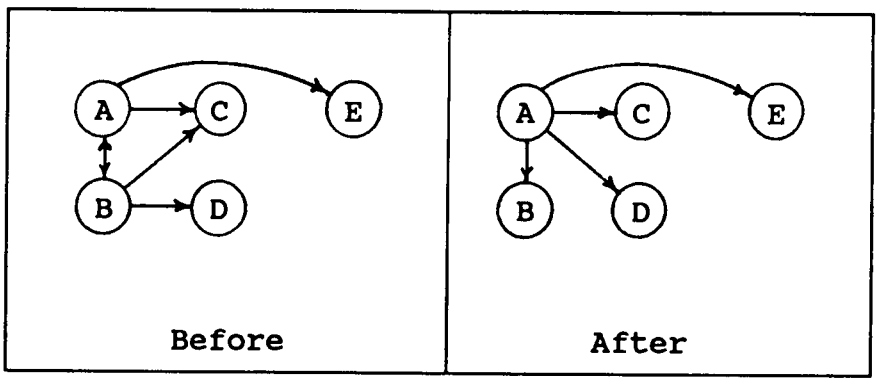


Figure 51. Rerouting of alternate key (B) dependencies to the primary key (A).

After the process shown in Figure 51 is complete, all the functional dependencies of the alternate key are intentionally omitted from the workbench. The DMNS knows that these dependencies actually still exist because it has marked the alternate key in the VERTEX array. Removing these dependencies from the workbench's list of functional

dependencies simplifies both further normalization steps and the grouping of vertices into relations during the relational design steps.

It is also possible for the data modeler to declare both candidate keys to be the primary keys of separate relations by putting both data item expressions in the RELATION.DBF file. In this case, no dependencies will be changed. If any vertex is functionally defined by both of the primary keys in question it will be considered an intersecting attribute as described later in the normalization process.

Step 5 from the normalization phase is to mark any vertices which have not already been matched to a relation from the RELATION.DBF file but which functionally define some other vertex as primary keys. Primary keys are very important to the relational data base design. Every relation must have a primary key; and in the data model, the other attributes in the relation are organized around them. Primary keys found in this step will all be named "Unknown" but will, of course, have primary key data item expressions which are unique from those of all other relations.

The third and last way a primary key can occur in the workbench is found in Step 6 of the normalization phase. Isolated attributes are those vertices which have no FDs entering or leaving them. Isolated attributes will become

unary (single data item) relations. They may have one or more MVDs coming in or going out.

Once all vertices which are primary keys have been located, normalization Step 7 examines which primary keys have a dependency between them in one direction but have no dependency indicated in the reverse direction. The dependencies existing in both directions between two primary keys must be known in order to know how to join tables. This search does not apply to primary keys and their attributes within the same relation, but between primary keys of different relations.

When the reverse association between two primary keys is not known, the dependency assumed in the reverse direction will be the opposite type of the dependency which is known. That is, if the known dependency is an FD then the reverse dependency will be an MVD. If the known dependency is an MVD then the reverse dependency will be made an FD. This new dependency has a role name of either "Assumed FD" or "Assumed MVD" as appropriate. Opposite dependency types are assumed as they cause less disruption to the data model than always assuming either an FD or an MVD. If the known dependency were an FD and an FD were assumed in the unknown direction, then a candidate key situation is formed. In the case where the dependency in the known direction is an MVD, if a multivalue dependency were to be assumed in the unknown direction, then a

bidirectional MVD would occur. Bidirectional MVDs require intersection relations and have already been eliminated from the data model during the merge phase, and any reappearance at this stage in the normalization process would be inappropriate.

The result of finding an unknown reverse association is depicted in Figure 52. In Figure 52, vertices A and B are the primary keys of two different relations with an MVD from A to B given in a user view. Since the reverse association from B to A is unknown, a dependency must be created to complete the model. This dependency will be an FD from B to A with a role name of "Assumed FD."

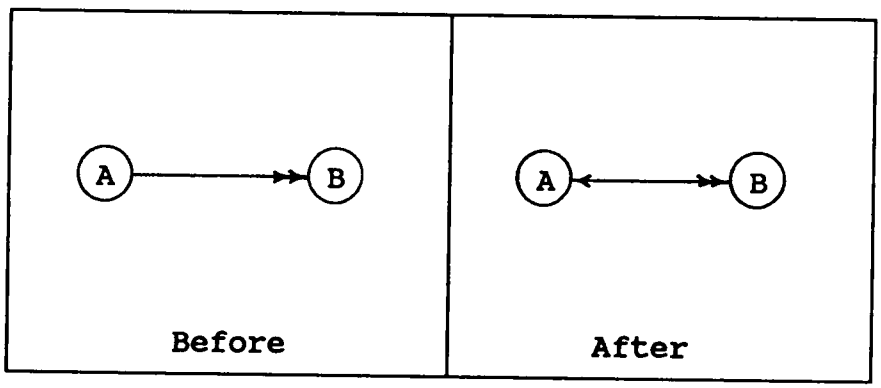


Figure 52. Creating an FD for an unknown reverse primary key association.

A hanging key is a unary relation formed from a data item which is an individual component of an exploded compound data item expression. Hanging keys are root keys which do not have any attributes. The only other vertex to

which a hanging key is connected is the compound data item expression of which it is a component data item. That is, it "hangs" from its bigger relative.

Step 8 from Figure 49 finds the hanging keys and removes them unless the modeler wishes to retain particular ones. In two situations the modeler may choose to retain a hanging key. The first is if this unary relation is to be used to validate the domain of this data item in the relation with the compound data item expression as its key. The second is if the data modeler decides that at some later point it is likely that the hanging key will have some attributes, although one is not present in the current model, it would be a good idea to make a single data item relation of this data item.

An example will help clarify the situation. A relation TEXTSREQ (which lists textbooks required for a course) has the compound data item expression of COURSENO+TEXT as its key. This key was exploded during the merge phase as described earlier creating vertices TEXT and COURSENO and connecting edges between these vertices and COURSENO+TEXT. Assume for this example that vertex COURSENO functionally defines some other vertex, and so, is not a hanging key. However, the data model does not indicate that the exploded component TEXT functionally defines any other vertex. Vertex TEXT is then a hanging key.

Should TEXT be left in the graph and become a single data item relation or should it be deleted as a vertex? If it is desirable to ensure that the names of the texts entered in the TEXTSREQ relation are within the proper domain, then TEXT should become the single data item in a unary relation named TEXTBOOKS (or some other descriptive name). The TEXTBOOKS relation can then supply the domain validity check needed for the names of the texts entered in TEXTSREQ. Even if domain validity checks are not needed, the unary relation TEXTBOOKS with the single field TEXT might be created if the data modeler (or data base administrator) feels that it is likely that at some point information about text books, such as price, publisher, number of pages, etc., might need to be retained. If neither of the prior cases applies, then removing the TEXT vertex, the FD from COURSENO+TEXT, and the MVD to COURSENO+TEXT, all of which were created when the COURSENO+TEXT vertex was exploded, is the preferred course of action. Removal of an unwanted hanging key TEXT is shown in Figure 53.

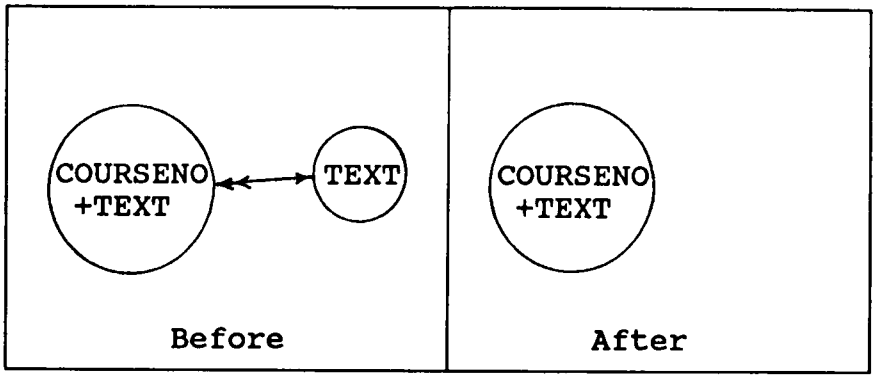


Figure 53. Removal of an unwanted hanging key.

How does the data modeler indicate whether or not the DMNS is to keep a single data item relation? Once again, the data modeler's control mechanism is the RELATION.DBF file. For the example above, if the DMNS finds the hanging key such as TEXT in Figure 53 it will check in the RELATION.DBF file to see if TEXT is listed as the primary key of a relation. If TEXT is in the RELATION.DBF file, a unary relation will be created for TEXT; if not, the TEXT vertex will be removed and its two connecting edges to COURSENO+TEXT are removed.

The prior step is the last one involving direct DMNS modification of the data model. The last four steps are checks for anomalies in the data model which may lead the data modeler to go back and revise some user views in order to eliminate them. The DMNS will develop a relational data base design even if these anomalies are not removed.

Step 9 is a search for intersecting attributes. An intersecting attribute is functionally defined by primary

keys from more than one relation. Rather than guess which relation the attribute should be placed, the workbench assigns the attribute to both relations. The modeler is warned of this condition, however, so that corrective action may be taken.

Intersecting attributes normally arise due to an error in the data model. However, assigning the intersecting attribute to both relations whose primary keys functionally define it, does provide the data modeler with a mechanism with which to force a common field into two relations. This intersecting attribute in both relations allows the two relations to be joined for a query when they might otherwise not be "joinable." Such a capability is present in other normalization software, but the author does not foresee why it is needed. The penalty for introducing an intersecting attribute is that the schema is no longer in the fourth normal form.

Step 10 is a test for parallel edges. Parallel edges may exist if they are of the same dependency type and have different role names. Cases where parallel edges may be used are when there is more than one association between a pair of vertices. Take the case of Vertices LETTER and POSTOFFICE. There could be two FDs from LETTER to POSTOFFICE, where one FD is labeled "RECEIVED-AT" and the other is labeled "DELIVERED-TO."

This provides a convenience in modeling but entails a problem when the relational data base from the schema is designed. The relation for LETTER must have a POSTOFFICE field to indicate where the letter was received and another POSTOFFICE field to indicate where the letter was delivered from, since this is often not the same post office. However, a relation may not have two fields with the same name, so this design is not implementable as just described.

The solution lies in the hands of the data modeler. The data modeler can create what are sometimes known as **subentities** or **ISA** relationships.[14] Subentities are subsets of other entities. For example, MANAGERS might be a subset of EMPLOYEES, or RECEIVINGPOSTOFFICE might be a subentity of POSTOFFICES. Subentities provide convenient ways to indicate multiple relationships with other entities or to store data specific to items that fall into the subentity without saving these fields for the entity as a whole. The data modeler must eliminate the duplicate naming problem by creating subentities to allow unique names to be included in every relation.

Step 11 in the normalization phase is to locate cycles in functional dependencies. A **cycle** in a directed graph is a circuit of edges following the direction of the edges which returns to the original vertex. In the DMNS, cycles

---

[14]Jeffrey D. Ullman, Principles of Data Base Systems (Rockville, Md.: Computer Science Press, 1982), pp. 13-14.

are more loosely defined. Cycles in the DMNS must include at least three edges (must be larger than a connected pair of vertices), must be all FDs, and do not consider the direction of the edges to be important. The DMNS locates cycles by use of a recursive depth-first search.[15] Whenever a cycle is found, the vertices in the cycle are marked so that another search for a cycle does not begin at one of these vertices, thus preventing the exact same cycle warning message from appearing twice in the printout but allowing the vertex to appear in a different cycle, if there is one. Why are cycles a concern? Commonly a cycle in a data model is indicative of omitting a functional dependency from the model. Consider the cycle shown in Figure 54.

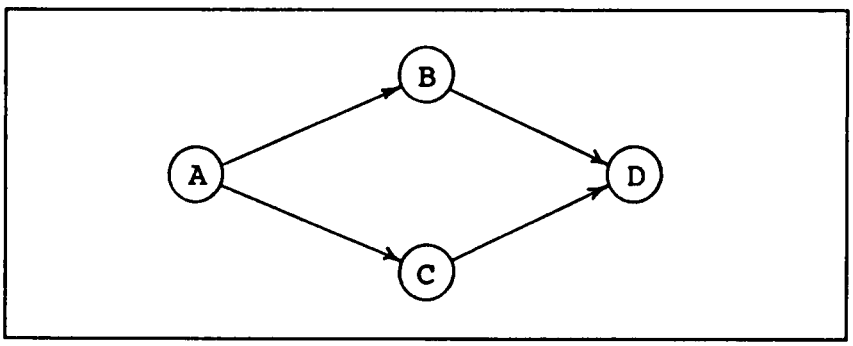


Figure 54. A cycle in a data model.

When missing dependencies are added to a data model they often cause another dependency to become transitive and

[15]Narsingh Deo, Jurg Niebergelt, and Edward M. Reingold, Combinatorial Algorithms: Theory and Practice (Englewood Cliffs, N.J.: Prentiss-Hall, 1977), pp. 327-330.

drop out of the model, thus eliminating the cycle. In the cycle in Figure 54 the missing dependency might be B——▸C or C——▸B. What would happen to the cycle if it is decided that one of these dependencies does exist and is then included in the workbench? If the dependency B——▸C is inserted, then A——▸C and B——▸D are transitive and will be removed. When these dependencies are removed, the cycle disappears.

As mentioned previously, a secondary key is not actually a key; it is an attribute in a relation upon which an index file may be built in order to decrease the access time for some transaction defined by one of the user views in the workbench. The last step in the normalization process is to determine if all secondary key accesses are feasible. In the directed graph, secondary keys are indicated by an MVD from some attribute to either the primary key of its own relation or to a field of some other relation. If the MVD is directed to a data item in a relation other than the relation which the attribute is in, the relation which it is in and the relation to which it is pointing must be joinable. To be joinable there must be appropriate combinations of foreign keys and primary keys for these two relations to be joined either directly with each other or by chaining with other relations. If the two relations cannot be joined then the transaction defined by the user view with the MVD in question cannot work properly.

In order to ascertain whether the relations in question can be joined, a depth-first search is performed from the primary key of the relation where the MVD originates, looking for the primary key of the relation where the MVD terminates. If the primary key of the relation where the MVD terminates is found, the search terminates successfully and the secondary key is feasible. If the search goes to completion without finding the target primary key, then the relations cannot be joined and a diagnostic message to that effect is printed indicating that the relations cannot be joined.

Conversion of Normalization Violations to 4NF

Just presented was the process to which user views entered in the DMNS are subjected. Has the original schema really been converted to a relational data base design which conforms to 4NF? This section will examine five specific instances of violation of various normal forms and describes the transformation made to 4NF in each case.

To discuss the DMNS' normalization process it is useful to discuss the transformation of user views entered into the system as though they were from an existing file structure or relation. More often than not such an existing file will not be source of the user view. But, identical user views can come from creating a user view of a report, an input form, or an existing file structure or relation in an older

data base or application program. The reason that it is handy to talk in terms of an old relation is that if there were truly no data base existing before, then the concepts of relations and normalization might not apply at all to the old environment. Therefore, the following discussion deals with transforming old relations which have known levels of normalization into relations in the 4NF. The process which the DMNS performs would be identical, however, had the user views been from an all-manual system undergoing its first computerization.

Before presenting the processes which normalize to fourth normal form for various instances of lesser normalization one final note is required. In the discussion of normalization in Section IV, normalization was undertaken in an interative manner. A relation would be transformed from one level of normalization to the next level up. The DMNS does not work progressively from unnormalized files to 1NF to 2NF to 3NF to BCNF and finally to 4NF; instead it works directly towards the 4NF.

The first case of interest is when a file structure is modeled that is unnormalized. File structures that are unnormalized have repeating fields within their records. Repeating fields reserve space within the record for a programmer-defined maximum number of entries much like an array does in memory. Consider the COBOL record (file) structure below:

```
RECORD NAME IS EMPLOYEE-RECORD
  01    EMPNO PICTURE "9(5)"
  01    EMPNAME TYPE CHARACTER 25
  01    EQUIPNO PICTURE "9(6)" OCCURS 50 TIMES
  01    PHONENO "9(7)".
```

In the record structure above, EQUIPNO identifies a piece of equipment assigned to an employee and is a repeating field. No piece of equipment is assigned to more than one employee. In an individual employee's record, there can be from 0 to 50 EQUIPNOs listed for the employee. Conceptually, this record structure can be drawn as in Figure 55.

