1

CANADIAN THESES ON MICROFICHE

I.S.B.N.

THESES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

**THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED**

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

**LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE**

135

0-315-12103-3

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Division Division des thèses canadiennes

Ottawa, Canada
K1A 0N4

60016

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

RICHARD MAREK WILLIAMS

Date of Birth — Date de naissance

JANUARY 11, 1953

Country of Birth — Lieu de naissance

CANADA

Permanent Address — Résidence fixe

11 BAYSIDE
OTTAWA, ONT
K1V 93R

Title of Thesis — Titre de la thèse

THE DESIGN AND IMPLEMENTATION OF A CONCEPTUAL SCHEMA
DESIGN AID FOR RELATIONAL DATABASES.

University — Université

CARLETON UNIV., OTTAWA, ONT.

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

MASTER OF ENGINEERING

Year this degree conferred — Année d'obtention de ce grade

1982

Name of Supervisor — Nom du directeur de thèse

J.S. RORDON

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

Date

13 SEP 82

Signature

The Design and Implementation of a
Conceptual Schema Design Aid
for
Relational Databases

by

© Richard Marek Williams, B.Eng.

A Thesis submitted to the Faculty of
Graduate Studies in partial fulfilment
of the requirements for the degree of

Master of Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
Ottawa, Ontario

August, 1982

The undersigned recommend to the Faculty of Graduate
Studies and Research acceptance of the Thesis

"THE DESIGN AND IMPLEMENTATION OF A CONCEPTUAL
SCHEMA DESIGN AID FOR RELATIONAL DATABASES"

submitted by Richard Marek Williams in partial
fulfillment of the requirements for the degree of
Master of Engineering.

J.S. Riordon
Thesis Supervisor

B. Pagurek
Chairman,
Department of Systems
and Computer Engineering

Carleton University

Ottawa, Ontario

14 September 1982

ABSTRACT

This research is concerned with the development and implementation of an automated conceptual schema design aid for relational databases. Based on the concepts of logical normal forms the technique of schema decomposition is developed into a practical and feasible software tool. This tool is then shown to accept a broader range of database descriptions than those currently discussed in the literature, producing conceptual schema designs which are both logical and efficient.

The development of the principal algorithm embodied in the design aid is fully described and contrasted with three algorithms found in the current literature. As necessary background for the schema design algorithm we present a review of the relational model, normalization and several fundamental supporting algorithms including an original and improved algorithm for the determination and manipulation of the dependency basis. Finally, both the capabilities and limitations of binary decomposition methods in general and this design aid in particular are discussed.

Acknowledgements

I would like to express my sincere thanks to all those who have played some part in the developement of this work.

To my thesis advisor, Dean J. S. Riordon, goes my appreciation for his careful and patient support and encouragement.

To my fellow Systems graduate students - E. Sinyor, P. Jamieson and Dr. S. L. Janssen - go my thanks for their assistance over many hours of fruitful discussion.

I must also add a note of special appreciation to my parents for their constant encouragement and love over these many years.

This work was supported by the Department of National Defence.

To my wife Peg -

for her love and support
through the difficult times.

Table of Contents

Abstract	iii
Acknowledgements	iv
Table of Contents	vi
List of Figures	ix
	Page
CHAPTER 1 THE RELATIONAL DATABASE DESIGN PROBLEM	
1.1 Introduction and Thesis Organization	1-1
1.2 Relational Database Design - An Overview	1-3
1.3 The Relational Model	1-7
1.3.1 Basic Definitions	1-7
1.3.2 Data Dependencies	1-10
1.3.3 Common Assumptions	1-25
1.3.4 Representation and Redundancy	1-27
1.4 Normalization	1-30
1.4.1 Introduction	1-30
1.4.2 Anomalies	1-31
1.4.3 Normal Forms	1-32
1.5 The Design Process	1-36
1.6 Current Research	1-39
CHAPTER 2 PRINCIPAL ALGORITHMS IN SCHEMA DESIGN	
2.1 Introduction	2-1
2.2 The Membership Problem	2-2
2.2.1 Functional Dependency (FD) Membership Algorithm	2-4
2.2.2 Multivalued Dependency (MVD) Membership Algorithm	2-8
2.3 A 1NF Size Estimator for Relational Schemes	2-18
2.4 Query Cost Algorithm	2-25
2.4.1 Implementation Details	2-25
2.4.2 Query Processing Strategies	2-30

CHAPTER 3 SCHEMA DECOMPOSITION ALGORITHMS

3.1 Introduction	3-1
3.2 The Bernstein Synthesis Algorithm	3-2
3.3 The Zaniolo and Lien Decomposition Algorithms	3-10
3.3.1 Introduction	3-10
3.3.2 A Representation Priority Approach	3-12
3.3.3 A Normalization Priority Approach	3-20
3.3.4 Summary	3-29
3.4 Compromise Schema Decomposition Algorithm	3-31
3.4.1 Introduction	3-31
3.4.2 Description	3-32

CHAPTER 4 SOFTWARE DESCRIPTION

4.1 Introduction	4-1
4.2 System Architecture	4-2
4.3 A Typical User Session	4-5
4.3.1 Introduction	4-5
4.3.2 Entering the Data Base Description	4-7
4.3.3 Evaluating the Output	4-12
4.4 Main Module	4-17
4.4.1 Introduction	4-17
4.4.2 The RDD Data Structure	4-17
4.4.2.1 Attributes	4-18
4.4.2.2 Dependencies	4-20
4.4.2.3 The Schema Representation	4-22
4.4.2.4 Data Lists	4-23
4.4.3 Library Routines	4-23
4.5 Input Module	4-24
4.5.1 Procedure INPUT ATTRIBUTES	4-24
4.5.2 Procedure INPUT DEPENDENCIES	4-26
4.5.3 Procedure INPUT QUERIES	4-27
4.6 Decomposition Module	4-27
4.7 Dependency Redundancy Module	4-28
4.8 Query Cost Module	4-29
4.9 Scheme Size Module	4-29

CHAPTER 5 DATABASE DESIGN SESSIONS USING RDD

5.1 Introduction	5-1
5.2 Example 5.1	5-2
5.3 Example 5.2	5-8
5.4 Example 5.3	5-13
5.5 Example 5.4	5-18
5.6 Example 5.5	5-23
5.7 Limitations of the Decomposition Approach	5-27

CHAPTER 6 CONCLUSIONS AND SUGGESTIONS
FOR FURTHER RESEARCH

6.1 Thesis Contributions	6-1
6.1.1 Schema Design Algorithm	6-2
6.1.2 Dependency Basis Algorithm	6-4
6.1.3 Implementation of a Schema Design Aid	6-4
6.2 Suggestions for Further Research	6-6

Appendices

A	A Summary of Significant Module Routines	A-1
	A.1 Procedures Library Module	A-1
	A.2 Input Module	A-3
	A.3 Scheme Decomposition Module	A-3
	A.4 Dependency Redundancy Module	A-6
	A.5 Query Cost Module	A-7
	A.6 Scheme Size Module	A-8

References

R-1

List of Figures

1-1	An Example of the User View and Conceptual Schema Layers	1-6
1-2	Terminology - Relation, Scheme, Schema	1-9
1-3	Sample Data Base Relation	1-11
1-4	ER Representation for Figure 1-3	1-12
1-5	Relation R1 Decomposed Into R11 and R12	1-21
1-6	Use of an EMVD outside its context	1-21
1-7	A Lossy Join with Erroneous Tuples	1-22
1-8	Example 1.3 (a) Sample Relation and ER Representation (b) Incorrect use of an MVD	1-23
1-9	An example of a Redundant Decomposition	1-30
1-10	An example of a 3 Join Dependency	1-35
2-1	Derivation Tree Analysis	2-5
2-2	A Linear Time Membership Algorithm for Functional Dependencies	2-7
2-3	Basic Dependency Basis Algorithm	2-10
2-4	Dependency Basis For Example 2.1 In Binary Tree Form	2-12
2-5	The Modified Dependency Basis of Example 2.1	2-13
2-6	Improved Basis Algorithm	2-14
2-7	Example 2.2 - After Initialization	2-16
2-8	Example 2.2 - After g1 Modifies Basis	2-16
2-9	Example 2.2 - After g3 Modifies Basis	2-16
2-10	An Algorithm to Determine the Estimated Size of a given Relation R	2-22
2-11	Example 2.5 - 1NF Size of a given relation	2-24
2-12	Summary of Physical Storage Details	2-28
2-13	Graph Representation Approach	2-34
2-14	4NF Decomposition Tree Approach	2-35
3-1	The Bernstein 3NF Synthesis Algorithm in Simplified Form	3-3
3-2	Example 3.1 - The Office Database (a) Entity-Relationship Description (b) Dependency Description	3-4
3-3	Example 3.1 - Dependency Grouping (a) Initial Grouping (b) Final Grouping	3-7
3-4	Example 3.1 - Final Schema Design	3-8
3-5	A Representation Priority Algorithm by Zaniolo	3-14
3-6	Final Decomposition Tree for Example 3.2	3-18
3-7	A Normalization Priority Algorithm by Lien	3-26
3-8	Example 3.4 - ER Representation	3-27
3-9	The Compromise Decomposition Algorithm	3-32
4-1	Relational Data Base Design Aid (RDD) - System Architecture	4-4
4-2	Attribute Entry Phase (a) Response to Missing Information (b) Illegal Characters in Input (c) Numeric Values Beyond Limits	4-7 4-8 4-8

(d)	Duplicate Entry Checks	4-9
(e)	User Help - Attribute Summary	4-9
(f)	The Complete Attribute Summary	4-10
4-3	Dependency Entry Phase	
(a)	Attribute Identifiers - Case Significance	4-10
(b)	Dependency Input Error Handling	4-11
(c)	Final Dependency Summary	4-11
4-4	Query Entry Phase and Summary	4-12
4-5	Design Phase	
(a)	User Prompt - Start of Design Phase	4-12
(b)	Example 4.1 - Basic Design Output	4-13
(c)	Example 4.1 - Final Schema Design	4-15
4-6	Sample Output at Level 2	4-15
4-7	Sample Output at Level 4	4-16
4-8	Pascal Set Operations	4-19
4-9	Dependency Processing Data Structure	4-22
5-1	Example 5.1 - Data Base Description	5-3
5-2	Example 5.1 - Sample Design Output	5-7
5-3	Example 5.2 - Data Base Description	5-9
5-4	Example 5.2 - Sample Design Output	5-12
5-5	Example 5.3 - Data Base Description	5-14
5-6	Example 5.3 - Sample Design Output	5-17
5-7	Example 5.4 - Data Base Description	5-19
5-8	Example 5.4 - Sample Design Output	5-22
5-9	Example 5.5 - Data Base Description	5-24
5-10	Example 5.5 - Sample Design Output	5-26
5-11	Example 5.6 - A Realistic Design Problem	5-28

CHAPTER 1

THE RELATIONAL DATABASE DESIGN PROBLEM

1.1 INTRODUCTION AND THESIS ORGANIZATION

One of the most significant trends in modern computing is clearly the rapidly increasing demand for access to very large stores of non-numerical data - the 'database'. The traditional concept of the database as a vast library accessible only to application programmers is now outdated. In its place there are sophisticated database management systems (DBMS) which offer services at a range of levels to meet the requirements of the user, database administrator (DBA) and system maintainer, which enable easier and more effective use of the intrinsic 'information' value of the data. As a result personnel with very little basic training are now allowed access to extremely large stores of data in the course of their duties.

Database management systems do not simply provide access to the raw data itself but provide a means to 'map' the data of the real world system into the hardware and software implementation in such a way that access is both logical and efficient and that the costs of maintaining the data in a consistent state are minimized. Two key developments continue to hasten this trend: firstly, the growing demand for products to support the 'office of the future' is placing terminals in more and more locations within the business environment. Secondly, at the same time, the decrease in cost per bit of on-line mass storage is bringing about an increase in both the number of database installations and their average size. It is therefore a demanding role for the DBA to maintain a logical and efficient system under these conditions.

This work deals with design and implementation of an automated tool to assist the DBA to design and maintain an efficient conceptual schema. We begin by presenting the concepts of the relational model and logical normalization in the first chapter. In the second chapter we present a series of algorithms which are used to deal with the basic elements of the design problem: the attributes, dependencies and queries. In Chapter 3 we then describe the development of the complete design algorithm comparing it with three current methods. Next, in Chapter 4, we describe the Relational Database Design Aid (RDD) software which implements the design algorithm. Chapter 5 then

presents a series of examples to highlight both the capabilities and limitations of the design aid. Finally we conclude in Chapter 6 with a summary of the contributions of the work and suggestions for further research in this area.

1.2 RELATIONAL DATABASE DESIGN - AN OVERVIEW

In order to describe information in a computer oriented manner two groups of conceptual data models have been developed. The first group emphasizes the 'datalogical' content of the database which is to say that it is more concerned with what the data elements look like and how they are to be linked together. These include the original hierarchical, network and relational models although the latter has the greatest degree of data independence and therefore many of the desirable properties found in the second group. This more recent group, called 'infological', is, as the name suggests, more concerned with the information content of the data.

The relational model was introduced by E.F. Codd in 1970 [CODD70] and has received considerable attention and research effort. Commercial relational products are now available for all sizes of computers (micro, mini and mainframe) and, perhaps more significantly, in an effort to capture the customers' attention, many older non-relational products often cite a supposed 'relational appearance' to the user. The apparent simplicity to

the user and inherent flexibility are the main reasons for the success of relational products.

The Entity-Relationship model described in [CHEN80] is the most successful of the infological models and will also be discussed in this work. There are two distinct elements in the ER model - the entity and the relationship. An entity is a conceptually concrete object such as an employee, a project or a department, while a relationship is a semantic link between two entities representing a one-to-one, one-to-many or many-to-many correspondence. These are shown in graphical form using rectangles for entities and diamonds for relationships with a descriptive caption to explain their meaning. For example we may refer to the one-to-many correspondence between the entities employee and department as 'works on' to indicate the nature of the relationship. We then add a third figure, the circle, to enclose unique attribute labels that are associated with either an entity or relationship. Hence the entity Employee will be shown with the unique attributes Employee Name or Address while a relationship between employees and projects might possess information such as 'Hours Worked' or 'Supervisors Name'. Analysis of these ER diagrams augments the relational approach by graphically presenting these correspondences in a clean and concise manner.

To support a layered approach to the design and

implementation of a DBMS we follow the three level architecture put forward in the ANSI X3/SPARC document [ANSI75] which describes:

1. the user view or enterprise schema;
2. the conceptual schema; and
3. the physical schema.

The focus of this work is the conceptual schema. In the relational model a 'schema' is described as a series of tables or relations each with a set of column headings to describe the material contained in the rows of the table. It is through this description that the DBA exercises the majority of his control of the database. In addition to describing what combination of column headings he will define for his system the DBA designates certain columns as 'keys', creates various 'index tables' and assigns the authority to read, insert or modify to the system users. The table or 'relation' is therefore a basic unit of the relational system and the schema is an overall map of these relations. The 'user view' is also defined as a series of relations although they may differ from the underlying conceptual schema in many ways in order to simplify the users understanding of what data is available, to protect some of the data from unauthorized access or to summarize or pre-process the data in some manner. Fig. 1-1 highlights an example of these two levels. In this case perhaps for reasons of security the user is not allowed access to the performance rating of the manager or to

the actual budget which is currently authorized. Furthermore, although to the user it appears that only one table is involved, he is actually using data from three relations in the conceptual schema.

The User View

DEPT_NO	DEPT_NAME	MGR_NAME	EMP_NO	EMP_NAME	BUDGET
---------	-----------	----------	--------	----------	--------

The Conceptual Schema

DEPT_NO	DEPT_NAME	MGR_NAME	MGR_PERFORMANCE_RATING
DEPT_NO	EMP_NO	EMP_NAME	
DEPT_NO	REQUESTED_BUDGET	ALLOTTED_BUDGET	

An Example of the User View and Conceptual Schema Layers
Figure 1-1

Below the conceptual schema lies the physical schema where we define the actual storage details, create the index tables and establish the searching techniques. Relational DBMS do not allow the DBA very much control over these internal activities although he can usually select the indices and determine whether the relations will be stored in adjacent disc storage areas (called clusters) to improve the performance of the system.

The problem facing the database designer therefore is that of effectively dealing with the large number of design alternatives presented by the data that constitutes the information system. An automated tool to assist the DBA or

database designer in this task - whether it is part of the DBMS or an independent product - would be of great help when the number of alternatives become too numerous for simple analysis. The implementation of an automated tool to assist in this task and a discussion of its usefulness is the aim of this work.

1.3 THE RELATIONAL MODEL

1.3.1 BASIC DEFINITIONS

A brief summary of the definitions and assumptions used throughout this work is given in this section. We attempt to conform to the accepted nomenclature found in the current references. The basic element of the relation is the unique attribute ' a_m ' chosen from a fixed set

$$\{ a_1, a_2, a_3 \dots a_n \}.$$

We will use the lower case to represent a single attribute and upper case to indicate a set of attributes such as

$$B = \{ a_1, a_2, a_3 \dots a_{10} \}.$$

The letter 'R' will also be used to describe both a relation and a scheme where the context will clarify the particular case. Each attribute is given a unique identifier such as PART NUMBER, or EMPLOYEE NAME and has associated with it a specific domain of acceptable values from which the attribute assumes a particular value. We refer to the attribute EMPLOYEE NAME and to the value

Mr. Jones. The meaning should be clear in the context. We therefore write that the value :

$$a_i \in \text{dom}(a_i)$$

We define a relation $R(X)$ as a set of m -tuples (or records) where $m = |X|$ and $R(X)$ is a subset of the Cartesian product

$$\text{dom}(a_1) \times \text{dom}(a_2) \times \dots \times \text{dom}(a_m).$$

Thus the relation can be considered to be a table of information having m columns with each m -tuple being a distinct row of the table.