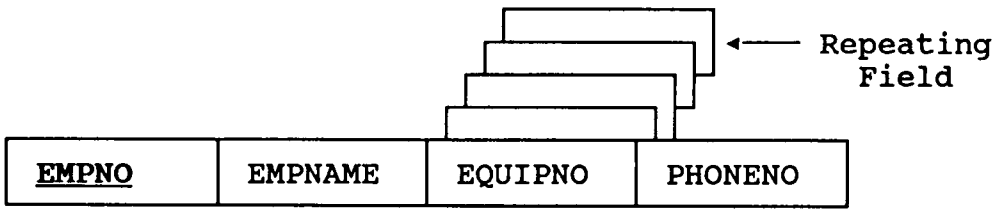


Figure 55. A COBOL record structure which is unnormalized.

Again, the file structure of Figure 55 is unnormalized because it contains a repeating field in which to identify the items of equipment for which each employee is responsible. Each employee appears only once in the file. If modeled properly, the user view for the data dependencies from the record structure in Figure 55 would appear as in Figure 56.

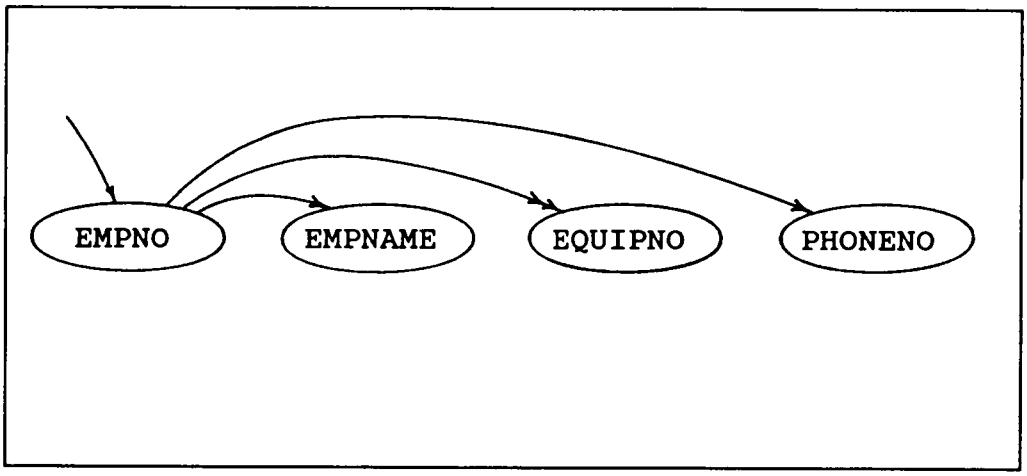


Figure 56. User view for the record structure of Figure 55.

If the user view from Figure 56 were to be the only one entered into the DMNS, what would the DMNS do with it? In the normalization phase, the DMNS decides that the data item EQUIPNO is an isolated attribute since it has no FDs in or out and should, therefore, be treated as a primary key. EMPNO is found to be a primary key because it is a determinant of EMPNAME and PHONENO. Next, the DMNS discovers that the reverse associations from the primary key EQUIPNO and the primary key EMPNO is unknown. Since the known dependency is an MVD from EMPNO to EQUIPNO, the DMNS assumes that an FD exists from EQUIPNO to EMPNO. The end result of the normalization process is that two relations are created, one for employees and another for equipment as shown in Figure 57.

EMPNO	EMPNAME	PHONENO

EQUIPNO	EMPNO

Figure 57. Relational design of employee/equipment user view.

The two relations shown in Figure 57 are in the fourth normal form. This is an improvement over the design shown in Figure 55 because the employee reponsible for a piece of equipment can be found by referencing the EQUIPNO in the lower relation and there is no limit on the number of pieces of equipment which can be listed for a particular employee.

Before examining the second example of normalization from lower levels to 4NF in the DMNS, some insight will be provided into what can go wrong in the cannonical synthesis of a logical data model. Suppose that a COBOL record structure which is identical in format but which has different field names from the previous record had been given as shown below:

```
RECORD NAME IS SUPPLIER-RECORD
  01   SUPPLIER PICTURE "9(5)"
  01   SUPPNAME TYPE CHARACTER 25
  01   PARTNO PICTURE "9(6)" OCCURS 50 TIMES
  01   PHONENO "9(7)".
```

Although the format of SUPPLIER-RECORD above is the same as the EMPLOYEE-RECORD the semantic rules are

different. Suppliers can supply many parts and a part may be supplied by more than one supplier. The user view for this record structure looks just like the user view of Figure 56 and is shown in Figure 58.

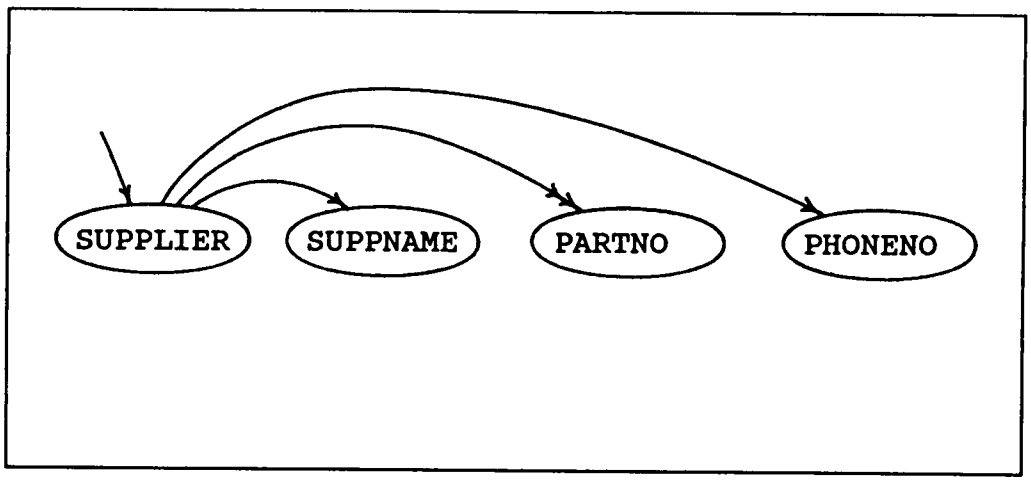


Figure 58. User view for the second COBOL record structure.

When entered into the DMNS, the user view in Figure 58 would be treated just as the prior example. PARTNO would be found to be an isolated attribute and would be treated as a primary key. SUPPLIER is a primary key because it is a determinant. The reverse association from PARTNO to SUPPLIER is unknown and because the known association is a MVD, the DMNS guesses that it is an FD. The relational design generated by the DMNS is shown in Figure 59.

SUPPLIER	SUPPNAME	PHONENO

PARTNO	SUPPLIER

Figure 59. Relational design of supplier/part user view.

Although this example underwent the same process as the prior example, this design is wrong. Since, semantically, the associations between PARTNO and SUPPLIER are bi-directional MVDs, the lower relation in Figure 59 should have PARTNO+SUPPLIER as the concatenated primary key instead of PARTNO by itself.

Where did this design go wrong? Although there is nothing wrong with the user view presented, this single user view is not sufficient to model properly the relationships between parts and suppliers. The DMNS could only guess what the unknown association from suppliers to parts is, and guessed that it is an FD. It is not. Another user view would be required (or a dependency added to the user view provided) to indicate the proper association from parts to suppliers.

Another problem with the design of Figure 59 is that it will not be stable. At some time there may be attributes for both PARTNO by itself and for PARTNO+SUPPLIER which are important to keep in the application. Clearly, there are not two relations in the model with these primary keys in

which these attributes can be correctly placed. If user views with an attribute(s) for PARTNO had been introduced into the model and a user view with the correct association from parts to suppliers the two necessary relations would have been created and design would be stable.

With the dependencies it was given and the assumed dependencies, the DMNS created a design in 4NF. However, this design is not an accurate data model because of the incorrect assumption as to the reverse association between SUPPLIER and PARTNO. Normalization does not make a data model correct if the semantic rules on which the model is based are incomplete or incorrectly modeled.

Returning to the example cases of normalization, the second case to be considered is the modeling of data which is in first normal form but not in the second normal form. For this example, consider the relation shown in Figure 60.

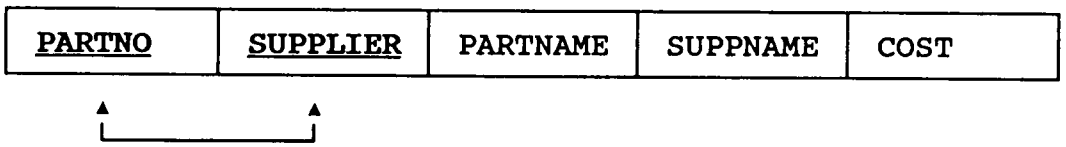


Figure 60. A relation which is not in 2NF.

In order to verify that the above relation is not in the second normal form it is necessary to examine the data dependencies for the data items in the file. These dependencies are shown in Figure 61.

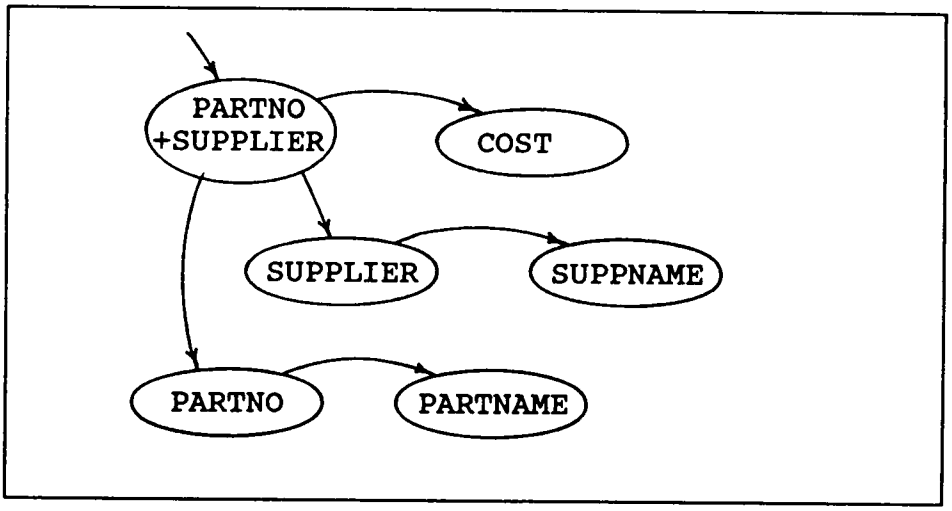


Figure 61. User view for parts file of Figure 60.

To be in the second normal form all fields in a relation must be fully functionally dependent on the primary key of the relation. The primary key for the relation is PARTNO+SUPPLIER. Obviously, neither PARTNAME nor SUPPNAME are fully functionally dependent on PARTNO+SUPPLIER, so this relation is not in the 3NF.

How does the DMNS rearrange the relation structures for the dependencies above so that the resulting relations are in fourth normal form? For the user view shown in Figure 60 all the work performed by the DMNS is done during the merge phase. When the concatenated key of PARTNO+SUPPLIER is seen by the system, it is exploded into its components and these dependencies are added:

PARTNO+SUPPLIER ——▸ PARTNO,
PARTNO+SUPPLIER ——▸ SUPPLIER,
PARTNO ——▸▸ PARTNO+SUPPLIER, and
SUPPLIER ——▸▸ PARTNO+SUPPLIER.

This is all that the DMNS does with the schema before it produces a relational data base design. The design based on the user view of Figure 61 is shown in Figure 62.

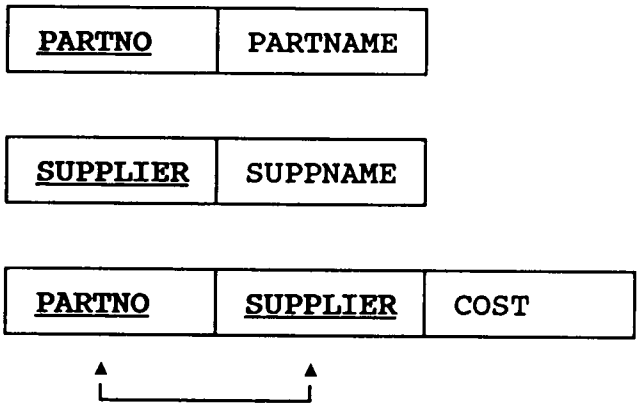


Figure 62. Fourth normal form design for Figure 61.

The data base design of Figure 62 is in fourth normal form. All attributes in every relation are fully functionally dependent on the primary keys of the relations.

Case three is for a relation which is in second normal form but not in third normal form. To be in second normal form all attributes of a relation must be FFD on the primary key. To be in third normal form these FFDs cannot be transitive functional dependencies. Consider the relation structure shown in Figure 63.

EMPNO	EMPNAME	SALARY	DEPTNO	DEPTNAME

Figure 63. A 2NF relation which is not in 3NF.

Note that EMPNO is the primary key of the file above, which implies that each employee, since he/she can appear in the file only once, can be in only one department. The dependencies for the relation of Figure 63 are shown in Figure 64.

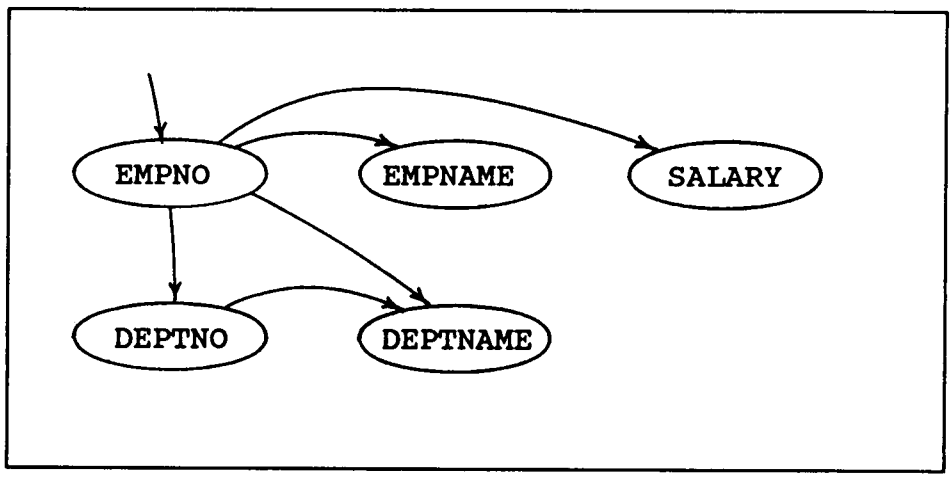


Figure 64. User view for relation of Figure 63.

All attributes are functionally dependent on the primary key. However, DEPTNAME is also transitively dependent on EMPNO. When the DMNS encounters the user view above, it determines that the dependency EMPNO⟶DEPTNAME is a transitive dependency and deletes it. Next, since EMPNO and DEPTNO are both determinants, they are both marked

as primary keys of relations. These two primary keys are associated with each other because EMPNO——▸DEPTNO, but the association in the reverse direction is unknown. The DMNS assumes that the reverse association is DEPTNO—▸▸EMPNO. After these modifications have been made, the dependencies in the DMNS workbench are shown in Figure 65.

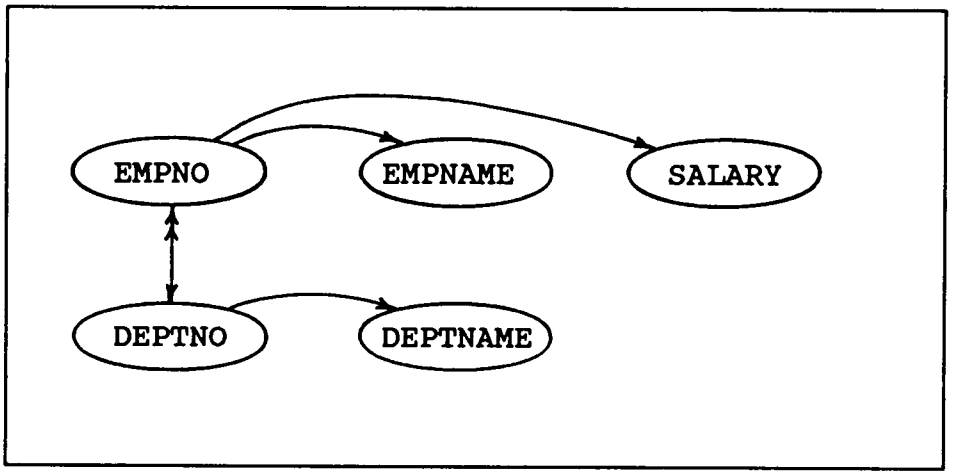


Figure 65. DMNS workbench after modification.