We will use the term 'scheme' to represent only the attribute (or column heading) information while 'relation' refers to a table filled with tuples. A 'schema' consists of all schemes associated with the database of interest. The order of attributes within the relation is immaterial. An example of this is presented in Fig. 1-2.

A RELATION

PART_NO	DESCRIPTION	PRICE
1100110	Wheelbarrow	\$45.95
2223022	Lawn Mower	\$99.99
4512334	Hedge Trimmer	\$35.25
2678909	Wheelbarrow	\$42.95

A SCHEME

PART_NO	DESCRIPTION	PRICE
---------	-------------	-------

A SCHEMA

PART_NO	DESCRIPTION	PRICE
PART_NO	SUPPLIER	
SUPPLIER	LOCATION	

Terminology - Relation, Scheme, Schema
Figure 1-2

We define the following operations on the relation $R(X)$:

PROJECT The projection of $R(X)$ on attribute set Y is written as

$$R[Y] \text{ where } Y \subseteq X.$$

This may simply be thought of as the selection of only the columns headed by attributes in the set Y with all duplicate rows removed.

NATURAL JOIN The natural join of $R_1(A)$ and $R_2(B)$ is defined,

$$\text{if and only if } A \cap B \neq \emptyset$$

and is written $A*B$. The term equi-join implies that chosen tuples in A and B have the same values for their intersecting attributes. The use of a predicate may impose other conditions on a natural join such as an acceptable range of values ($0 \leq a_i \leq K$) or specific value ($a_i = K$).

RESTRICTION We restrict the admissibility of tuples by defining a specific value or range of values for attributes within the scheme. Thus a restriction reduces the number of tuples selected while projection reduces the number of attributes to consider within a given tuple.

1.3.2 DATA DEPENDENCIES

Having defined the attributes of the data which will comprise the database we must now consider the semantic constraints which are present in the information to be maintained. We will consider semantic constraints presented as data dependencies of the following forms:

1. functional dependencies (FD);
2. multivalued dependencies (MVD); and briefly discuss
3. join dependencies.

Although there are many other dependencies discussed in the literature ([AROR81] lists 17 of them) these are by far the most important.

EXAMPLE 1.1 To demonstrate these dependencies we present the sample relation in Fig. 1-3. This relation might well represent a small instance of an engineering firm's database where administrative and personnel information is maintained. The parallel ER diagram is shown in Fig. 1-4.

EMP#	EMP_NAME	DEPT#	PROJ#	HOURS	SKILL#	CHILD_NAME
111	Brown	Car	A	3.5	Machinist	Mary
111	Brown	Car	A	3.5	Welder	Mary
111	Brown	Car	A	3.5	Machinist	Sue
111	Brown	Car	A	3.5	Welder	Sue
111	Brown	Car	B	1.0	Machinist	Mary
111	Brown	Car	B	1.0	Welder	Mary
111	Brown	Car	B	1.0	Machinist	Sue
111	Brown	Car	B	1.0	Welder	Sue
222	White	Paint	C	5.5	Painter	Alice
333	Black	Paint	C	2.5	Painter	David

Sample Database Relation
Figure 1-3

Entity-Relationship Representation
of Figure 1-3
Figure 1-4

Let us now describe which of these correspondences can be interpreted using these dependencies.

We refer to a functional dependency (FD) and write $X \rightarrow Y$ (read as X determines Y) if and only if for each value of X we have associated only one value of Y .

This models the classic one-to-one correspondence. A full functional dependency $X \rightarrow Y$ exists if no proper subset of X , say X' determines Y or $X' \rightarrow Y$. Referring to the example database of Fig. 1-3, we state that the FDs

1. $EMP\# \rightarrow EMP_NAME$
2. $EMP\# \rightarrow DEPT\#$; and
3. $EMP\# PROJ\# \rightarrow HRS$

hold in the database from our semantic knowledge of the situation. Notice that it is not immediately apparent from the dependency whether it represents a semantic link between entities, attributes or relationships and so all dependencies must be considered of equal importance. Since none of the tuples violate the dependency constraints they are all considered to be valid and the relation accurately reflects the real world situation. Furthermore it is important to note that the existence of dependencies should not necessarily be inferred from the tuples found in the relation but are in fact imposed from the real world semantics.

The nature of functional dependencies has been studied extensively [ARMS74] [BEER77] [BEER78] and a proven complete axiomatization has been realized. It has therefore been proven that the following inference rules hold for FDs :

1. FD1 Reflexitivity IF $Y \subseteq X$ then $X \rightarrow Y$
2. FD2 Augmentation IF $X \rightarrow Y$ and $Z \subseteq W$ then $XW \rightarrow YZ$
3. FD3 Transitivity IF $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$
4. FD4 Pseudo-Transitivity IF $X \rightarrow Y$ and $YW \rightarrow Z$ then $XW \rightarrow Z$
5. FD5 Union IF $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$
6. FD6 Decomposition IF $X \rightarrow YZ$ then $X \rightarrow Y$ and $X \rightarrow Z$.

The juxtaposition of attributes AB implies the union of A and B or $A \cup B$ in all cases.

We also note that FDs possess projectivity and inverse projectivity; that is, for $X \rightarrow Y$:

1. if $X \rightarrow Y$ holds in $R(S)$ then $X \rightarrow Y$ holds in $R(S')$ where $S' \subseteq S$ and $X \in S'$; and
2. if $X \rightarrow Y$ holds in $R(S')$ then $X \rightarrow Y$ holds in $R(S)$, where $S' \subseteq S$.

Based on the above inference rules we may develop the set of all derivable FDs from a given set F. Because these rules form a

complete system [BEER77] we refer to this set of dependencies as the closure of F and represent it with the symbol F^+ . A 'covering' is a set of dependencies such that we may derive the closure from it using the above inference rules. We write this as any set T' such that $T'^+ = T^+$. There are a finite number of possible covers to consider for any given database. A covering is nonredundant (or minimal) if no proper subset forms a cover. The group of dependencies that we are initially given must contain a cover or else we cannot infer the remaining valid dependencies and the final design will be in error.

The 'key' of a relation R is a set of attributes X such that $X \rightarrow R \in T^+$ and thus it uniquely determines each distinct tuple in R . A 'superkey' is any set of attributes S such that S contains a key plus other attributes as well. All members of a key are called 'prime attributes'.

Obviously not all relationships are functional in nature and an extension is therefore required to model both the one-to-many and many-to-many correspondences described in the ER diagram. A special class of dependency called 'multivalued' has been shown to model certain cases of these correspondences [FAGI79a] [ZANI81]. A multivalued dependency (MVD) $X \twoheadrightarrow Y$ (read as X multidetermines Y) is said to exist in the context $X+Y+Z$ if, for each value of X that we choose, the associated values of Y are the same independent of our choice of Z . Expressed in general

terms, the sets of values $\{X', Y'\}$ formed from the operation defined by:

$$R(X_i, Y_j, Z_k) [XY]$$

is the same for all valid Z_k values. Not all one-to-many or many-to-many correspondences meet this criteria and so MVD's are not as readily useful as we might like them to be. The example in Fig. 1-3 highlights two valid instances which we may write as:

1. EMP#->->SKILL - implies that each employee possesses a distinct set of skills; and
2. EMP#->->CHILD_NAME - implies that each employee may have more than one child.

We therefore observe in Fig. 1-3 that tuples exist to show that Mr. Brown is a machinist independent of the choice of the values of his CHILD_NAME attributes.

Complete axiomatization of MVDs is also cited in [BEER77] with the following inference rules for the general case where $X \cup Y \cup Z = R$ and $Y \cap Z \subseteq X$:

1. MVD0 Complementation	$X \twoheadrightarrow Y$ holds in context R where $R = X+Y+Z$ if and only if $X \twoheadrightarrow Y$ and $X \twoheadrightarrow Z$
2. MVD1 Reflexitivity	IF $Y \subseteq X$ then $X \twoheadrightarrow Y$
3. MVD2 Augmentation	IF $X \twoheadrightarrow Y$ and $Z \subseteq W$ then $XW \twoheadrightarrow YZ$
4. MVD3 Transitivity	IF $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$ then $X \twoheadrightarrow Z$
5. MVD4 Pseudo-Transitivity	IF $X \twoheadrightarrow Y$ and $YW \twoheadrightarrow Z$ then $XW \twoheadrightarrow Z-YW$
6. MVD5 Union	IF $X \twoheadrightarrow Y$ and $X \twoheadrightarrow Z$ then $X \twoheadrightarrow YZ$
7. MVD6 Decomposition	IF $X \twoheadrightarrow Y$ and $X \twoheadrightarrow Z$ then $X \twoheadrightarrow Y \cap Z$, $X \twoheadrightarrow Y-Z$, and $X \twoheadrightarrow Z-Y$

and two additional rules which link FD and MVDs

1. FD-MVD1 Duality	IF $X \rightarrow Y$ then $X \twoheadrightarrow Y$
2. FD-MVD2 Generation	If $X \twoheadrightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z-Y$

FD-MVD2 is the only rule which can generate new FDs from MVDs. The closure due to MVDs is given the symbol G^* although in some references this implies the closure due to both FDs and MVDs.

The most useful properties of the multivalued dependency are complementation and decomposition (MVD0 and MVD6) which we can use to effect the decomposition of a scheme by dividing it into 2

sub-schemes, without the loss of any of our original information and with a significant reduction in the duplication of data.

Given :

$R(X, Y, Z)$ with $X \twoheadrightarrow Y$

from MVDO we infer $X \twoheadrightarrow Z$ and decompose R into :

$R_1(X, Y)$ and $R_2(X, Z)$.

While MVDs possess the projectivity property, they do not possess inverse projectivity because of the context within which the dependency is said to be valid. That is, we cannot infer that a dependency which is valid in a subset of the relation may necessarily hold in the entire relation.

We can therefore have MVDs with a reduced context which are 'embedded' or 'latent' within the scheme and only hold in projections which are subsets of the scheme. Complete axiomatization of these dependencies has not been proven and some references claim it is ultimately impossible [BEER80]. For the purposes of notation let us assume that $X \twoheadrightarrow Y$ implies that the context for the dependency includes all valid projections R' of the entire scheme R provided:

1. $X \subseteq R'$; such that all left side attributes are present;
2. $Y \cap R' \neq \emptyset$; at least one attribute in the right side is present.

An alternate notation may be used to detail all of the MVD's with

the same left side and the same context. We write this as follows: (note the ':' instead of the '->->' symbol)

A : B | C | D | ... | Z for A,B,C,D .. Z disjoint..

We read this as $A \rightarrow \rightarrow B$, $A \rightarrow \rightarrow C$, $A \rightarrow \rightarrow D$, ..., $A \rightarrow \rightarrow Z$ which hold in the relation containing $A \cup B \cup C \cup D \cup \dots \cup Z$ and no other attributes. Recognizing that the MVD or EMVD exists and defining its proper context is very difficult in some situations.

We now present two examples where the incorrect use of an MVD or EMVD can cause erroneous results. In the first we show the use of an EMVD outside its proper context while the second shows the results when an MVD is used to incorrectly represent a many-to-many correspondence.

EXAMPLE 1.2 An embedded MVD (EMVD) exists in example 1.1 and can be written:

DEPT# : EMP# EMP_NAME CHILD_NAME SKILL# | PROJ#

where we are interested in the situation now that there is no attribute linking PROJ# and EMP# (by projecting out HOURS). The context is therefore the scheme R1 (which is R less the attribute HOURS). We now wish to state that a DEPT has many employees and many projects and that all employees in that department work on all of the department's projects. Considering only tuples for EMP_NAME = Brown we can use MVD6 and show the decomposition of R1 into R11 and R12 in Fig. 1-5. A subsequent equi-join on EMP# returns

the original relation R_1 . Fig. 1-6, on the other hand, shows the decomposition using the EMVD outside its valid context (in this case the original relation R). Fig. 1-7 shows the results of a subsequent equi-join on DEPT and the appearance of erroneous tuples.

EXAMPLE 1.3 Consider the relation R and its equivalent ER diagram shown in Fig. 1-8(a). Let us assume that the designer recognizes the many-to-many relationship between INSTRUCTOR and STUDENT and so defines the MVD $\text{INSTRUCTOR} \twoheadrightarrow \text{STUDENT}$. By the complementation property this also implies that $\text{INSTRUCTOR} \twoheadrightarrow \text{CLASS DAY ROOM}$ as well and we can therefore decompose the initial relation into the relations R_1 and R_2 in Fig. 1-8(b). We now have the situation where all students are associated, through their instructor, with all classes that the instructor teaches - even those that the student does not take. As in our previous example with a lossy join, when we attempt to recover the original relation we will produce many invalid tuples and so obviously our original assumption of the MVD was in error.

DEPT#	PROJ#	DEPT#	EMP#	EMP_NAME	SKILL#	CHILD_NAME
Car	A	Car	111	Brown	Welder	Sue
Car	B	Car	111	Brown	Machinist	Sue
Paint	C	Car	111	Brown	Welder	Mary
		Car	111	Brown	Machinist	Mary
		Paint	222	White	Painter	Alice
		Paint	333	Black	Painter	David

R11

R12

Relation R1 Decomposed into R11 and R12
Figure 1-5

DEPT#	PROJ#	DEPT#	EMP#	EMP_NAME	SKILL#	CHILD_NAME	HOURS
Car	A	Car	111	Brown	Welder	Sue	3.5
Car	B	Car	111	Brown	Machinist	Sue	3.5
Paint	C	Car	111	Brown	Welder	Mary	3.5
		Car	111	Brown	Machinist	Mary	3.5
		Car	111	Brown	Welder	Sue	1.0
		Car	111	Brown	Machinist	Sue	1.0
		Car	111	Brown	Welder	Mary	1.0
		Car	111	Brown	Machinist	Mary	1.0
		Paint	222	White	Painter	Alice	5.5
		Paint	333	Black	Painter	David	2.5

Use of an EMVD outside its context
Figure 1-6

EMP#	EMP_NAME	DEPT#	PROJ#	HOURS	SKILL#	CHILD_NAME
111	Brown	Car	A	3.5	Machinist	Mary
111	Brown	Car	A	3.5	Welder	Mary
* 111	Brown	Car	A	1.0	Machinist	Mary
* 111	Brown	Car	A	1.0	Welder	Mary
111	Brown	Car	A	3.5	Machinist	Sue
111	Brown	Car	A	3.5	Welder	Sue
* 111	Brown	Car	A	1.0	Machinist	Sue
* 111	Brown	Car	A	1.0	Welder	Sue
111	Brown	Car	B	1.0	Machinist	Mary
111	Brown	Car	B	1.0	Welder	Mary
* 111	Brown	Car	B	3.5	Machinist	Mary
* 111	Brown	Car	B	3.5	Welder	Mary
111	Brown	Car	B	1.0	Machinist	Sue
111	Brown	Car	B	1.0	Welder	Sue
* 111	Brown	Car	B	3.5	Machinist	Sue
* 111	Brown	Car	B	3.5	Welder	Sue
222	White	Paint	C	5.5	Painter	Alice
333	Black	Paint	C	2.5	Painter	David

A Lossy Join with
Erroneous Tuples marked *
Figure, 1-7

INSTRUCTOR	STUDENT	CLASS	DAY	ROOM
White	A	Math	Mon	ME200
White	B	Hist	Tue	ME300
Black	A	Hist	Tue	ME400
Black	B	Math	Mon	ME300

RELATION R

Sample Relation and ER Diagram
Figure 1-8(a)

INSTRUCTOR	STUDENT	INSTRUCTOR	CLASS	DAY	ROOM
White	A	White	Math	Mon	ME200
White	B	White	Hist	Tue	ME300
Black	A	Black	Hist	Tue	ME400
Black	B	Black	Math	Mon	ME300

RELATION R1

RELATION R2

Incorrect Use of an MVD
Figure 1-8(b)

The appearance of these erroneous tuples is termed a 'lossy' decomposition [AH079] [ULL80] in the sense that the original relational information is now lost amongst the original and erroneous tuples. We have seen that, by definition, valid MVD decompositions are lossless. A theorem proven in [AH079] gives a necessary and sufficient condition to ensure that these errors do not occur. It states that the join of the relations $R(X)$ and $R'(Y)$ will be lossless

if the intersecting attributes of R and R'

imply an FD or MVD for $X \rightarrow Y$ or $Y \rightarrow X$;

That is : $X \cap Y \rightarrow X \rightarrow Y$ or $X \cap Y \rightarrow Y \rightarrow X$ (for FD)

$X \cap Y \twoheadrightarrow X \rightarrow Y$ or $X \cap Y \twoheadrightarrow Y \rightarrow X$ (for MVD).

A third class of dependency, the n -join dependency [AH077] [RISS77], deals with the lossless join constraint and is defined for a relation that can be decomposed into n sub-relations, but no fewer, in order to return the original relation using lossless joins. An MVD may therefore be regarded as a two join dependency. Current techniques for schema design do not handle join dependencies and therefore they are a limiting factor for all decomposition methods. An example of a join dependency along with a discussion of its implications is deferred until the next section.

1.3.3 COMMON ASSUMPTIONS

The uniqueness assumption [BEER78] [BERN76] is an important, though controversial, concept in schema design. It states that attributes must be both syntactically and semantically distinct. This ensures that the operations defined for relations (the restrict, project and natural join) result in meaningful tuples. Furthermore a dependency is meant to imply a unique semantic meaning. For example the use of EMP_NUMBER $\rightarrow$ NAME_OF_CHILD to imply the children of the employee as well as the children that the employee will be responsible for on the company picnic is said to violate the uniqueness assumption. Another violation follows from the ambiguous definition of an attribute such as 'age' if we use it to indicate:

1. the age of a child; and
2. the age of a wine bottle; and
3. a period in history.

Queries which join on ambiguous attributes within the relation will obviously be in error. Similarly the classic example is given in [KENT81] for the case of the transitive dependencies:

1. Section $\rightarrow$ Project; and
2. Project $\rightarrow$ Manager; from which we can infer
3. Section $\rightarrow$ Manager

The manager implied or determined by 3 may not be the same type of manager implied by 2, and thus a query to "Select all

managers" may provide an ambiguous result including not only Project Managers but Section Managers as well. Diagrammatic representation such as the ER approach can be used to resolve such ambiguities.