The DMNS' relational design for the dependencies shown in Figure 65 is shown in Figure 66. This design is in the fourth normal form after all the normalization steps have been applied to the dependencies.

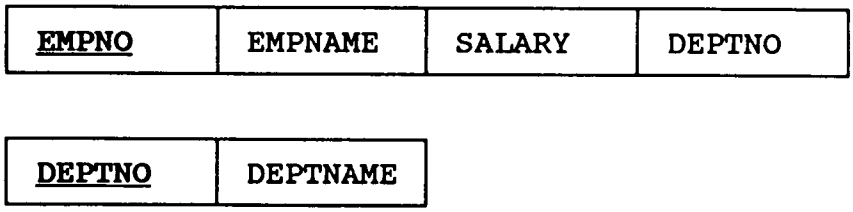


Figure 66. Fourth normal form design for Figure 65.

The fourth case of transformation of a relation which violates lower levels of normal forms into a proper fourth normal form design to be considered here is for a file which is in the 3NF but violates the BCNF. Consider the relation in Figure 67.

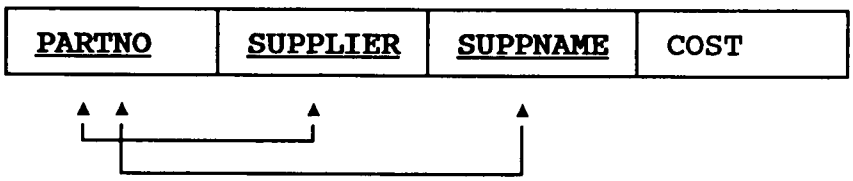


Figure 67. A relation which is not in BCNF.

This file has two candidate keys. They are PARTNO+SUPPLIER and PARTNO+SUPPNAME. To show this relation is in the 3NF, consider that although SUPPNAME is functionally defined by SUPPLIER, that, in this relation, SUPPNAME is part of one of the candidate keys of this file. 3NF requires that attributes be FFD on the entire key, but does not require that a component of a key cannot be functionally defined by some other component of a key. Thus the relation of Figure 67 does not violate the 3NF.

Since the two candidate keys have an element (PARTNO) in common, they are known as overlapping candidate keys. The user view for the relation of Figure 67 is shown in Figure 68.

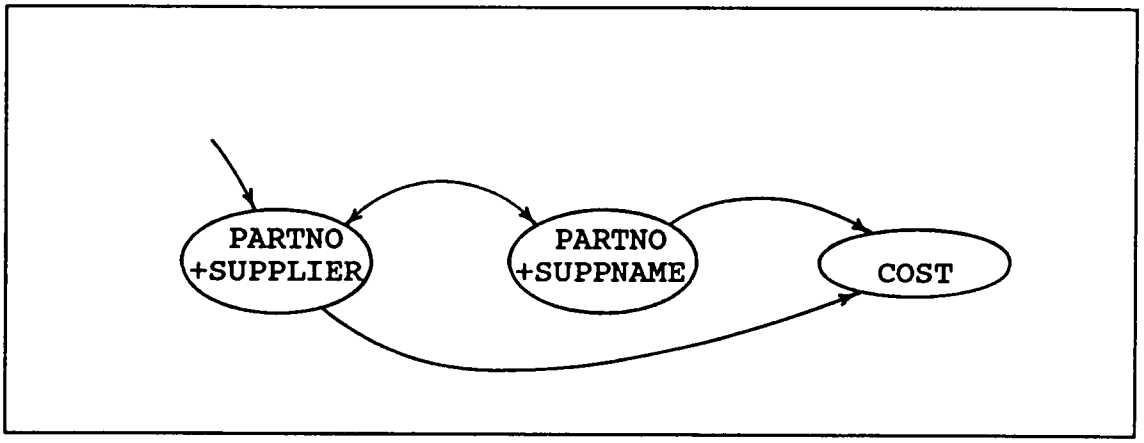


Figure 68. User view of file in Figure 67.

After trivial dependencies are added in the merge phase, the dependencies related to this relation inside the workbench are shown in Figure 69.

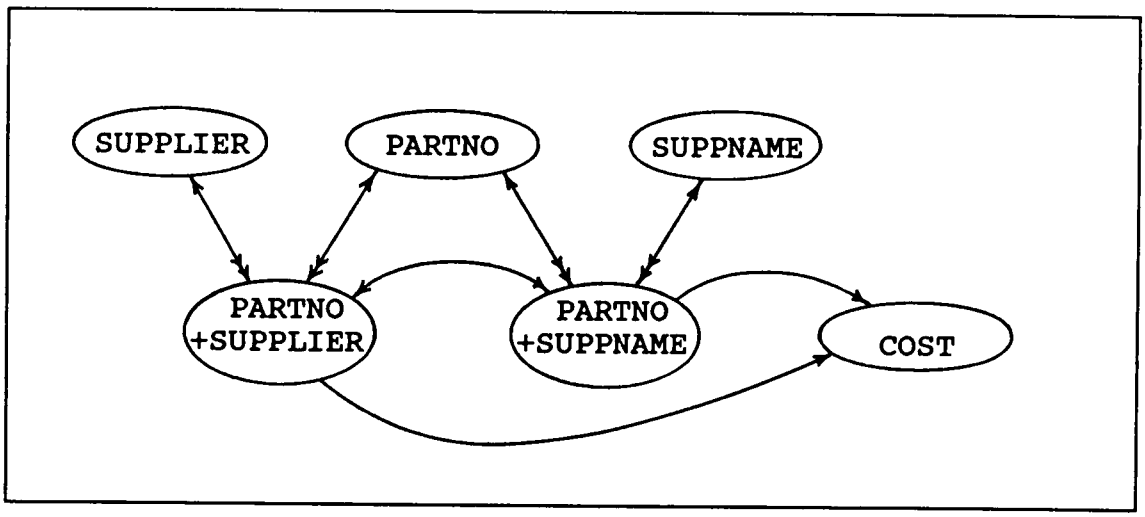


Figure 69. Dependencies after merge phase.

As described in Section IV, the overlapping candidate keys shown above constitute a violation of BCNF. How does the DMNS resolve this problem? Consider the mirror image dependencies:

PARTNO+SUPPLIER ——▸ PARTNO+SUPPNAME and
PARTNO+SUPPNAME ——▸ PARTNO+SUPPLIER.

These dependencies indicate that PARTNO+SUPPLIER and PARTNO+SUPPNAME mutually identify each other. But PARTNO always identifies PARTNO, so cannot PARTNO be removed from both sides and the dependencies still hold for the remaining data items? Yes, it can. If PARTNO were removed from both data item expressions, then SUPPLIER and SUPPNAME identify each other and, therefore, will be candidate keys but not in the same relation with PARTNO. Recognizing that SUPPLIER——▸SUPPNAME and SUPPNAME——▸SUPPLIER must both

hold, those dependencies can be added to the data model and one of the candidate keys of the original relation; that is, PARTNO+SUPPLIER or PARTNO+SUPPNAME may be dropped and the other candidate key will be the primary key of the original relation. If PARTNO+SUPPLIER is chosen as the primary key of the original relation, then the modified dependencies would appear as in Figure 70.

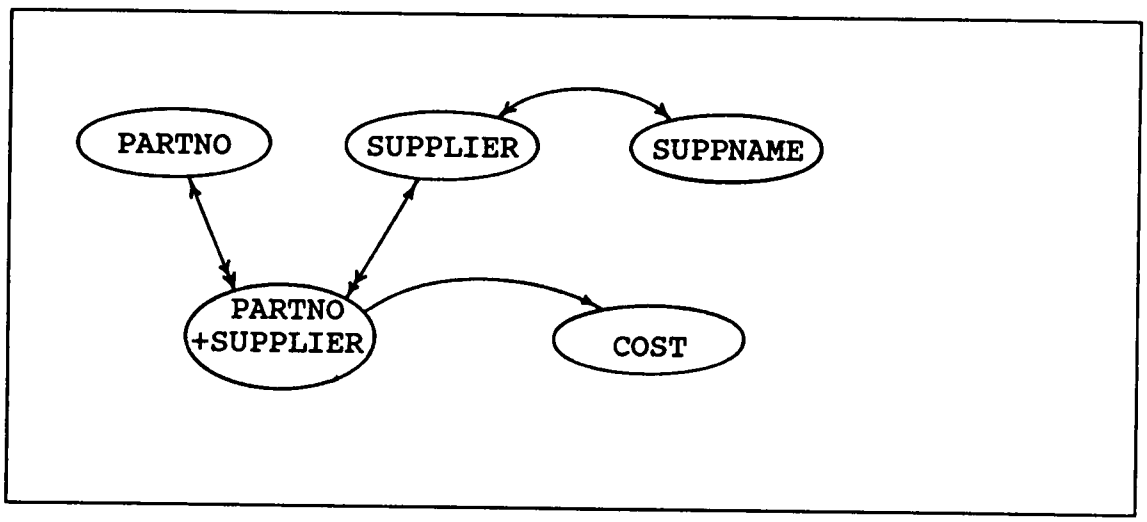


Figure 70. Elimination of an overlapping candidate key.

The resulting relational data base design for the dependencies in Figure 70 is shown in Figure 71 using the assumption that SUPPLIER is chosen as the primary key of the supplier relation and that PARTNO+SUPPLIER is chosen as the primary key of the cost of goods relation. This design is now in fourth normal form. The unary relation for data item PARTNO will later be removed by the DMNS unless other user views produce some attributes for the relation or unless the

data modeler wishes to keep it by including an entry in the RELATION.DBF file for a relation with primary key PARTNO.

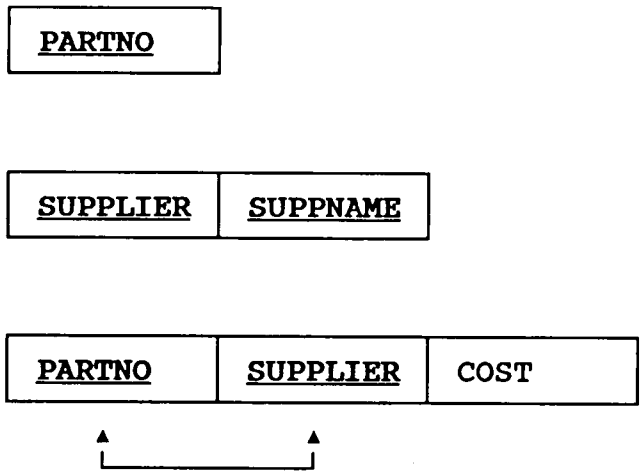


Figure 71. 4NF design for the relation of Figure 67.

The fifth and last case of data conversion in lower normal forms to fourth normal form is for a file in BCNF which does not comply with the fourth normal form. A relation meeting this condition is shown in Figure 72. In this relation, the semantics which exist between the data items are that a course may have different teachers and more than one text book. A teacher may teach different courses and a text may also be used for different courses. The dependencies for the relation are shown in Figure 73.

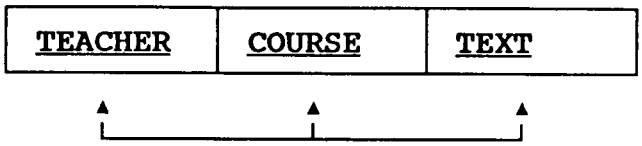


Figure 72. A relation which is not in 4NF.

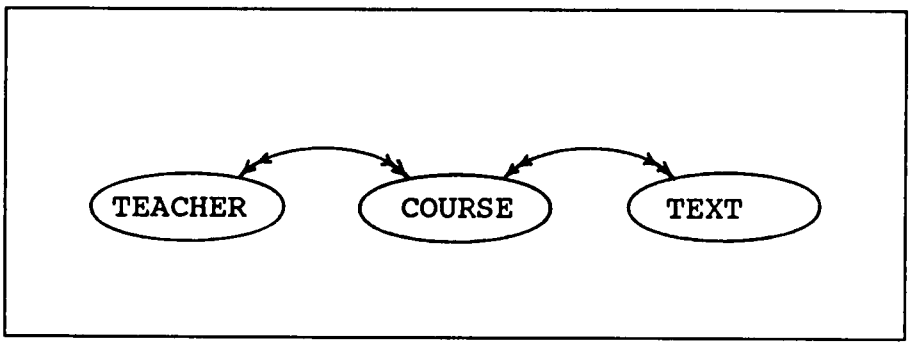


Figure 73. Dependencies for relation of Figure 72.

The relation in Figure 72 is an all-key relation; that is, every field is a component of the primary key. There are no functional dependencies in the relation, but there are multivalue dependencies. This file must be rearranged to eliminate possible update anomalies. The DMNS does this by introducing intersection relations in the merge phase for the relationships which exist between courses and teachers and between courses and textbooks. After the merge phase of the workbench has been completed, the dependencies appearing in Figure 74 will be in the workbench.

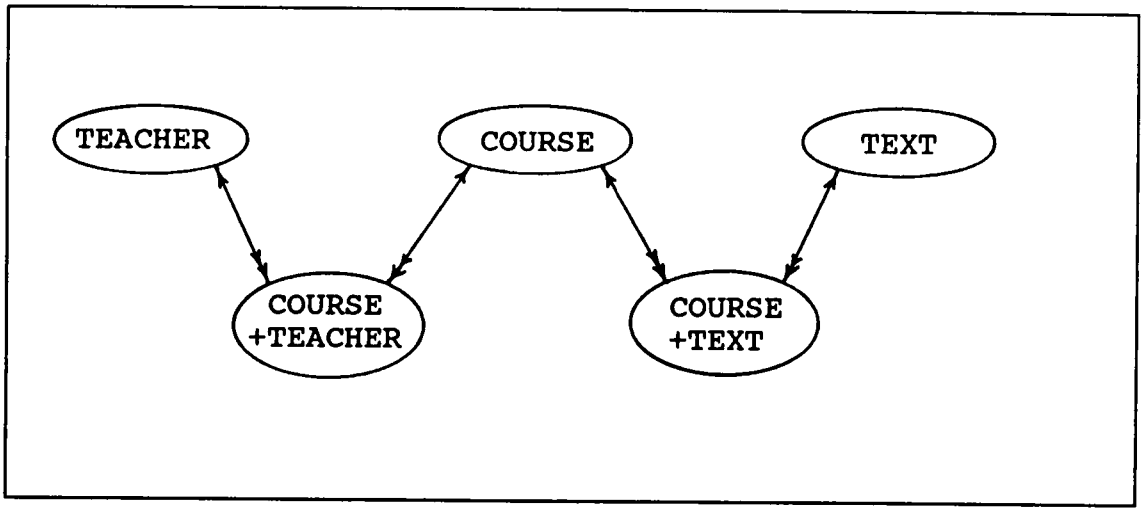


Figure 74. Dependencies after the merge phase.

Two new compound data item expressions (COURSE+TEACHER and COURSE+TEXT) have been introduced as a result of the bidirectional MVDs between the original data items. After the step just described has been completed, the only modification to the schema above will be that the unary relations for teachers, courses, and textbooks will be thrown away unless the data modeler wishes to keep them for data integrity or future data base growth purposes. If the data modeler decides to allow these individual data item relations to be eliminated, then remaining relational data base design will appear as in Figure 75.

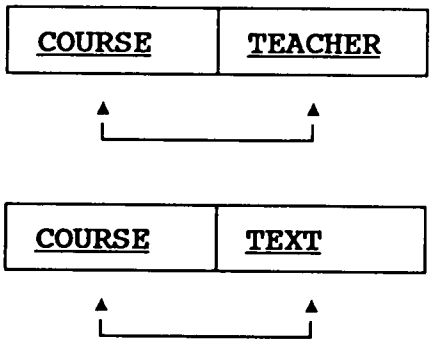


Figure 75. 4NF design for the relation of Figure 72.

Each of the cases previously examined were described as isolated situations. In fact, however, as user views are merged, the normalization problems just described may appear temporarily and then disappear as a new user view is added, or problems which might not exist in any individual user view may arise after the merger of various user views. The combinations of user views are infinite which makes tools like the DMNS valuable to data base designers.