Following directly from the uniqueness assumption is the concept of a 'universal relation' in which we assume that all information can be stored in one relation using all of the attributes. Schemes are therefore simple projections onto reduced sets of attributes from which we may recover the original using a sequence of lossless joins. A complete discussion of the implications of this may be found in [KENT81]; however, the following problems should be noted. Incomplete tuples, that is, those with null entries for any attribute, pose problems under this assumption. For example a foster child who has no employee as a parent but who is to be recorded within the relation is one case; another is the addition of a subsequent attribute such as BUDGET which has no explicit relationship to any of the other attributes. A further case would be the creation of two similar relations with different semantic meanings such as R1 and R2 having (DEPT , BUDGET) and meaning

R1 Projected Budget,

R2 Committed Budget.

In all cases the assumption that a series of universal tuples can record the information correctly is questionable.

Normalization, as we shall see, is a partial solution to this problem. In the case of the child entity, we may create a scheme where such a tuple could be entered even though it does not participate in the universal relation. Such a tuple is said to dangle [TSIC82]. The latter instances require a modular approach to the creation of universal relations with reduced scope. A hospital database, for example, will certainly involve the use of medical, financial, administrative and personnel data attributes with many interactions. To approach this in a modular fashion we would consider the attributes which concern each area separately and then study the interactions of the areas for the given schema design. The 'universal relation assumption' is not quite as universal as it implies! These factors lead one to conclude that even though a formal set of axioms can be developed to cover certain types of constraints there will always be situations where the theory will require considerable human interpretation.

1.3.4 REPRESENTATION AND REDUNDANCY

When defining a 'good' schema design it is common to describe its capability to 'represent' the original scheme while reducing any inherent 'redundancy'. Clearly these notions require explanation. First, following the summary in [BEER78] we can describe the representation of the schema in three ways, namely:

1. data representation - the relation(s) contain(s) only valid tuples;
2. dependency representation - the relation(s) maintain(s) all applicable dependencies; or
3. full representation - the relation supports both data and dependencies.

We say that a schema represents the data of the original relation if the natural equijoin of the constituent schemes yields the original relation. Hence we can use the lossless join principle and decomposition property to constantly maintain this relationship. On the other hand the maintenance of dependency representation is more difficult to achieve.

We shall define that dependency representation for FD's is maintained

$$\text{if and only if } T^+ = \left(\bigcup_{i=1}^n d_i \right)^+ .$$

That is to say that the closure of the union of all dependencies d_i in the constituent schemes must be the same as the original closure. Consider the following brief example found in [DATE80] using the scheme $R(\text{NAME}, \text{JOB}, \text{STATUS})$ with FD cover T :

1. $d_1 : \text{NAME} \rightarrow \text{JOB} ;$
2. $d_2 : \text{JOB} \rightarrow \text{STATUS} ;$ and
3. $d_3 : \text{NAME} \rightarrow \text{STATUS} .$

In this example decomposing with d_3 gives :

$R_1(\text{NAME}, \text{STATUS})$ and $R_2(\text{NAME}, \text{JOB})$

which does not support dependency d_2 of the original scheme. An

extra integrity check will be required to support the existence of this inter-relational constraint and so this is not considered a 'good' decomposition. If we use only those dependencies which result in schemas from which we can recover the original FD cover then we avoid the need to add these extra constraints. Using MVDs in a similar manner has been proposed in [ZANI80] where a 'good' decomposition having 'complete relatability' is one in which the FD and MVD covers of the original relation R may be derived from the FD and MVD covers of the sub-schemes R_1 and R_2 . Clearly the last form of representation maintaining both data and dependencies is the preferred although as we shall see that this is not always possible when seeking a schema without redundancy.

We say that a relation R_i is redundant:

if and only if the join of all relations in the schema without R_i yields the universal relation or

$$R = \bigstar_{\substack{j=1 \\ j \neq i}}^n (R_j).$$

Similarly, an FD cover T is redundant:

if and only if the schema closure T^+ may be derived from the union of all of the scheme covers, less cover T_i or ;

$$T^+ = \left(\bigcup_{\substack{j=1, \\ j \neq i}}^n T_j \right).$$

EXAMPLE 1.4 An example of a redundant decomposition is shown in figure 1-9. R21 is redundant because R can be recreated using a lossless join of R22 and R1 on b. A non-redundant decomposition could have been achieved instead by using d3 and d4 to immediately get R1 and R22.

Consider the scheme R (a b c d e f)

with the dependencies

d1 : ab → f

d2 : e → bcd

d3 : b → af

d4 : b → cde.

Decomposing with d1 we are left with

R1 (a b f) and

R2 (a b c d e)

Using d2 to decompose R2 we have

R1 (a b f)

* R21 (a e)

R22 (b c d e)

An Example of a Redundant Decomposition
Fig. 1-9

1.4 NORMALIZATION

1.4.1 INTRODUCTION

In this section we discuss the potential anomalies and redundancies that result from the use of certain schema designs and review the process of normalization.

1.4.2 ANOMALIES

Three basic anomalies can be described:

1. insertion;
2. deletion; and
3. modification or update.

An insertion anomaly occurs when we attempt to add valid information into the relation R, we find we cannot create the necessary tuple or by adding a valid tuple we introduce an inconsistency into the relation. For example, we have seen in Fig. 1-3 that the entry of a child with no employee parent is inconsistent with the relation R because we cannot introduce nulls within the key attributes. Deletion of a tuple can also create an anomaly in the semantics of the relation. For example, in Fig. 1-3, if tuple #2 were deleted then the MVD

EMP# : EMP_NAME DEPT# ; SKILL# ; CHILD_NAME ; PROJ# HOURS
would not hold because Brown as a machinist and Brown as a Welder have different sets of children. Similarly if we introduce a new attribute, SUPERVISOR_NO which represents the employee number of the supervisor and then we delete the Supervisor's employee record, we have the logical anomaly that personnel are working for a non-existent employee. Modifications to any existing tuple can create similar problems as the above. We note also that there is a considerable amount of duplicate information within Fig. 1-3 such as the repetition of EMP_NAME, SKILL#, PROJ#.

HOURS and CHILD_NAME for all valid combinations for each EMP#.

1.4.3 NORMAL FORMS

To reduce these anomalies and redundancies a series of normal forms based on the various dependency types have been developed. Due to the great number of dependency types there are a considerable number of normal forms to be found in the literature. Of the 10 listed in [AROR80] we consider only the following normal forms and group them into four classes based primarily on the dependency used in its definition. These are:

1. Reduction of repeating groups to atomic values
-First Normal Form (1NF)
2. Functional Dependencies
-Second Normal Form (2NF)
-Third Normal Form (3NF)
-Boyce-Codd Normal Form (BCNF)
3. Multivalued Dependencies
-Fourth Normal Form (4NF); and
4. Join Dependencies
-Fifth Normal Form (5NF)
(also called Projection Join Normal Form).

A special pseudo-normal form called Domain Key Normal Form (DKNF)-[FAG81] can also be mentioned as the ultimate semantic normal form. Higher normal forms presuppose that the relation is in the preceding normal form so we may write :

$$1NF \subseteq 2NF \subseteq 3NF \subseteq BCNF \subseteq 4NF \subseteq 5NF \subseteq DKNF$$

The three normal forms of primary interest are 3NF, BCNF and 4NF.

Third normal form is attributed to Codd. It resolves the interactions of functional dependencies to ensure that all non-prime attributes are fully functionally dependent upon the primary key of the relation. That is, there can be no dependencies between non-prime attributes. 3NF is the minimum desired level of normalization.

An improvement on 3NF attributed to Boyce and Codd handles the possibility of several candidate keys, not necessarily disjoint, within a given relation. A determinant is a set of attributes which fully functionally determine other attributes (that is, a determinant is the left side X of any given dependency $X \rightarrow Y$ in T^+). We may now define BCNF as [DATE80]:

A relation is in BCNF if and only if it is
in 3NF and every determinant is a candidate key.

It is obvious that the intent of normalization is a clear separation of dependencies. The introduction of MVDs in [FAG77b] led to the definition of a stronger normal form which improves upon the 3NF and BCNF. Fourth normal form (4NF) is defined strictly in terms of MVDs, although due to FD-MVD1 we have noted that all FDs are MVDs. We thus define 4NF as:

A relation $R(X,Y)$ is in 4NF if and only if,
for all MVDs $X \twoheadrightarrow Y$ valid in relation R , $X \rightarrow Y$.

Decomposition to 4NF is more desirable than 3NF or BCNF

because effort and space is required to maintain all of the possible combinations of attributes due to the presence of the MVDs. It is also conceptually straightforward because it separates the repeating groups and their determinants into distinct tables.

As previously discussed, in some cases a relation must be decomposed into more than two relations to maintain a lossless join.

EXAMPLE 1.5 Fig. 1-10 illustrates both the dependency structure and the ER diagram for a subset of Fig. 1-3 for such a case. Only a decomposition into the 3 schemes

$R1(EMP\#, DEPT\#)$ $R2(EMP\#, PROJ\#)$ $R3(PROJ\#, DEPT\#)$

will retain the original dependency representation.