VIII. DMNS SAMPLE DATA MODELING PROJECT

In order for the reader to gain a better understanding of how the Data Modeling and Normalization System works on a real problem, this section presents a sample data modeling problem with seven user views. The problem is taken from Martin's Managing the Data Base Environment.[16] Martin begins with seven user views which, during the manual normalization process, are revised as inconsistencies in the user views are found. Rather than duplicate this entire process and the quantity of output which the DMNS would produce for multiple runs, the example in this section will pick up with user views in Martin's final form.

Problem Definition

The data modeling problem to be shown is a transportation problem involving sea ports, ships, and shipments. The names of data items have been shortened in many cases to allow them to fit in the ten-character maximum length of data item names in the DMNS (the ten-character maximum was chosen to be consistent with the maximum length of dBASE III field names). The seven user views collected for the project are shown in Figures 76, 77, 78, 79, 80, 81, and 82.

---

[16]James Martin, Managing the Data-Base Environment, (Englewood Cliffs, N.J.: Prentiss-Hall, 1983), pp. 252-274.

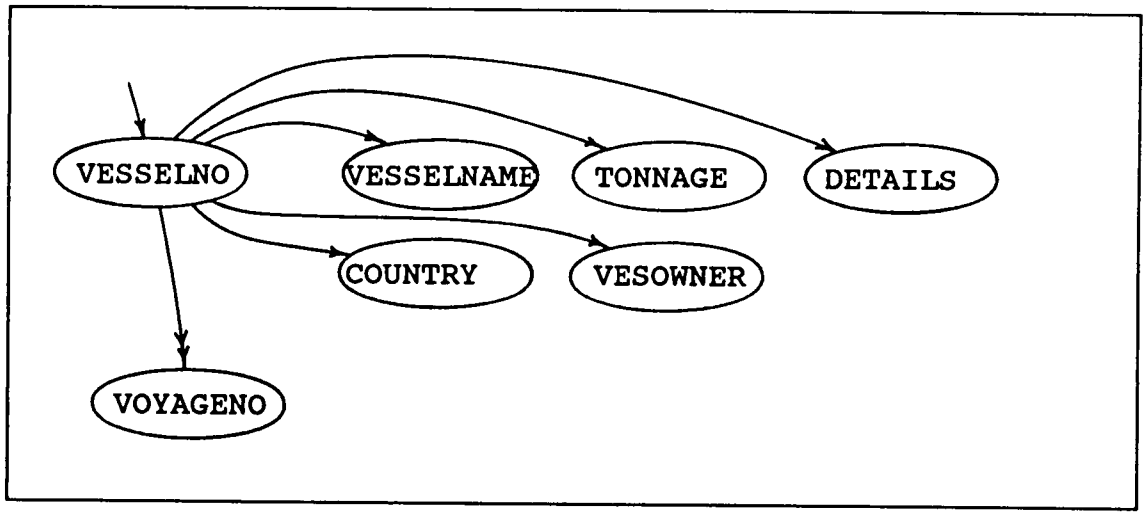


Figure 76. User view number one.

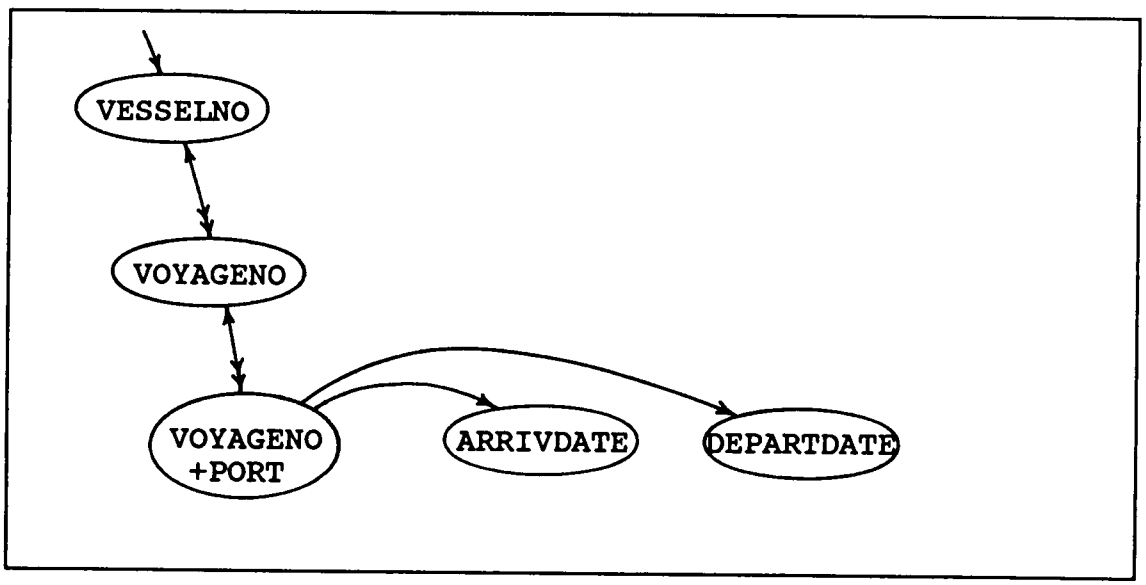


Figure 77. User view number two.

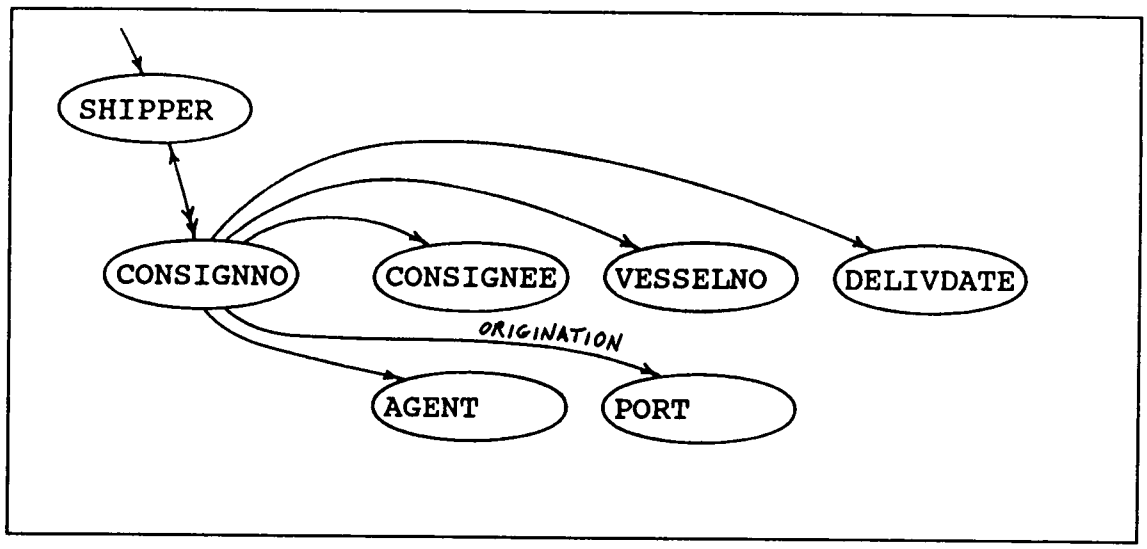


Figure 78. User view number three.

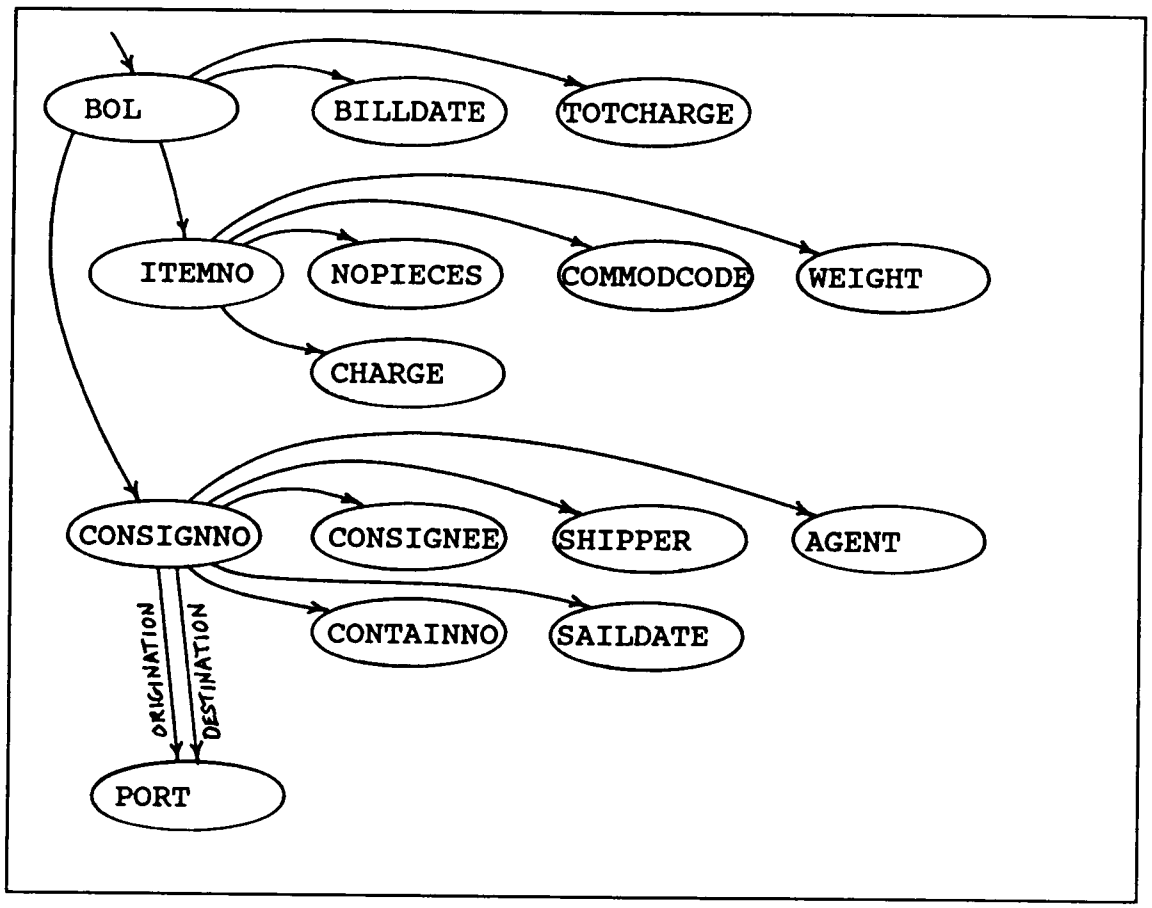


Figure 79. User view number four.

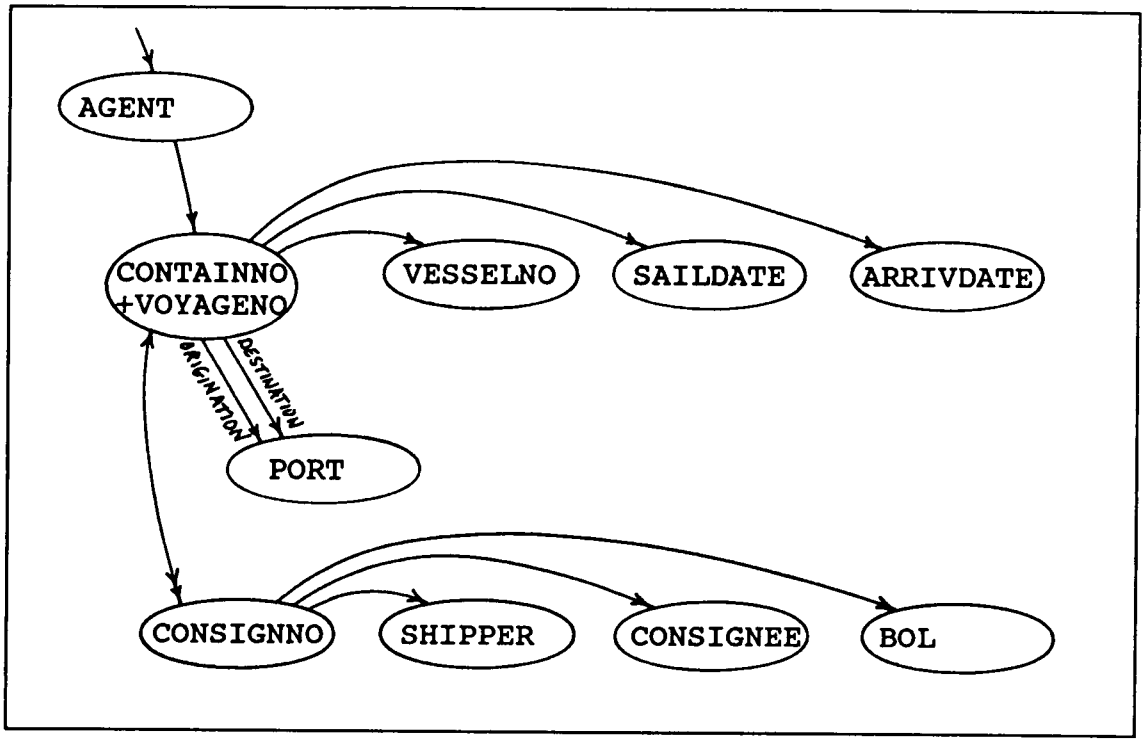


Figure 80. User view number five.

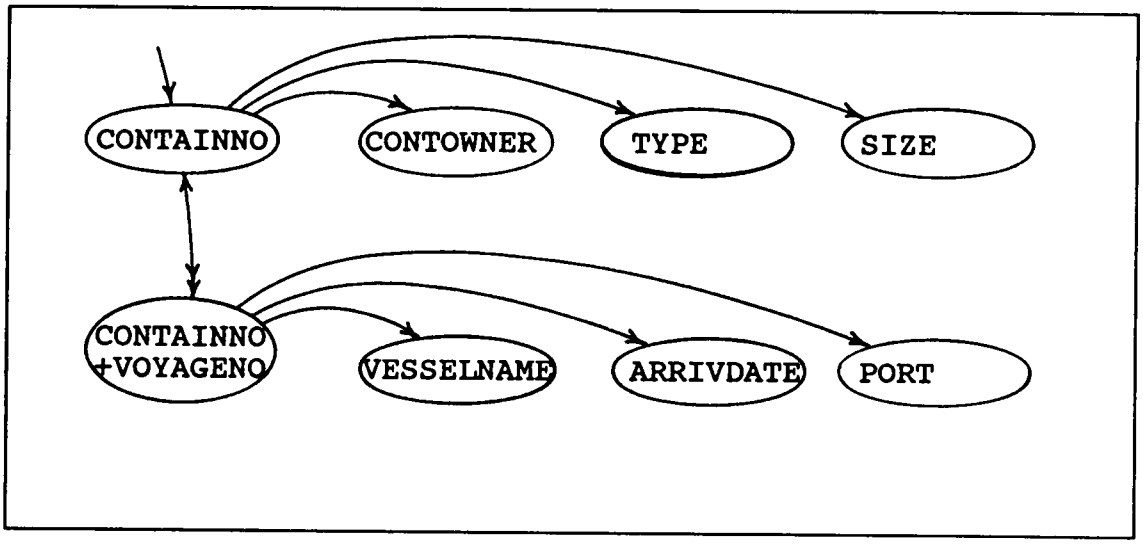


Figure 81. User view number six.

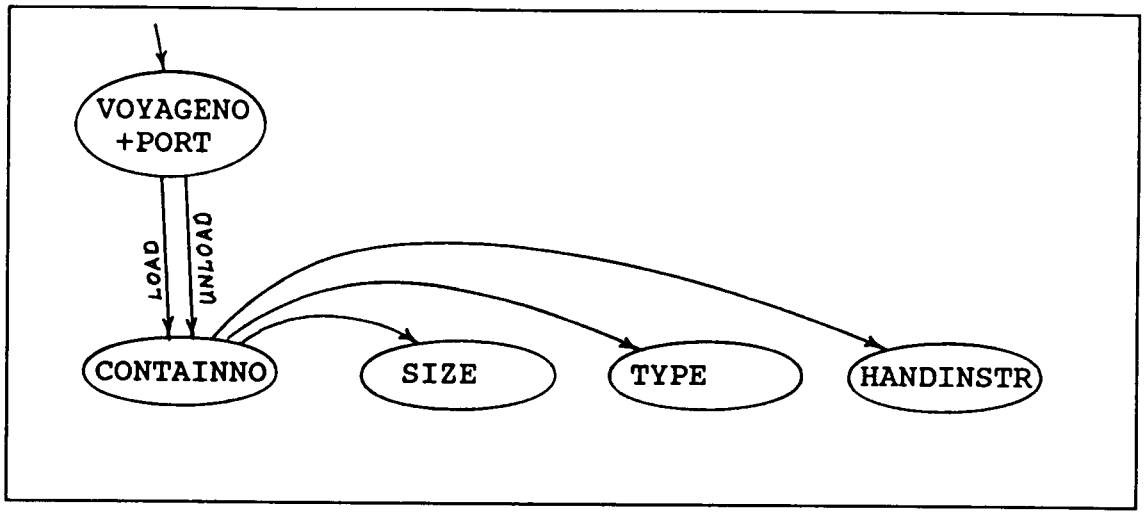


Figure 82. User view number seven.