A fifth normal form (also called Projection-Join normal form) has been based on this behaviour. As defined in [DATE80] [AH079] we say that a relation is in 5NF if and only if all join dependencies are implied by keys. 5NF has been cited as the ultimate normal form for the basic relational operations of project, join and restrict [DATE80]. Join dependencies are therefore a significant constraint on all binary decomposition algorithms.

with dependencies

d1: EMP# $\rightarrow$ DEPT#
 d2: EMP# $\rightarrow$ PROJ#
 d3: PROJ# $\rightarrow$ DEPT#
 d4: PROJ# $\rightarrow$ EMP#

For example, decomposing

using d1 or d4

yields

R1 (EMP#, DEPT#)	R1 (PROJ#, EMP#)
R2 (EMP#, PROJ#)	R2 (PROJ#, DEPT#)

which do not support either

PROJ # $\rightarrow$ DEPT # or EMP# $\rightarrow$ DEPT#

An Example of a 3 Join Dependency

Figure 1-10

It can be shown, however, that transforming a scheme to these various normal forms reduces but does not completely eliminate the possibility of insertion, deletion or update anomalies. A higher Domain Key Normal Form has been proposed to completely eliminate these anomalies by enforcing all constraints implied by relational keys and attribute domains. Fagin [FAG81] has proven it to contain BCNF, 4NF and 5NF. This normal form is significant because it concludes that no static schema can ever be completely protected from a transaction anomaly unless a runtime check of key and domain constraints is conducted. Seen in perspective, then, it is obvious that a static schema design does not provide total protection in a dynamic situation.

1.5 THE DESIGN PROCESS

In this section we discuss the database design process and the role of the schema design aid. In essence, a structured approach to the logical design of a database will consist of at least the following steps:

1. collection of data and requirements;
2. dependency evaluation or analysis; and
3. design and refinement of the conceptual schema.

The identification and collection of all pertinent data elements comprising a large information system is not a trivial

task and may require several man months of work. The analyst must examine the entire information system to determine the attributes and domains of the data, the approximate volume of data, methods of input and specific output requirements such as reports or transactions. This information is then used to prepare the data dictionary and supporting conceptual diagrams.

The designer must now deduce the inherent data dependencies. Use of data dependencies to design the schema is not the only technique but it offers a basis on which the design alternatives can be assessed in a formal manner. Approaching the system in a modular fashion to ensure that the 'uniqueness' and 'universal relation' assumptions are maintained, the analyst then extracts all of the valid dependencies and their proper contexts. There is no current methodology that guarantees the correctness of this procedure and so the product may be incomplete, ambiguous or in error. Once we have determined the dependency structure of the database we are prepared to use the schema design aid.

There are two common approaches to the design phase, namely, synthesis and decomposition. The synthesis approach is taken by [BERN76] and follows the original work by Codd. We approach the normalization process on a relation by relation basis. We group attributes which have the same determinants and then remove any partial or transitive dependencies leaving relations in 3 NF. Bernstein [BERN76] has published a synthesis algorithm that

yields the minimal 3NF schema which completely represents both data and dependencies. It does not consider the existence of MVDs or EMVDs which is a significant limitation. Decomposition on the other hand assumes a universal relation and recursively decomposes it using multivalued dependencies producing 4NF relations. Current decomposition algorithms consider both FD's and MVD's but at the cost of not guaranteeing either data or dependency representation or minimal redundancy.

We note therefore that synthesis is FD dependent and is bound by the initial selection of relations in the sense that it cannot recombine and separate the relations in order to optimize the final design. Decomposition on the other hand is MVD dependent and can choose any combination chosen from the universal relation.

To compare alternative designs we examine the storage costs of the given relations and the CPU effort required to answer the queries or transactions put to the database. Normalization is advantageous because it reduces the possibility of anomalies and data redundancy but this incurs additional query processing costs to project, join and restrict. The determination of the optimal normal form for a given database is therefore difficult to assess. We will consider only a part of this problem involving the storage costs and the CPU effort to satisfy a specified 'query profile'. Each query within the profile is a set of

attributes that are commonly requested together as a global report from the database. For example, the query "Employee_Name, Employee_Address, Job_Title" will produce a list of all employees with their address and job title. We then consider the cost of the query in terms of the number of secondary storage accesses needed to gather the information from the schema. We show that this information can be of significant benefit to the decomposition process in selecting the best dependency to use at each step.

1.6 CURRENT RESEARCH

The study of database design methods is sub-divided into logical design (conceptual schema) and physical design (storage and access). Logical design is further sub-divided into the areas of data modelling and schema design.

Descriptive infological data models are widely used within systems analysis methodologies because they present the information in straightforward manner which can be discussed with the ultimate user who may not be familiar with a schema-like representation. Automated support tools exist to support these models including the IBM Inferential Aspect Model [BUBE77] [IBM75] and the Problem Statement Analyzer (PSA) [TEIC75]. These software products enable the analyst to describe a system in terms of its entities and relationships using a special high

order language syntax. The tool parses and verifies the entries and generates a series of tabular reports to examine various interactions of the attributes. A schema is not one of the final reports however. Implementation of the database is also not supported.

Trial implementations of infological models are currently conducted by researchers in the area of artificial intelligence with the semantic data model (SDM) developed at MIT [HAMM76] being perhaps the most advanced. Semantic models use 'object' or 'actor' programming environments to support complex run-time activities. For example, in their ship and dockyard database, when the coordinates of a tropical storm are entered the system automatically generates rerouting information for all ships at sea, reschedules port facilities because of expected ship transit delays and reorganizes the loading of ships in port to speed the shipment of perishable goods. The schema is generated internally as the designer describes the relationships through a LISP-like syntax.

Initial research into schema design began with [ARMS74] and [DELO73] who applied the theory of boolean switching functions and predicate logic to the concepts put forward by Codd. The derivation of the various normal forms is well documented in [CODD70] [FAG77b] and [AHO79]. Axiomatizations for FD and MVD's appear in [ARMS74] and [BEER77]. Research into the computational

aspects of dependency theory and its application has also been conducted. Algorithms to determine if a given dependency is a member of the closure of a group of dependencies and the determination of minimal covers may be found in [BEER79] [BERN76]. The best summaries of schema design and normalization may be found in [DATE80] [ULL80] [BEER78] and [TSIC82].

A successful algorithm which synthesizes minimal 3NF relations from a set of FD's has been published by [BERN76]. Subsequent work in [BEER79] reveals that extending the algorithm to produce BCNF relations makes the problem NP-hard which is to say that an algorithm with polynomial time complexity is unlikely to be found. The original decomposition approach and a basic algorithm may be found in [FAG77a] and [FAG77b]. Two subsequent algorithms have been published in [ZANI81] and [LIEN81] and each play a major role in the development of the schema design aid.

CHAPTER 2

PRINCIPAL ALGORITHMS IN SCHEMA DESIGN

2.1 INTRODUCTION.

This chapter describes the various algorithms which are the basic tools of the design aid. The next chapter then describes their utilization in actual schema design algorithms. In the first section we discuss the membership algorithms for both FD and MVD's and include an original improved algorithm for the manipulation of the dependency basis. In the next section we discuss the algorithm used to estimate an upper bound for the 1NF size of given relation based on its valid dependencies and estimated domain sizes. Finally, in the last section we describe an original algorithm which estimates the cost in secondary storage accesses to satisfy a given query from any schema design. This completes the description of all basic algorithms which constitute the design aid.

2.2 THE MEMBERSHIP PROBLEM

1

The membership problem for both FD and MVD's is fundamental to all schema design algorithms and the overall performance of any design aid is directly related to the efficiency of its solution to this problem. Essentially the problem may be stated as follows:

Given a dependency d_i and a cover F or G , determine if d_i is a member of the closure of the dependency cover that is,

is $d_i \in F^+$ or $d_i \in G^*$?

A brute force approach would involve the computation of the complete closure which, for all but the simplest of covers, would require exponential time and memory. Since the membership problem is useful in many circumstances, this approach is clearly impractical.

Before discussing the proposed algorithms we present several uses for the membership problem. For example, in many situations we need to ensure that the closure of one cover is equivalent to a given closure. Testing equality using both complete closures is even more impractical than the computation of either.

Instead, by ensuring that each member of cover A say, is a member of the closure of B and vice versa, this comparison may be done in time directly proportional to the sizes of the covers and the time for each test. Reducing a cover to a minimal cover is also accomplished more efficiently using the membership algorithm. Each dependency is tested, one at a time, to determine if it is a member of the closure determined by the remaining dependencies. It is removed if it is found to be redundant. In this way when all have been tested the dependencies that remain form a minimal cover. For example, given $R(ABC)$ with:

```

d1 : A -> B
d2 : B -> C
d3 : A -> C

```

d1 is not a member of the closure defined by d2 and d3 and similarly for d2. However d3 is supported transitively by d1 and d2 and may be removed leaving d1 and d2 as the minimal cover. We can also find the keys and superkeys for a given relation by testing all combinations of attributes X, in the scheme $S(a,b,c,d,...z)$, to see if :

$$X \rightarrow S \in F^+ .$$

Lastly, a key is a superkey (that is, it is not a full functional dependency) if any of the attributes on the left side can be removed without affecting the right side of the FD. In this case we remove each attribute individually and use the membership algorithm in the same way as the minimal cover technique.

Two approaches, very similar in concept, are used to

effectively compute the membership decision directly without computing the complete closure. For FD's the approach is directly analogous to the derivation of productions used in formal language analysis. We seek to derive the goal symbol (in this case the right side of the dependency) from the given symbol (the left side) using a set of defined rules (the dependency cover and the properties of functional dependencies). The approach for MVD's requires an additional data structure called the 'dependency basis' which partitions the scheme. We show that a series of operations on the basis using the dependency cover will allow us to determine whether the given MVD is also a member of the MVD closure.

2.2.1 FUNCTIONAL DEPENDENCY (FD) MEMBERSHIP ALGORITHM

An efficient algorithm of time complexity $O(n)$ where n is the number of dependencies has been published in [BEER79] and was implemented in the design aid. The following discussion is therefore quite brief and the reader is urged to seek the details in this reference.

Given the complete set of inference rules for FD's we can develop logical derivations for all valid dependencies in the closure using FD2 and FD3 (augmentation and transitivity). We therefore consider the membership problem to be whether a derivation exists which proves the FD. Taking advantage of the

similarity between algebraic derivations and the productions of formal languages, Bernstein proved that derivation trees could be used to determine all unambiguous derivations. [BEER79] then develops an efficient algorithm using a directed graph approach to construct the derivation tree for the right side of the dependency $X \rightarrow Y$. The membership problem is resolved by seeing if the left side, X , is contained in the tree. If it is fully contained then the dependency is a member of the closure. Fig. 2-1 demonstrates the derivation tree approach.

Consider the scheme $R(a,b,c,d,e,f)$
 with F^+ :
 $d1: ab \rightarrow cd$;
 $d2: c \rightarrow e$;
 $d3: de \rightarrow f$; and
 $d4: ae \rightarrow d$;

Determine if $ab \rightarrow f \in F^+$

Since a and b are both in the tree, $ab \rightarrow f$ is valid.

Derivation Tree Analysis
 Figure 2-1

The details of the algorithm by [BEER79] are shown in Fig. 2-2 and may be summarized as follows: given a set F of n FD's in

the form

Left Side (LS[n]) $\rightarrow$ Right Side (RS[n])

defined for the universal relation $(a_1 \dots a_m)$

and the test FD $X \rightarrow Y$, determine if $X \rightarrow Y \in F^+$

using the following data structure:

DATA STRUCTURE

- (1) attributes are represented by integers between 1 and m.
- (2) FD's in F are represented by integers between 1 and n.
- (3) LS[i] and RS[i] are arrays of sets containing the attributes in the left and right sides of the i'th FD in F.
- (4) FOUND_DEPENDENT is a set of attributes already found to be functionally dependent on X.
- (5) TO_BE_CHECKED is a subset of $\{a_1 \dots a_m\}$ which contains attributes which have not been considered as yet.
- (6) ATTR_LIST[i] is an array of lists indicating which dependencies have attribute i in their left sides.
- (7) COUNTER[i] is an array containing the number of attributes in the left side of dependency i which have not yet been considered.

The complete algorithm is shown in Fig. 2-2.

PROGRAM FD MEMBERSHIP

begin

{ STEP 1 - initialize the data structure }

set all ATTR_LIST entries to nil;

for each dependency i do

begin

COUNTER[i] <- 0;

for each attribute j in the left side of dependency i do

begin

ATTR_LIST[j] <- ATTR_LIST[j] + {i};

COUNTER[i] <- COUNTER[i] + 1 ;

end;

end;

FOUND_DEPENDENT <- X ;

TO_BE_CHECKED <- FOUND_DEPENDENT ;

{ STEP 2 } while TO_BE_CHECKED $\neq \emptyset$ do

begin

Pick NEXT_TO_CHECK from TO_BE_CHECKED

TO_BE_CHECKED <- TO_BE_CHECKED - NEXT_TO_CHECK

{ STEP 3 } for each i in ATTR_LIST[NEXT_TO_CHECK] do

begin

COUNTER[i] <- COUNTER[i] - 1;

if COUNTER[i] = 0 then

for each attribute j in RS[i]

add {j} to FOUND_DEPENDENT

TO_BE_CHECKED <- FOUND_DEPENDENT

end

end;

{ STEP 4 } if Y $\in$ FOUND_DEPENDENT then

output 'DEPENDENCY IS A MEMBER OF F '

else output 'DEPENDENCY IS NOT A MEMBER OF F '

end.

A Linear Time Membership Algorithm
 For Functional Dependencies
 Figure 2-2

2.2.2 MULTIVALUED DEPENDENCY (MVD) MEMBERSHIP ALGORITHM

Consider the collection of all subsets Y of R which are multidependent upon an arbitrary set of attributes X in the multivalued closure G^* : that is -

$$X \twoheadrightarrow Y \in G^* .$$

MVD5 and MVD6 guarantee that the collection is closed under the operations of union, difference and intersection. Furthermore, by MVD0 and MVD1 we have:

1. $X \twoheadrightarrow X$; and
2. $X \twoheadrightarrow R-X$

so the collection of subsets therefore partitions the universal set R .

Let us now define that an MVD $X \twoheadrightarrow Y$ holds in G^* if and only if Y is the simple union of one or more disjoint sets in the basis formed by these subsets. We call this collection the dependency basis of X relative to G and write this as $DEP(X)$ [BEER 80].

To show that a given MVD $X \twoheadrightarrow Y$ holds in G^* it therefore suffices to show that Y is a union of the elements in $DEP(X)$.

EXAMPLE 2.1 Consider the relation R defined over the attributes $R(a,b,c,d,e,f,g)$ with the dependency basis:

$$DEP(abc) = [\{a\} \{b\} \{c\} \{de\} \{fg\}].$$

We can therefore prove that :

1. $abc \twoheadrightarrow defg$; or
2. $abc \twoheadrightarrow de$; but not that
3. $abc \twoheadrightarrow f$ because the set containing f holds g as well.

As shown in Examples 1.2 and 1.3, attributes which are logically 'clustered' cannot be separated without inducing 'lossy' join errors.

Finding the dependency basis of a set of attributes X is an iterative process which uses the given MVD's and successively modifies the collection of subsets until no further modification is possible. Following [BEER 80] this may be described using the algorithm given in Fig. 2-3.

PROCEDURE Dependency Basis 1

given the set of MVD's $g_1 \dots g_n$
 as : $g_i : LSi \twoheadrightarrow RSi$
 valid in the relation $R(a_1, a_2, a_3, \dots, a_m)$
 and a set of attributes $X \subseteq R$
 FIND DEP (X).

```

begin
Step 1  Initialize
        Basis  $\leftarrow [\{a\} | a \in X \text{ and } \{R-X\}]$ 
          { that is the singleton sets for }
          { all attributes  $a \in X$  and the }
          { set  $\{R-X\}$  }
        Basis_Changed  $\leftarrow$  True ; { Simply a boolean flag }
        While Basis_Changed do
          begin
Step 2   Basis_Changed  $\leftarrow$  False
          for each dependency  $g_i$  do
            begin
              Y  $\leftarrow$  union of all sets which contain
                attributes in  $RS_i$ , that is ;
                 $U \{R | R \cap RS_i \neq \emptyset, R \in \text{Basis}\}$ 
              Z'  $\leftarrow$   $LS_i - Y$ 
              IF (Z' is not the null set) and
                (Z' is not a union of Basis elements)
              then
                begin
                  modify Basis so that a simple union
                    of Basis elements will cover Z'
                  Basis_Changed  $\leftarrow$  True
                end
            end
          end
        end
        return Basis
end.
```

Basic Dependency Basis Algorithm
 Figure 2-3

Step 1 follows from MVD5 and MVD6. [BEER 80] shows by induction that after each pass through step 2, $X \twoheadrightarrow Y$ holds for all Y formed as the simple union of basis elements. Consider the case when an arbitrary dependency $W \twoheadrightarrow Z$ is used that changes the basis. Let the union of all sets which cover W be Y . Since the basis partitions R , W is contained in Y (Y may contain other attributes as well). By MVD2 we can augment $W \twoheadrightarrow Z$ to be $Y \twoheadrightarrow Z$. By our original assumption $X \twoheadrightarrow Y$ is in G^* and by transitivity so is $X \twoheadrightarrow Z - Y$ (for Y and Z not necessarily disjoint). Thus the action of separating $Z' = Z - Y$ does not affect the original assumption that $X \twoheadrightarrow Y$ still holds for all sets in the basis.