In order to illustrate some of the output features of the DMNS, most of the data items in the illustrated user views have been added to the data dictionary, and most of the relations which will be formed by the DMNS have been named. Data items and relations may be added to their respective lists at any time. A good practice would be to add a data item as soon as it becomes known, even before user views incorporating the data item are entered. With regard to relations, the best practice would probably be to name relations after the workbench has been run on the schema in question. When the DMNS tells the modeler what the primary keys of the relations in the normalized data model will be, the data modeler can go back and name the relations before making another workbench run to show the newly assigned relation names.

Review of DMNS Output

The printout of the results of the DMNS workbench for the example problem is included as Appendix B beginning on p. 166. Each page of the printout will be discussed in the order in which it appears.

The first page of the output is printed during the selection phase of the workbench. It shows that seven separate user views were requested to be merged. The list of unique user views merged following the first list is identical in this case since only user views were requested. If a view set containing several user views had been requested, the selection routine would have filtered out any duplication in user views before it printed the final list.

Page 166 is for the merge phase of the workbench. Using the list of unique user views from the selection phase, the merge phase adds each dependency in the user views into the schema. Vertices are added to the directed graph for each new data item expression. Edges are added to one of the digraphs for each new dependency. In addition, the merge routine adds trivial dependencies and creates intersection vertices. For the first user view, the vertices and the edges for these dependencies are added without further additions from the DMNS. In the second user view, when the compound data item expression VOYAGENO+PORT is encountered in the dependency VOYAGENO⟶VOYAGENO+PORT, the DMNS adds the additional dependencies:

VOYAGENO+PORT—►►VOYAGENO,
PORT——►VOYAGENO+PORT, and
VOYAGENO+PORT—►►PORT.

The DMNS adds the label "DMNS added" to each of these new dependencies. When the dependency VOYAGENO+PORT——►VOYAGENO appears as the next dependency from user view two, the "DMNS added" label is removed and the dependency label from the user view (blanks) replaces it. The merge phase continues routinely until another compound data item expression, CONTAINNO+VOYAGENO, is encountered. Here again, the compound data item expression components are added to the schema by the DMNS.

The normalization phase follows the merge phase. A series of operations are performed in the normalization phase. Diagnostic messages are printed for each operation by the DMNS. The first message in the normalization section on p. 168 "No conflicting associations between data items found." indicates that no pair of vertices had both an FD and an MVD between them in the same direction.

Next, transitive dependencies are located and removed. The sample problem had six transitive dependencies as shown in the DMNS output. The first of these dependencies, CONTAINNO+VOYAGENO——►VESSELNO, is transitive because of the existence of the following two functional dependencies: CONTAINNO+VOYAGENO——►VOYAGENO and VOYAGENO——►VESSELNO. The other five dependencies can similarly be shown as transitive.

After all the transitive dependencies are listed, the next message concerns a pair of vertices which appear to be candidate keys. This pair is made up of the vertices BOL (bill of lading) and CONSIGNNO (consignment number). BOL is chosen as the primary key because it appears as a relation identifier in the relation list, and CONSIGNNO is assigned to be an alternate key. The functional dependency from CONSIGNNO to BOL is removed, and other dependencies from CONSIGNNO to other vertices are rerouted to BOL.

Next, a message informs the data modeler that the vertices VESSELNO and BOL are primary keys (because they each have an FD leading from them), and are known to be associated with each other through some dependency but have no dependency in the opposite direction between them. The DMNS must know the associations between keys in both directions. The DMNS resolves the problem of the unknown association by guessing that the unknown dependency is the type of dependency opposite that of the known dependency. Since the known dependency is an FD from BOL to VESSELNO, the dependency added by the DMNS is VESSELNO—►►BOL (Assumed MVD).

The label "Assumed MVD" is weaker than the label ("DMNS added") used in the merge phase. The "DMNS added" dependencies are true and absolutely correct dependencies in every situation. Assumed dependencies are guesses and

should be reviewed carefully by the data modeler to ensure their accuracy.

Three other unknown reverse key associations were found and listed. Functional dependencies were added in each of these cases since the known associations were multivalue dependencies.

After the unknown reverse association messages, the attribute PORT is denoted as having multiple dependencies (indicating separate roles) from the vertex CONTAINNO+VOYAGENO. Since PORT cannot appear twice as a foreign key in the same relation, renaming must be performed to implement both roles in a relational data base. The same can be said for the vertices VOYAGENO+PORT and CONTAINNO in the next message.

This message indicates that a cycle among the functional dependencies has been found. The dependencies which make up the cycle are shown following the introductory message. One can trace the route indicated by the dependencies as follows: VESSELNO, VOYAGENO, VOYAGENO+PORT, CONTAINNO, CONTAINNO+VOYAGENO, BOL, and back to VESSELNO.

The last message in the normalization phase indicates that all secondary key accesses are feasible. That is, all accesses from an attribute of one relation to some field of another relation can be implemented with the proper join operations.

Following the normalization phase messages, a final list of the dependencies remaining in the workbench is printed. From this list the relational data base design is formed.

The part of the printout shown on p. 171 is the overall design of the relational data base required to implement the application named the Overall Schema Report. Relations, their primary keys, their descriptions, and their associations with other relations are shown. The DMNS uses a heuristic to attempt to order the relations on the report in such a way that whenever two relations are associated with each other in a 1:M fashion, the FD points up and the MVD points down. If two relations do not have an association between them, then they are not linked with a line on the report.

Following the Overall Schema Report, the remainder of the DMNS printout (pp. 172-178) are seven separate reports called Relation in Context Reports. Each Relation in Context Report provides greater detail on an individual relation (called the primary relation) created by the DMNS. Fields contained in the primary relation are enclosed in a box of asterisks. Information about other relations which the primary relation is associated with are in boxes of dashed lines.

On Relation in Context Reports relations are named and described using data from the RELATION.DBF file. If there

is a relation whose primary key cannot be matched with an entry from the RELATION.DBF file it is named "Unknown" and the description is left blank. The relation at the top of p. 172 is an example of this. The primary key VESSELNO was not found in the RELATION.DBF file, so "Unknown" was used as the name for the relation of which VESSELNO is the primary key.

The hierarchical parent relations, if any, and child relations, if any, of the primary relation are enclosed in line boxes above and below the primary relation, respectively. Any associated relations shown with the primary relation have the name of the associated relation, the description, and the primary key expression within the line box.

Each primary relation has a header block which contains the name of the relation, the description, the primary key expression, and the details of the field(s) making up the primary key.

In addition to the header block, there may be one or more additional blocks containing other fields which are performing various roles within the relation. These role groupings are candidate keys, foreign keys, indexed attributes, and attributes.

Candidate key expressions for the primary relation, if any exist, will be listed in a block directly below the

primary key. Each component field of a candidate key is then listed separately below the candidate key expression.

Foreign keys, if any, are placed in the next block. These data item expressions are primary keys in some parental relation above the primary relation. Their presence in the primary relation is required in order to perform joins between the relations. The individual fields in each data item expression are listed below them.

Attributes make up the last two blocks which may exist in the primary relation. All attributes are solitary data items, so the field data for the attribute is listed without the data item expression.

The first attribute block is for indexed attributes (secondary keys). These attributes have a multivalue dependency from them to the primary key of their relation or to a field in another relation. If the data modeler decides that the access indicated by the MVD(s) from the attribute is important enough, then an index file associated with this attribute may be built and maintained in order to facilitate faster access.

The second attribute block is for regular attributes, or just "attributes." Again, these are listed in a column since they are all single data items.

The most complete example of a primary relation page as described above is p. 175 for Relation BOL. Above the primary relation are two other relations which have an MVD

from them to the BOL relation. These are the relation whose name is not known (with primary key VESSELNO) and the Relation SHIPMENT.

Within the primary relation box, the header block contains the name and description of the relation and the data item expression of the primary key (BOL). Since BOL is a single data item, it alone is detailed below the data item expression. The description for the BOL field is "Bill of Lading," the type of field is "N" for numeric, and the width is six places with no decimal places.

The next block in the primary relation box is for the relation's alternate key, CONSIGNNO. As a candidate key, CONSIGNNO must follow all the referential and integrity rules that the primary key follows.

The middle block is for the two foreign keys required to join with the two parental relations mentioned above. These are VESSELNO and CONTAINNO+VOYAGENO. Details for these fields follow each expression.

The fourth block is for two indexed attributes present in this relation. These are SHIPPER and AGENT. A review of the final list of workbench dependencies on a prior page of the output will reveal which MVDs cause these attributes to appear on the indexed attribute list. The data modeler should review these accesses to ensure that an index needs to be built and maintained for them instead of just building

the indexes on demand or processing the accesses sequentially.

The last block in the primary relation is for the attributes of BOL. These are listed in a column. One attribute in the list, CONSIGNEE, was not found in the data dictionary and is labeled "Field not found." Below the primary relation are its child relations. In this case the ITEM relation is multivalue dependent on BOL.

Using the information provided by the DMNS will allow a data modeler to verify their inputs and eventually complete an accurate and stable relational data base design.

Comparison with Martin's Results

The overall relational model of the results by Martin[17] are shown in Figure 83. Martin's design included six relations. The results of the DMNS' design are shown in Figure 84. The DMNS suggests seven relations be created for the example application. Since Martin does not assign names to the relations in his design, relations will be referred to by their unique primary keys. There are four relations in both designs which are identical. These are the four pairs of relations from both designs with the following primary keys--VESSELNO, BOL, CONTAINNO+VOYAGENO, and ITEMNO.

---

[17]James Martin, Managing the Data-Base Environment, (Englewood Cliffs, N.J.: Prentiss-Hall, 1983), pp. 272.

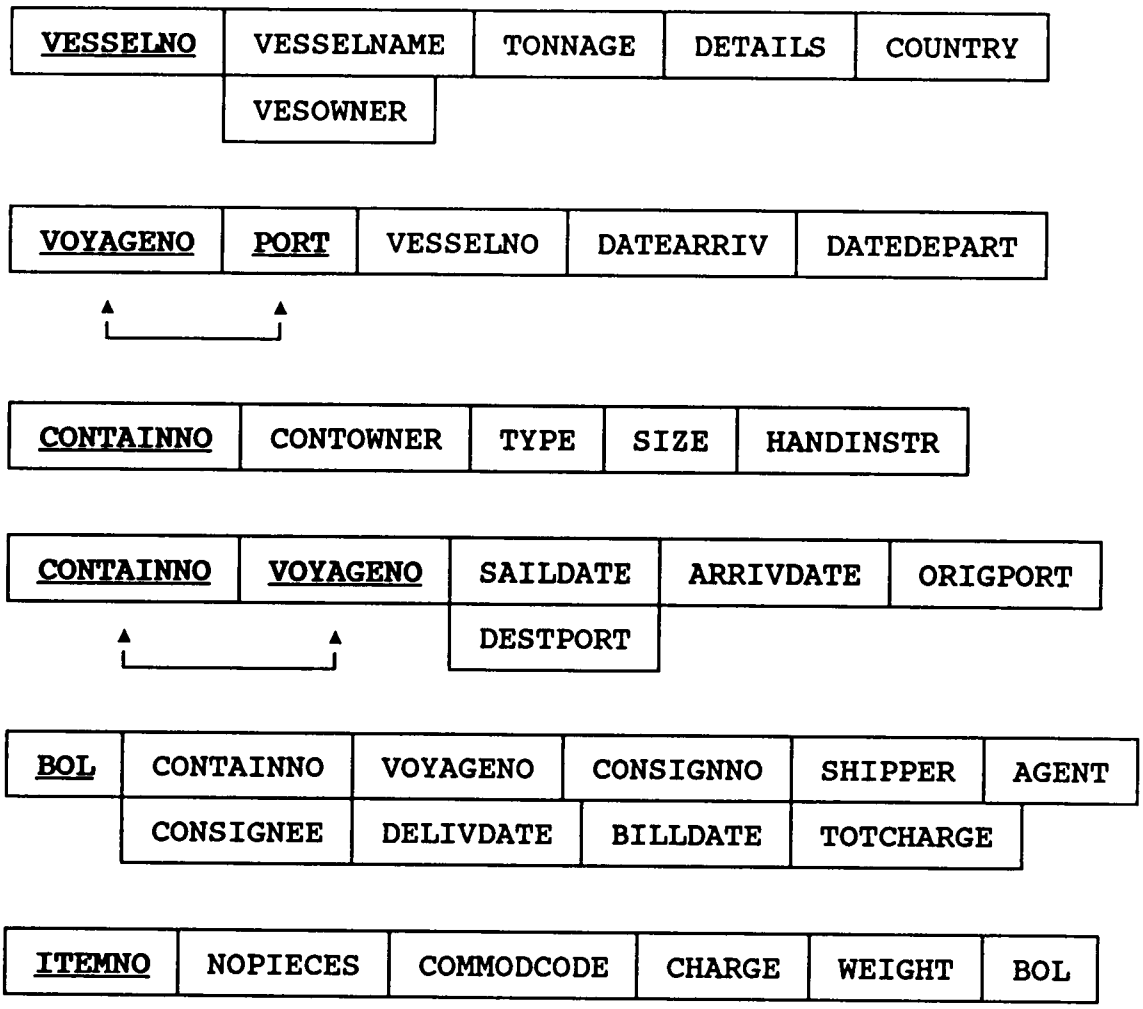


Figure 83. Martin's relational design.

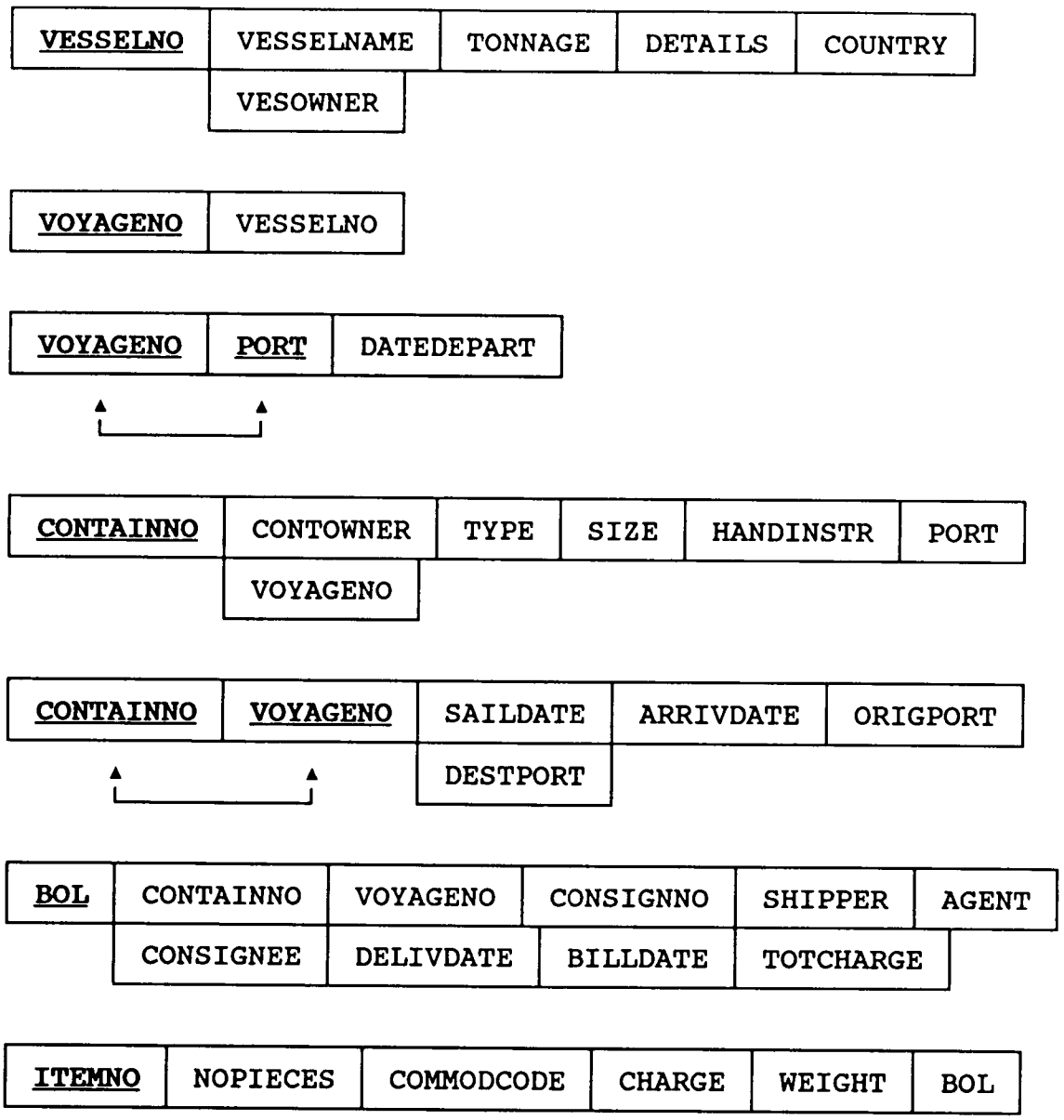


Figure 84. The DMNS' relational design.

The differences between the two designs will now be compared in more detail. The additional relation in the DMNS' design has VOYAGENO as its primary key with VESSELNO as a foreign key. This relation provides the domain check for the relation with the primary key CONTAINNO+VOYAGENO and provides a cross-reference for VOYAGENOs to VESSELNOs. Martin instead places VESSELNO as a foreign key in the relation where VOYAGENO+PORT is the primary key and omits any relation with VOYAGENO as primary key. Martin's design requires that whenever the vessel assigned to a voyage is changed, that multiple tuples associated with the various ports on the voyage be updated in the VOYAGENO+PORT relation.

The difference in the relations with the data item expression VOYAGENO+PORT as their primary key is that VESSELNO is included as a foreign key in Martin's design as described above.

The pair of relations with CONTAINNO as their primary keys differ in that in the DMNS' design VOYAGENO and PORT are foreign keys (VOYAGENO+PORT is the primary key of another relation). This relationship arises in the DMNS' design due to the assumption of a functional dependency as an unknown reverse association between CONTAINNO and VOYAGENO+PORT. That is, CONTAINNO——▸VOYAGENO+PORT was assumed by the DMNS because there is an MVD from VOYAGENO+PORT to CONTAINNO (actually there are two MVDs from

VOYAGENO+PORT to CONTAINNO). Because two different relationships need to be represented (loading a container and unloading a container), the modeler should introduce subentities for the VOYAGENO+PORT data item expression with different names. By introducing subentities, both relationships can be included in the model. Afterwards, the CONTAINNO relation would appear as follows:

CONTAINNO	CONTOWNER	TYPE	SIZE	HANDINSTR
	LVOYAGENO	LPORT	UVOYAGENO	UPORT

The "L" and "U" prefixes would represent load and unload, respectively. Martin shows the load and unload MVDs from VOYAGENO+PORT to CONTAINNO in his final dependency drawing but makes no provision in his design for implementation of these dependencies.

One final note on the two designs is that both designs indicate that the fields SHIPPER and AGENT in the BOL relation are secondary keys. As such, index files would probably be built on these fields to improve access speed for finding particular values of these fields.

In summary, the two designs are quite similar, but the better of the two designs is the one created by the DMNS. Having the relation with VOYAGENO as the primary key allows a domain check for proper VOYAGENOs in the relation with VOYAGENO+PORT as the primary key. The inclusion of the CONTAINNO and VOYAGENO fields in the relation with BOL as

the primary key allows an important reverse association to be mapped between the relations with BOL and CONTAINNO+VOYAGENO as the primary keys. As discussed previously, this needs further modification since two distinct relationships need to be mapped between them instead of only one, but at least the DMNS' design attempts to account for the relationships indicated in the current data model.

IX. CONCLUSIONS

The absence of useful tools for comprehensive data base design is an important problem in the microcomputer arena. Indeed, the lack of general information on how to build a data base design tool represents a void for all hardware levels of computing. The Data Modeling and Normalization System developed as part of this thesis should help fill these two gaps.

Microcomputer data base designers who may have succeeded in performing adequate designs for small data bases using their intuition or manual techniques in the past are now in need of help. Current microprocessors and hard disk subsystems are fast enough to handle complex data bases with thousands of records. Unless these data bases are designed properly the first time, developers will be spending more of their time reprogramming their old data bases to meet new needs than they spend developing new applications. In a society and business climate hungering for more and more information, inefficiency in data base development will not be tolerated.

The performance of the DMNS itself is also an issue. The example problem which is described in Section VIII requires only 11 seconds to complete the normalization on an Intel 80386-based personal computer. To forecast the running time to even larger problems, analysis of the workbench software showed that the worst-case algorithmic

complexity of the software is $O(f^5)$, where f is the number of functional dependencies. The average case algorithmic complexity was not determined but would be considerably better than the worstcase.

The DMNS performs functions which will make good data base design possible. However, no tool is useful if it is not easy to use. If data modelers do not feel that the menus and data entry into the DMNS are intuitive, then it will not be used. This is why much thought went into the design of the screens, menus, and data entry methods used in the DMNS.

Before proclaiming the DMNS as the answer to all data base design problems, it should be reiterated that data modeling is sometimes as much an art as it is a science. Without the data modeler's savvy and expertise, the first law of programming--"Garbage In, Garbage Out"--applies to the DMNS.

The lack of documentation on how normalization software works can be traced to the software implementation roots. Existing normalization software has been added onto data dictionary software written first for that purpose. These packages are commercial and proprietary, and the companies selling them do not wish to give away any trade secrets. Furthermore, because they are built upon other products and old data structures, if one had the documentation to the systems, it might not be worth building onto.

The DMNS does not represent all that can be done with the user views for a data base application. The DMNS could readily be extended to not only provide the data modeler with pertinent information on how to design a relational data base for an application but to also create the files with the appropriate structure itself. These files would then be populated with the actual application data. Even beyond this extension is the possibility of transforming the DMNS into a new type of application generator. Recall that user views, as described herein, are transaction based. User views not only tell the application builder about the semantic relationships between the application's data items, they also tell the designer when, where, and how data are entered and retrieved. To create advanced applications, additional data for each user view would have to be collected, such as, who has access to what data, how any data retrieved is to be ordered, and what integrity verifications must be performed. These types of enhancements would however fit the data modeling concepts which are presented within the thesis without major restructuring.

The DMNS' use of directed graphs to solve the normalization problem is simple, yet elegant. There is not likely any data structure more natural for representing a schema than directed graphs. An added benefit of directed graphs is that there is a rich library of software readily

available to adapt to solving various graph problems. Use of the breadth-first search to find transitive dependencies and use of the depth-first search to find cycles are examples of well-known algorithms that were adapted to solve normalization problems rather than starting completely from the beginning on the problems if one used a new or lesser known data structure.

Someday there will be a tool available, based on artificial intelligence, that uses data models to not only design data bases but to write entire applications from them without programming. But until that time comes, tools like the Data Modeling and Normalization System will prove very helpful to the data base designers and programmers who must design and program these systems today.

BIBLIOGRAPHY

BIBLIOGRAPHY

Amsbury, Wayne. Data Structures from Arrays to Priority Queues. Belmont, Calif.: Wadsworth Publishing Co., 1985.

Ashton-Tate. Learning and Using dBase III Plus. Unpublished work.

Ashton-Tate. Programming with dBase III Plus. Unpublished work.

Bertztiss, A.T. Data Structures: Theory and Practice. New York: Academic Press, 1975.

Chen, P.P. "The entity-relationship model: toward a unified view of data." ACM Transactions on Database Systems, 1, No. 1, pp. 9-36 (1976).

Codd, Edgar F. "Further normalization of the data base relational model." Data Base Systems, 6, pp. 33-64 (1972).

Date, C.J. An Introduction to Database Systems. Reading, Mass.: Addison-Wesley, 1981.

Deo, Narsingh, Jurg Niebergelt, and Edward M. Reingold. Combinatorial Algorithms: Theory and Practice. Englewood Cliffs, N.J.: Prentiss-Hall, 1977.

Fagin, R. "Multivalued dependencies and a new normal form for relational databases." ACM Transactions on Database Systems, 2, No.3, pp. 262-278 (1977).

Fox & Geller, Inc. Quickcode III. Elmwood Park, N.J.: Fox & Geller, Inc., 1984.

Holsapple, Clyde. A Primer on Data Base Management. Lafayette, Ind.: Micro Data Base Systems, 1982.

Korth, Henry F., and Abraham Silberschatz. Database System Concepts. New York: McGraw Hill, 1986.

Lattice Incorporated. Lattice dBC III Library for MS-DOS. Glen Ellyn, Ill.: Lattice, 1986.

Management Systems and Programming Limited. Designmanager. Lexington, Mass.: Management Systems and Programming, 1983.

Martin, James. An End-User's Guide to Data Base. Englewood Cliffs, N.J.: Prentiss-Hall, 1981.

Martin, James. Computer Data-Base Organization. Englewood Cliffs, N.J.: Prentiss-Hall, 1977.

Martin, James. Managing the Data-Base Environment. Englewood Cliffs, N.J.: Prentiss-Hall, 1983.

Microsoft Corporation. Microsoft C Compiler Library Reference Manual. Bellvue, Wash.: Microsoft, 1985.

Rhyne, B. Timothy. User's Guide to the Data Modeling and Normalization System. Unpublished manuscript.

The Research Group. Saywhat?!. San Mateo, Calif.: The Research Group, 1987.

Ullman, Jeffrey D. Principles of Data Base Systems. Rockville, Md.: Computer Science Press, 1982.

APPENDIXES

APPENDIX A

SAMPLE LISTING OF PROGRAMS

```
/*************************************************************************
*              Data Modeling and Normalization System                   *
*-----------------------------------------------------------------------*
*                              STRUCT.H                                 *
*                                                                       *
*  Defines structures for all program modules.                          *
*-----------------------------------------------------------------------*
*  Author: B. T. Rhyne                                                  *
*  Copyright (C) 1988, All Rights Reserved.                             *
*************************************************************************/

#define TAGLEN 43

/*************************************************************************
* Structures VERTEX and EDGE are for directed graph data representation.*
*************************************************************************/
typedef struct
{
        char    fieldexpr[TAGLEN+1];
        int     nofields;
        int     fdsin;
        int     fdsout;
        int     visited;
        char    key;
        int     fdndx;
        int     mvdndx;
}       VERTEX;

typedef struct
{
        int     lhs;
        int     rhs;
        char    name[12];
        int     depno;
        int     next;
}       EDGE;

/*************************************************************************
* PATHSTRUCT is a route saved during breadth first traversal of digraph.*
*************************************************************************/
typedef struct  PATHSTRUCT
{
        int     nohops;
        int     hop[20];
}       PATH;

/*************************************************************************
* Structure RELATION is used to store the list of relations in the      *
* workbench.                                                            *
*************************************************************************/
typedef struct
{
        char    relname[9];
        char    reldesc[26];
        char    relidflds[TAGLEN+1];
        int     vertexno;
        int     priority;
        int     alphaorder;
        int     priorityorder;
}       RELATION;
```

```
/*************************************************************************
*              Data Modeling and Normalization System                    *
*------------------------------------------------------------------------*
*                              STRUCT.H                                  *
*                                                                        *
*  Defines structures for all program modules.                           *
*------------------------------------------------------------------------*
*  Author: B. T. Rhyne                                                   *
*  Copyright (C) 1988, All Rights Reserved.                              *
*************************************************************************/

/*************************************************************************
* Function declarations for type checking.                               *
*************************************************************************/
/*global*/  void bombout(void);
/*global*/  void dberror(int ,char *);
/*global*/  int printwa(void);
/*global*/  void savewb(void);
/*global*/  int main(int ,char * * *);
/*global*/  void selectionphase(void);
/*global*/  void mergephase(void);
/*global*/  int bfs(char *);
/*global*/  int dfs(int ,struct PATHSTRUCT );
/*global*/  int dfs2(int ,struct PATHSTRUCT ,int );
/*global*/  void modcandidatekey(int ,int );
/*global*/  int printdigraph(void);
/*global*/  void normalizationphase(void);
/*global*/  void pageheading(char *);
/*global*/  void printrelations(void);
/*global*/  int relmvdsout(int );
/*global*/  int relmvdsin(int );
/*global*/  void fillrelationtable(void);
/*global*/  int po(int );
/*global*/  void setchin(int );
/*global*/  void setchmid(void);
/*global*/  void setchout(int );
/*global*/  void outputphase(void);
/*global*/  int index(char *);
/*global*/  char *tag(int );
/*global*/  int depnoused(int );
/*global*/  int addvertex(char *);
/*global*/  int explodekey(char *,char (*)[TAGLEN+1]);
/*global*/  int mergekeys(char *,char *,char *);
/*global*/  int anyfd(char *,char *);
/*global*/  int namedfd(char *,char *,char *);
/*global*/  int addfd(char *,char *,char *,int );
/*global*/  int delallfd(char *,char *);
/*global*/  int delfd(char *,char *,char *);
/*global*/  int anymvd(char *,char *);
/*global*/  int namedmvd(char *,char *,char *);
/*global*/  int addmvd(char *,char *,char *,int );
/*global*/  int delallmvd(char *,char *);
/*global*/  int delmvd(char *,char *,char *);
/*global*/  void enqueue(char *,int );
/*global*/  void dequeue(char *,int *);
/*global*/  int queuesize(void);
/*global*/  void showqueue(void);
/*global*/  void dblocfld(int ,char *,int *,int *,int *);
/*global*/  int dbgint(int ,char *,int );
/*global*/  float dbgfloat(int ,char *,int );
/*global*/  char dbgchar(int ,char *,int );
/*global*/  void dbgstr(int ,char *,char *,int );
/*global*/  void dbpint(int ,int ,char *,int );
/*global*/  void dbpfloat(float ,int ,char *,int );
/*global*/  void dbpchar(char ,int ,char *,int );
/*global*/  void dbpstr(char *,int ,char *,int );
/*global*/  int waopen(int ,char *,char *);
/*global*/  int waclose(int );
```

```
/*************************************************************************
*                  Data Modeling and Normalization System               *
*-----------------------------------------------------------------------*
*                                                                       *
*                               DIGRAPH.C                               *
*                                                                       *
*   These routines are the directed graph primitive functions described *
*   in Section VII.                                                     *
*-----------------------------------------------------------------------*
*   Author: B. T. Rhyne                                                 *
*   Copyright (C), 1988.  All Rights Reserved.                          *
*                                                                       *
*************************************************************************/


int     addfd(lhsid,rhsid,depname,depno)
/*************************************************************************
* addfd adds a functional dependency to the directed graph.             *
*************************************************************************/
char *lhsid, *rhsid, *depname;
int  depno;
{
     int     i, first;

     if (strcmp(depname,dmnsname) == 0) {
          if (anyfd(lhsid,rhsid) != NOMATCH) return FALSE;
     } else {
          /* If the only fd is a DMNS added fd, replace it. */
          first = namedfd(lhsid,rhsid,dmnsname);
          if (first == NOMATCH) {
               if (namedfd(lhsid,rhsid,depname) != NOMATCH) return FALSE;
          } else {
               fprintf(msg,"Replacing %25s ---> %-25s(%s)\n",lhsid,rhsid,depname);
               strcpy(fd[first].name, depname);
               fd[first].depno = depno;
               return TRUE;
          }
     }
     fprintf(msg,"Adding    %25s ---> %-25s (%s)\n",lhsid,rhsid,depname);

     fd[nofds].lhs = index(lhsid);
     fd[nofds].rhs = index(rhsid);
     strcpy(fd[nofds].name,depname);
     fd[nofds].depno = depno;
     fd[nofds].next = NONE;
     /*****************************************************************
     * Add to link list of for vertex(lhs)                            *
     *****************************************************************/
     if (vertex[fd[nofds].lhs].fdndx == NONE) {
          vertex[fd[nofds].lhs].fdndx = nofds;
     } else {
          for (i=vertex[fd[nofds].lhs].fdndx; fd[i].next!=NONE; i=fd[i].next) ;
          fd[i].next = nofds;
     }
     vertex[fd[nofds].lhs].fdsout++;
     vertex[fd[nofds].rhs].fdsin++;
     nofds++;
     return TRUE;
}
```

```
/*************************************************************************
*                  Data Modeling and Normalization System                *
*------------------------------------------------------------------------*
*                                                                        *
*                              NORMALIZ.C                                *
*                                                                        *
*   This routine modifies the schema already stored in two directed      *
*   graphs in order to create a fourth normal form relational data       *
*   base design from it.                                                 *
*------------------------------------------------------------------------*
*   Author: B. T. Rhyne                                                  *
*   Copyright (C), 1988.  All Rights Reserved.                           *
*                                                                        *
*************************************************************************/


int     bfs(sourcevertex)
/*************************************************************************
* Breadth first search for transitive dependencies.                      *
*************************************************************************/
char    *sourcevertex;
{
     int     dist, currentndx, i;
     char    current[TAGLEN+1];

     for (i=0; i<novertices; i++) vertex[i].visited = -1;
     dist = 0;
     enqueue(sourcevertex,0);
     /*****************************************************************
     * BFS works from the queue of unvisted vertices.                 *
     *****************************************************************/
     while (queuesize() > 0) {
          dequeue(current, &dist);
          currentndx = index(current);
          if (vertex[currentndx].visited > -1) {
               /*********************************************
               * This vertex has been visited before.       *
               *********************************************/
               if (dist > 1 && vertex[currentndx].visited == 1) {
                    /*******************************************************
                    * This vertex has already has an fd directly to the    *
                    * vertex of interest.  This is a transitive dependency.*
                    *******************************************************/
                    if (subset(sourcevertex,current)==FALSE) {
                         fprintf(msg,"\nThe following FD is transitive:\n");
                         delallfd(current,sourcevertex);
                         return -1;
                    }
               }
          } else {
               /*********************************************
               * Enqueue each parent of current node.       *
               *********************************************/
               for (i=0; i<nofds; i++) {
                    if (fd[i].rhs == currentndx) {
                         enqueue(tag(fd[i].lhs), dist+1);
                    }
               }
          }
          vertex[currentndx].visited = dist;
     }
     return 0;
}
```

APPENDIX B

RESULTS OF EXAMPLE NORMALIZATION PROBLEM

```
          --- DMNS Work Bench Begins ---

*** Selection Phase ***

Merging the following view sets / user views:
User View MARTIN1
User View MARTIN2
User View MARTIN3
User View MARTIN4
User View MARTIN5
User View MARTIN6
User View MARTIN7

The user views listed below will be included:
MARTIN1     MARTIN2     MARTIN3     MARTIN4     MARTIN5
MARTIN6     MARTIN7
```

```
*** Merge Phase ***


Merging User View: MARTIN1
Adding                    VESSELNO ---> VESSELNAME          (           )
Adding                    VESSELNO ---> TONNAGE             (           )
Adding                    VESSELNO ---> DETAILS             (           )
Adding                    VESSELNO ---> COUNTRY             (           )
Adding                    VESSELNO ---> VESOWNER            (           )
Adding                    VESSELNO -->> VOYAGENO            (           )

Merging User View: MARTIN2
Adding                    VOYAGENO ---> VESSELNO            (           )
Adding                    VOYAGENO -->> VOYAGENO+PORT       (           )
Adding              VOYAGENO+PORT ---> VOYAGENO             (DMNS added )
Adding              VOYAGENO+PORT ---> PORT                 (DMNS added )
Adding                       PORT -->> VOYAGENO+PORT        (DMNS added )
Replacing           VOYAGENO+PORT ---> VOYAGENO             (           )
Adding              VOYAGENO+PORT ---> DEPARTDATE           (           )

Merging User View: MARTIN3
Adding                    SHIPPER -->> CONSIGNNO            (           )
Adding                  CONSIGNNO ---> SHIPPER              (           )
Adding                  CONSIGNNO ---> CONSIGNEE            (           )
Adding                  CONSIGNNO ---> VESSELNO             (           )
Adding                  CONSIGNNO ---> DELIVDATE            (           )
Adding                  CONSIGNNO ---> PORT                 (ORIGINATION)
Adding                  CONSIGNNO ---> AGENT                (           )

Merging User View: MARTIN4
Adding                        BOL ---> BILLDATE             (           )
Adding                        BOL ---> TOTCHARGE            (           )
Adding                        BOL ---> CONSIGNNO            (           )
Adding                        BOL -->> ITEMNO               (           )
Adding                  CONSIGNNO ---> CONTAINNO            (           )
Adding                  CONSIGNNO ---> PORT                 (DESTINATION)
Adding                  CONSIGNNO ---> SAILDATE             (           )
Adding                     ITEMNO ---> NOPIECES             (           )
Adding                     ITEMNO ---> COMMODCODE           (           )
Adding                     ITEMNO ---> WEIGHT               (           )
Adding                     ITEMNO ---> CHARGE               (           )

Merging User View: MARTIN5
Adding                      AGENT -->> CONTAINNO+VOYAGENO   (           )
Adding          CONTAINNO+VOYAGENO ---> CONTAINNO           (DMNS added )
Adding                   CONTAINNO -->> CONTAINNO+VOYAGENO  (DMNS added )
Adding          CONTAINNO+VOYAGENO ---> VOYAGENO            (DMNS added )
Adding                   VOYAGENO -->> CONTAINNO+VOYAGENO   (DMNS added )
Adding          CONTAINNO+VOYAGENO ---> VESSELNO            (           )
Adding          CONTAINNO+VOYAGENO ---> PORT                (ORIGINATION)
Adding          CONTAINNO+VOYAGENO ---> SAILDATE            (           )
Adding          CONTAINNO+VOYAGENO ---> PORT                (DESTINATION)
Adding          CONTAINNO+VOYAGENO ---> ARRIVDATE           (           )
Adding          CONTAINNO+VOYAGENO -->> CONSIGNNO           (           )
Adding                   CONSIGNNO ---> BOL                 (           )

Merging User View: MARTIN6
Adding                   CONTAINNO ---> CONTOWNER           (           )
Adding                   CONTAINNO ---> TYPE                (           )
Adding                   CONTAINNO ---> SIZE                (           )
Replacing                CONTAINNO -->> CONTAINNO+VOYAGENO  (           )
Adding          CONTAINNO+VOYAGENO ---> VESSELNAME          (           )

Merging User View: MARTIN7
Adding              VOYAGENO+PORT -->> CONTAINNO            (LOAD       )
Adding              VOYAGENO+PORT -->> CONTAINNO            (UNLOAD     )
Adding                   CONTAINNO ---> HANDINSTR           (           )
```

```
*** Normalization Phase ***


No conflicting associations between data items found.

The following FD is transitive:
Deleting          CONTAINNO+VOYAGENO ---> VESSELNO                 (           )

The following FD is transitive:
Deleting          CONTAINNO+VOYAGENO ---> VESSELNAME               (           )

The following FD is transitive:
Deleting                    CONSIGNNO ---> PORT                    (ORIGINATION)
Deleting                    CONSIGNNO ---> PORT                    (DESTINATION)

The following FD is transitive:
Deleting                    CONSIGNNO ---> CONTAINNO               (           )

The following FD is transitive:
Deleting                    CONSIGNNO ---> SAILDATE                (           )

Dependency from candidate key to primary key deleted:
Deleting                    CONSIGNNO ---> BOL                     (           )
Dependencies between attributes and candidate key:
CONSIGNNO
Are being redirected to primary key:
BOL
Deleting                    CONSIGNNO ---> SHIPPER                 (           )
Adding                            BOL ---> SHIPPER                 (           )
Deleting                    CONSIGNNO ---> CONSIGNEE               (           )
Adding                            BOL ---> CONSIGNEE               (           )
Deleting                    CONSIGNNO ---> VESSELNO                (           )
Adding                            BOL ---> VESSELNO                (           )
Deleting                    CONSIGNNO ---> DELIVDATE               (           )
Adding                            BOL ---> DELIVDATE               (           )
Deleting                    CONSIGNNO ---> AGENT                   (           )
Adding                            BOL ---> AGENT                   (           )
Deleting                      SHIPPER -->> CONSIGNNO               (           )
Adding                        SHIPPER -->> BOL                     (           )
Deleting          CONTAINNO+VOYAGENO -->> CONSIGNNO                (           )
Adding            CONTAINNO+VOYAGENO -->> BOL                      (           )

Reverse association
from key: VESSELNO
to key:   BOL
Multivalued dependency assumed.
Adding                      VESSELNO -->> BOL                      (Assumed MVD)

Reverse association
from key: CONTAINNO
to key:   VOYAGENO+PORT
Functional dependency assumed.
Adding                    CONTAINNO ---> VOYAGENO+PORT             (Assumed FD )

Reverse association
from key: ITEMNO
to key:   BOL
Functional dependency assumed.
Adding                      ITEMNO ---> BOL                        (Assumed FD )

Reverse association
from key: BOL
to key:   CONTAINNO+VOYAGENO
Functional dependency assumed.
Adding                        BOL ---> CONTAINNO+VOYAGENO          (Assumed FD )
```

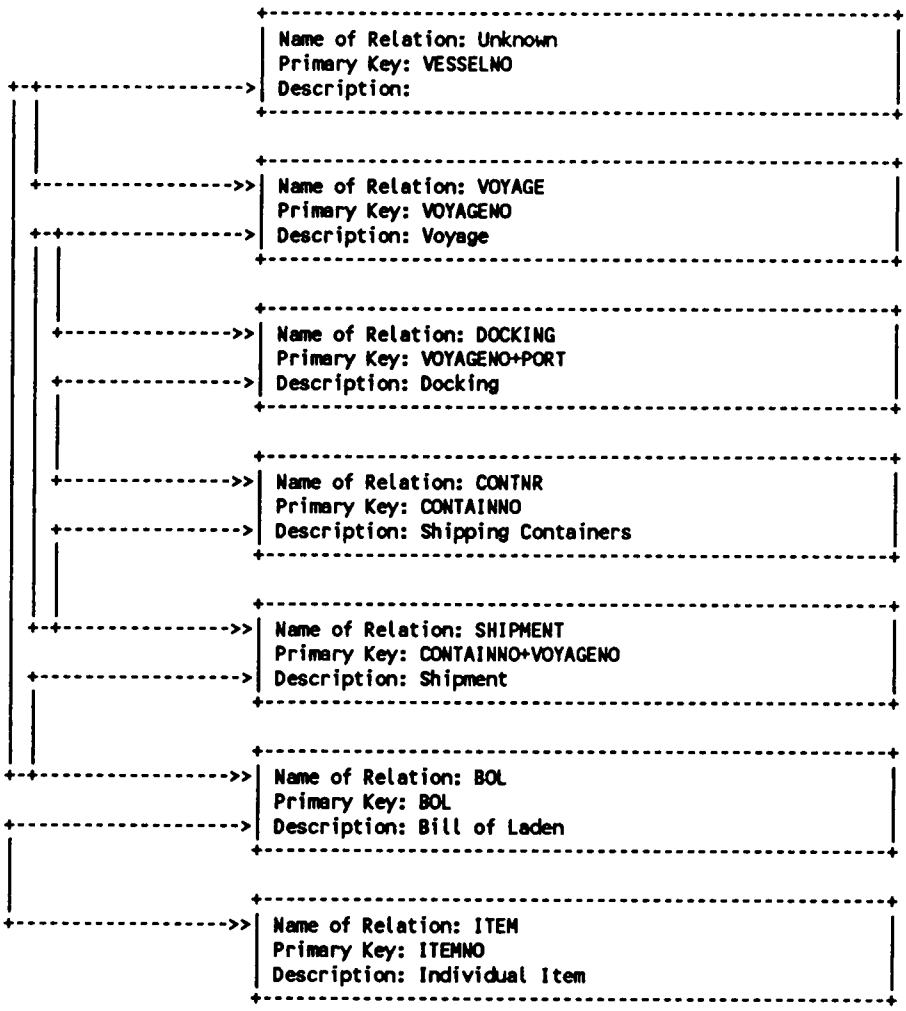
```
Single data item:
PORT
Does not stand as a key.
Deleting              VOYAGENO+PORT ---> PORT                    (DMNS added )
Deleting                      PORT -->> VOYAGENO+PORT           (DMNS added )

The attribute below:
PORT
Is listed as functionally dependent on the primary keys below:
CONTAINNO+VOYAGENO
CONTAINNO+VOYAGENO
It will be duplicated in the schema for both keys.

The R.H.S.  of the pair of functional dependencies below must be renamed
if two separate relationships exist between these two data items.
           CONTAINNO+VOYAGENO ---> PORT                          (DESTINATION)
           CONTAINNO+VOYAGENO ---> PORT                          (ORIGINATION)

The R.H.S.  of the pair of multivalued dependencies below must be renamed
if two separate relationships exist between these two data items.
                VOYAGENO+PORT -->> CONTAINNO                     (UNLOAD     )
                VOYAGENO+PORT -->> CONTAINNO                     (LOAD       )

The following FDs form a cycle:
                     VOYAGENO ---> VESSELNO
                VOYAGENO+PORT ---> VOYAGENO
                   CONTAINNO ---> VOYAGENO+PORT
           CONTAINNO+VOYAGENO ---> CONTAINNO
                         BOL ---> CONTAINNO+VOYAGENO
                         BOL ---> VESSELNO

All secondary key accesses are feasible.
```

List of final dependencies:

```
                VESSELNO ---> VESSELNAME                (           )
                VESSELNO ---> TONNAGE                   (           )
                VESSELNO ---> DETAILS                   (           )
                VESSELNO ---> COUNTRY                   (           )
                VESSELNO ---> VESOWNER                  (           )
                VOYAGENO ---> VESSELNO                  (           )
           VOYAGENO+PORT ---> VOYAGENO                  (           )
                     BOL ---> CONTAINNO+VOYAGENO        (Assumed FD )
           VOYAGENO+PORT ---> DEPARTDATE                (           )
      CONTAINNO+VOYAGENO ---> PORT                      (DESTINATION)
                     BOL ---> SHIPPER                   (           )
                     BOL ---> CONSIGNEE                 (           )
                     BOL ---> VESSELNO                  (           )
               CONTAINNO ---> SIZE                      (           )
                     BOL ---> DELIVDATE                 (           )
                     BOL ---> BILLDATE                  (           )
                     BOL ---> TOTCHARGE                 (           )
                     BOL ---> CONSIGNNO                 (           )
               CONTAINNO ---> CONTOWNER                 (           )
               CONTAINNO ---> TYPE                      (           )
      CONTAINNO+VOYAGENO ---> ARRIVDATE                 (           )
                  ITEMNO ---> NOPIECES                  (           )
                  ITEMNO ---> COMMODCODE                (           )
                  ITEMNO ---> WEIGHT                    (           )
                  ITEMNO ---> CHARGE                    (           )
      CONTAINNO+VOYAGENO ---> CONTAINNO                 (DMNS added )
      CONTAINNO+VOYAGENO ---> VOYAGENO                  (DMNS added )
               CONTAINNO ---> HANDINSTR                 (           )
      CONTAINNO+VOYAGENO ---> PORT                      (ORIGINATION)
      CONTAINNO+VOYAGENO ---> SAILDATE                  (           )
                     BOL ---> AGENT                     (           )
               CONTAINNO ---> VOYAGENO+PORT             (Assumed FD )
                  ITEMNO ---> BOL                       (Assumed FD )
                VESSELNO -->> VOYAGENO                  (           )
                VOYAGENO -->> VOYAGENO+PORT             (           )
                VESSELNO -->> BOL                       (Assumed MVD)
           VOYAGENO+PORT -->> CONTAINNO                 (UNLOAD     )
                     BOL -->> ITEMNO                    (           )
                   AGENT -->> CONTAINNO+VOYAGENO        (           )
               CONTAINNO -->> CONTAINNO+VOYAGENO        (           )
                VOYAGENO -->> CONTAINNO+VOYAGENO        (DMNS added )
                 SHIPPER -->> BOL                       (           )
           VOYAGENO+PORT -->> CONTAINNO                 (LOAD       )
      CONTAINNO+VOYAGENO -->> BOL                       (           )
```

*** Output Phase ***

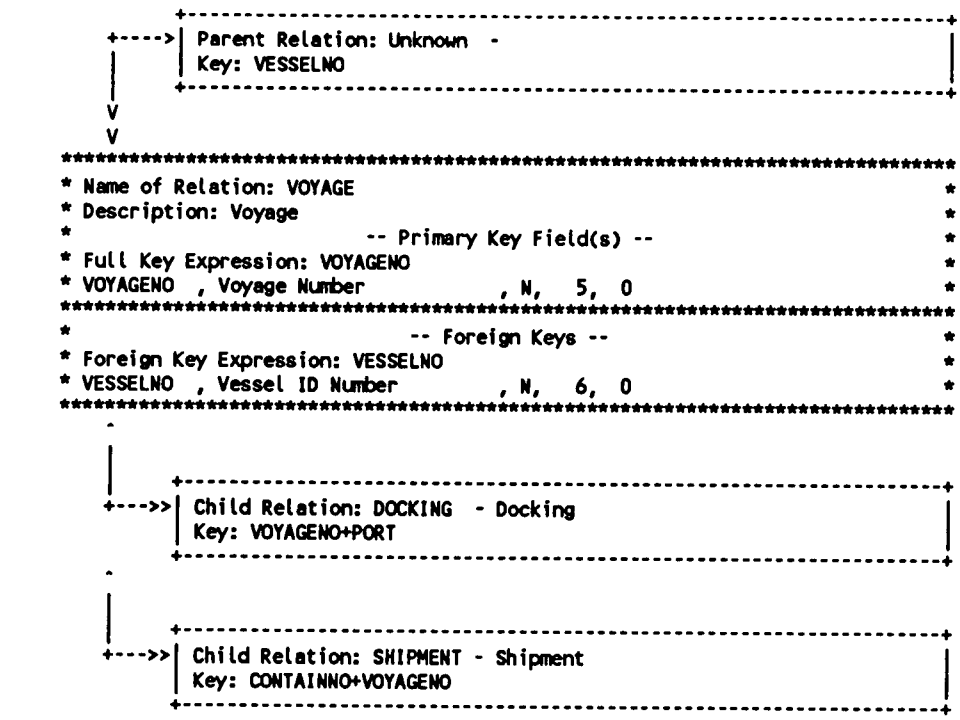
Overall Schema Report.



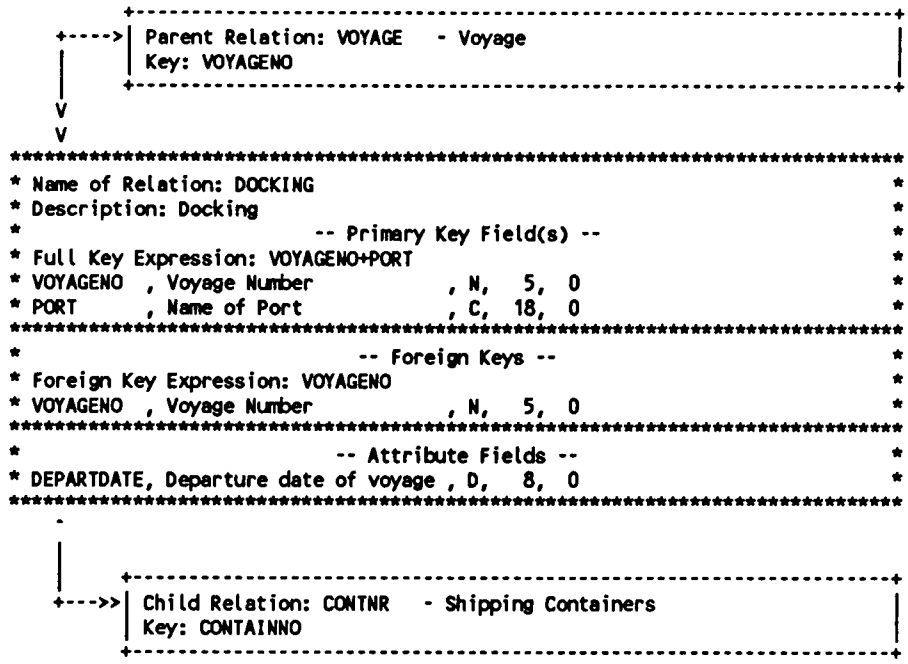
Relation in Context for Relation: Unknown

```
**********************************************************************************
* Name of Relation: Unknown                                                      *
* Description:                                                                   *
*                          -- Primary Key Field(s) --                            *
* Full Key Expression: VESSELNO                                                  *
* VESSELNO  , Vessel ID Number          , N,   6,  0                             *
**********************************************************************************
*                           -- Attribute Fields --                               *
* VESSELNAME, Name of Vessel            , C,  20,  0                             *
* TONNAGE   , Rated tonnage of vessel   , N,   5,  0                             *
* DETAILS   , Bill of Laden Details     , C,  25,  0                             *
* COUNTRY   , Country of Registry       , C,  12,  0                             *
* VESOWNER  , Owner of Vessel           , C,  15,  0                             *
**********************************************************************************
    .
    |
    |     +-------------------------------------------------------------+
    +--->>| Child Relation: VOYAGE   - Voyage                           |
          | Key: VOYAGENO                                               |
          +-------------------------------------------------------------+
    .
    |
    |     +-------------------------------------------------------------+
    +--->>| Child Relation: BOL      - Bill of Laden                    |
          | Key: BOL                                                    |
          +-------------------------------------------------------------+
```

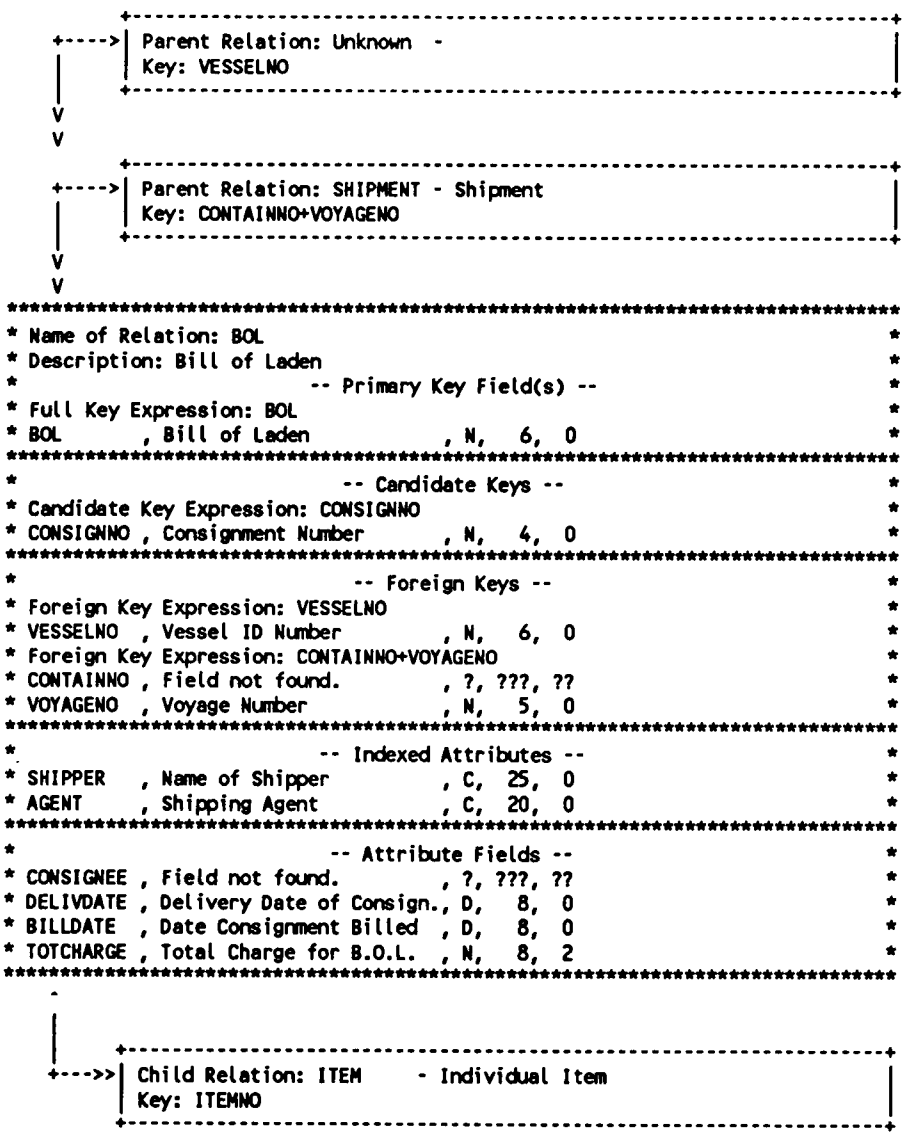
Relation in Context for Relation: VOYAGE



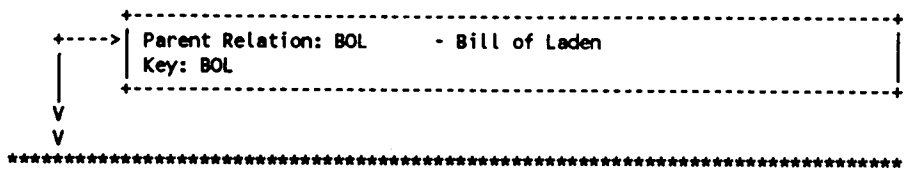
Relation in Context for Relation: DOCKING



Relation in Context for Relation: BOL

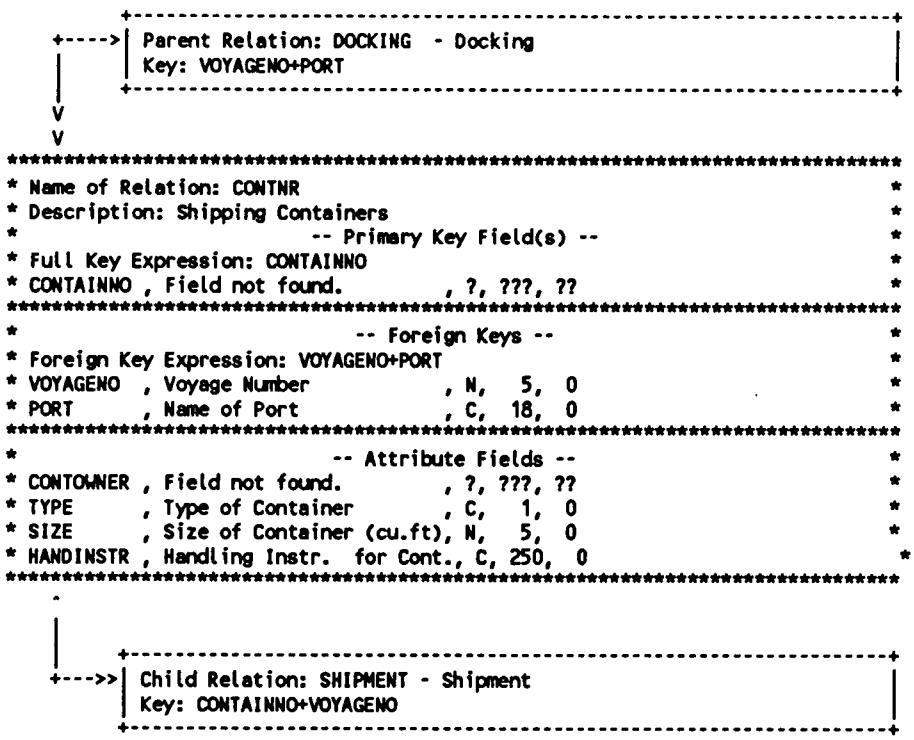


Relation in Context for Relation: ITEM

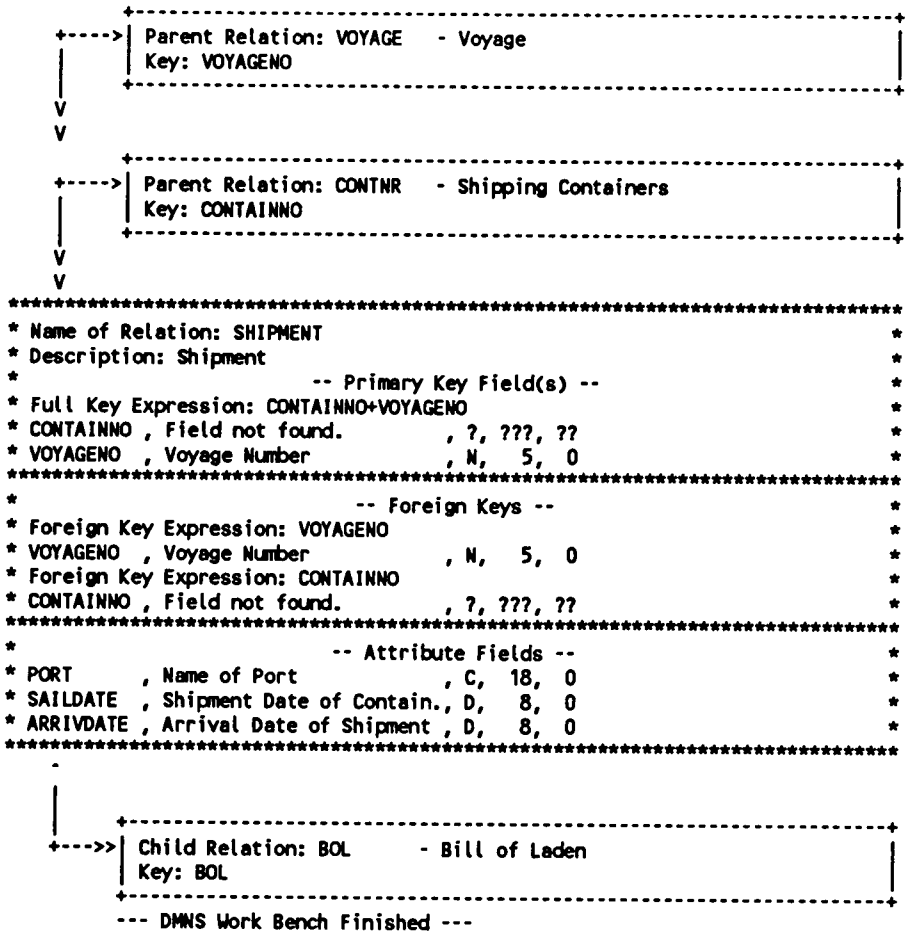



```
**********************************************************************************
* Name of Relation: ITEM                                                         *
* Description: Individual Item                                                   *
*                          -- Primary Key Field(s) --                            *
* Full Key Expression: ITEMNO                                                    *
* ITEMNO    , Item Number of Package   , N,   8,  0                              *
**********************************************************************************
*                            -- Foreign Keys --                                  *
* Foreign Key Expression: BOL                                                    *
* BOL       , Bill of Laden            , N,   6,  0                              *
**********************************************************************************
*                          -- Attribute Fields --                                *
* NOPIECES  , No.  of Individual Pieces , N,   4,  0                             *
* COMMODCODE, Commodity Code of Item    , C,   3,  0                             *
* WEIGHT    , Weight of Item (tons)     , N,   5,  1                             *
* CHARGE    , Charge for item shipment , N,   6,  2                              *
**********************************************************************************
```

Relation in Context for Relation: CONTNR



Relation in Context for Relation: SHIPMENT



VITA

Burl Timothy Rhyne was born in Maryville, Tennessee, on October 24, 1956. He attended Everett High School from which he graduated in June 1974. The following September he entered The University of Tennessee from which he received a Bachelor of Science degree in Industrial Engineering in August 1979. During the course of his attendance at the university he participated in the Co-operative Engineering Program, working at Tennessee Eastman Company in Kingsport, Tennessee, for six quarters.

Following graduation, Mr. Rhyne accepted employment with Union Carbide Corporation's Nuclear Division (now Martin Marietta Energy Systems) at the Department of Energy's Y-12 Plant in Oak Ridge, Tennessee, as a Management Information Systems Engineer. Later assignments included Data Resources Engineer at Y-12, Computing Analyst at the Oak Ridge National Laboratory (ORNL), and currently Computing Specialist and Section Head of the Microcomputer Applications Section located at ORNL.

In 1985 the author began taking courses at The Graduate School of The University of Tennessee, Knoxville, through the Oak Ridge Resident Graduate Program, towards a master's degree in computer science. This degree was awarded in August 1988.