Conceptually the effect of the algorithm is as follows: for each dependency $X \twoheadrightarrow Y$ which we know to be valid in the closure, we seek to ensure that Y is the simple union of some sets W_i in the basis. If this is not currently the case and Y is contained within the union of sets Z ($Z = \cup W_i$) then we must change the basis so that this is made to hold. Since $X \twoheadrightarrow Y$ was given and $X \twoheadrightarrow Z$ holds by our original hypothesis, then using the decomposition rule, we can infer that

$$X \twoheadrightarrow Y \cap Z \quad \text{and} \quad X \twoheadrightarrow Z - Y.$$

Using reflexivity we may then rewrite this each $W_i \in Z$ as

$$X \twoheadrightarrow Y \cap W_i \quad \text{and} \quad X \twoheadrightarrow W_i - Y.$$

Replacing each W_i by $Y \cap W_i$ and $W_i - Y$ now ensures that our original condition is still satisfied.

The computationally complex parts of the algorithm are therefore the steps necessary to determine the union of the sets which cover Z' , to modify the basis when separating Z' and to determine if a given dependency's right side attributes are a simple union of basis elements.

We propose a new method for these operations which uses a binary tree and which significantly improves the time complexity of the algorithm. Consider the basis in the previous example given in binary tree form such as shown in Fig. 2-4.

Dependency Basis for Example 2.1 in Binary Tree Form
Figure 2-4

We define two operations on this tree, namely :

1. seek (X) - returns all attributes in the leaf nodes containing any part of X and hence covering X ; and
2. separate (X) - splits a node containing the attribute set Y into $X \cap Y$ and $Y - X$ and resets the Basis_Changed flag to TRUE if new nodes are created.

Thus for the example in Fig. 2-4 :

1. seek(d) returns de ; and
2. separate(af) splits f from g, finds g already separate and modifies the tree to the form found as Fig. 2-5.

The Modified Dependency Basis of Example 2.1
Figure 2-5

Fig. 2-6 describes the improved algorithm and the following example demonstrates its use.

PROCEDURE Dependency Basis 2

```

begin
Step 1.  build tree with R
         for all ai ∈ X do
           split(ai)
         Basis_Changed ← True
         While Basis_Changed do
           begin
Step 2    Basis_Changed ← False
           for each dependency gi do
             begin
               Z' ← LSi - Seek(RSi)
               Separate(Z')
             end
           end
         write Basis (the leaves of the tree)
       end

```

Improved Basis Algorithm
Figure 2-6

EXAMPLE 2.2 Consider the set of dependencies G defined over the relation R with :

R(a,b,c,d,e,f,g,h,i,j) and

```

g1 : fg->->ehi
g2 : bce->->ij      FIND DEP (bfgh)
g3 : ef->->abh
g4 : i->->cdej

```

We first initialize the Basis tree according to step 1 as shown in Fig. 2-7. We then reset Basis_Changed to False and proceed with g1. Seek(fg) returns fg and so Z'=ehi-fg. Separate(ehi) modifies the tree as shown in Fig. 2-8 and resets Basis_Changed to True. For g2 we seek(bce) which

returns abcdeij. $Z' = ij - abcdeij = \emptyset$ and so no change is made. For g_3 we seek(ef) which returns efi and so $Z' = abh - efi = abh$. Modifying the tree again results in Fig. 2-9. Dependency g_4 does not change the Basis and we loop back to the start of the While loop. We now find that g_1 to g_4 make no further changes to the basis and the algorithm terminates. The final dependency basis is therefore :

$$DEP(bfgh) = [\{a\}\{b\}\{f\}\{g\}\{h\}\{cdj\}\{ei\}].$$

If we wish to know if a given dependency, say $bfgh \rightarrow Y$, is valid by using this technique then we simply execute a seek(Y). If it returns more than Y then the right side is not a simple union of basis elements and therefore the dependency is not supported by this cover.

Example 2.2 - After Initialization
Figure 2-7

Example 2.2 - After g1 modifies Basis
Figure 2-8

Example 2.2 - After g3 modifies Basis
Figure 2-9

Further improvements can be made to the algorithm as well by using secondary data structures. We use an array of booleans (a bit vector) to indicate those attributes which are singular (that is a singleton set of the attribute already exists in the basis). Singular attributes can be removed before executing a seek because they do not participate in any further basis modifications. Also, when all attributes in the right side of a dependency in G become singular, the dependency need not be considered any further. Considering these bit vectors to be of length $|R|$, removal of the singular attributes can be performed in fixed time by using the singular array as a bit mask.

We now examine the time complexity of the algorithm following [BEER80] closely for comparison purposes. Consider G containing n dependencies defined over R where $|R| = m$ attributes. Initialization therefore requires $O(m)$ steps. For all practical purposes, for values of m which are simple multiples of the standard computer word length, set operations on the bit vectors including intersection, union and difference can be performed in fixed time. Step 2 can be performed at most m times. Use of the binary tree results in at most $O(m)$ operations to return the covering sets with $O(\log_2 m)$ steps required if only 1 leaf covers the set. Beer's algorithm uses an $n \times m$ matrix and requires $O(m^2)$ steps for each comparison. Similarly, basis modification using the improved algorithm requires $O(\log_2 m)$ operations because we perform a single binary search of $O(\log_2 m)$.

and then create and fill 2 new leaves in fixed time. Since the loop can be executed n times for each change this step requires $O(nm)$ operations (although optimally we are $O(n \log_2 m)$). Thus the total of all loops through step 2 requires $O(nm \times m)$ operations which is $O(nm^2)$. Comparing this with Beer's algorithm which is $O(nm^3)$ we can see the marked improvement in time complexity.

2.3 A 1NF SIZE ESTIMATOR FOR RELATIONAL SCHEMES

In order to assess the cost to answer a query using projection and joins from a given schema, estimates must be made of the size, in numbers of tuples and bytes of storage, for each constituent scheme. A method for establishing a consistent upper bound based on the existence of valid dependencies within each scheme is now described. We state the following assumptions:

1. the multiplicity value entered for each multivalued dependency is the expected value for all determinant attribute values, that is for : EMP $\twoheadrightarrow$ SKILLS, 3 each employee possesses an average of 3 skills;
2. the full range of the root domain is used, that is there are as many entities as the domain allows.

The algorithm proceeds similarly to Warshall's algorithm for the transitive closure of a directed graph. For each possible root A of the given scheme $R(a_1 \dots a_i \dots a_n)$ we begin by creating a selected set of attributes which will initially

contain the root itself. We then seek all unused dependencies (valid in R) which have their determinant attributes contained in the selected set. Successively as each such dependency is found we add the attributes which are elements of its right side to the selected set and adjust the overall multiplicity. We can then use MVDO (complementation) to show

for all $X, Y, Z \in R$
 if $X \twoheadrightarrow Y$ with multiplicity m_1 holds in R and
 $X \twoheadrightarrow Z - Y$ with multiplicity m_2 holds in R
 where $R - Y = Z$

then

$$|XY*XZ| = |X| \times m_1 \times m_2$$

EXAMPLE 2.3 For example consider the EMP_NO, CHILD_NAME, SKILLS relation where:

1. EMP_NO $\twoheadrightarrow$ CHILD_NAME, 3 ;
2. EMP_NO $\twoheadrightarrow$ SKILLS, 5.

We will assume that 50 employees exist, each with an average of 3 children and 5 skills. The expected value for the number of tuples required to represent this relation in 1NF is therefore $50 \times 3 \times 5$ or 750 tuples.

If the next unused dependency with its left side contained in the selected set has its right side attributes already contained as well then we consider that it has already been dealt with and make no further change to the multiplicity factor. This

deals with:

1. bijections $X \rightarrow Y$ and $Y \rightarrow X$; or
2. cycles such as $A \rightarrow B$, $A \rightarrow C$, $CA \rightarrow B$.

If we have selected accurate estimates for the expected multiplicity factors and domain sizes then the overall size will be the same regardless of the order in which we select the dependencies. On the other hand if we have many dangling tuples where the domain size of one attribute bears no relation to its actual association with the other attributes then large errors will develop.

EXAMPLE 2.4 Considering part of the school relation with:

1. INSTRUCTOR $\rightarrow$ CLASS, 4; instructors teach 4 classes
2. CLASS $\rightarrow$ STUDENT, 30 ; each class has 30 students
3. CLASS $\rightarrow$ INSTRUCTOR ; a class has only 1 instructor
4. STUDENT $\rightarrow$ CLASS, 6 ; each student takes 6 classes.

Using CLASS as the root with size 10 we will have an estimated upper bound of $10 \times 30 = 300$ records. Using INSTRUCTOR with domain size 10 we will have $10 \times 4 \times 30 = 1200$ records. Finally, using STUDENTS with size 500 we will have

500x6=3000 records. Each of these indicate situations where there are dangling tuples which do not participate in the universal relation. The estimates chosen must therefore be more uniform in order to arrive at a consistent value. In this case we must assume 10 classes, 25 instructors and 500 students to maintain approximately 3000 records.

All individual attributes and the left hand sides of valid dependencies are tested as possible roots. The maximum overall size is returned. Compound roots, that is those with more than 1 attribute, require interaction with the user in order to determine their estimated domain size. A list of previously used compound roots and their assigned domain values is therefore kept by the design aid for future use to reduce these interactions.

When the final estimate is made all unused attributes are reported to the user. This signifies that the attribute cannot be derived from the root and may indicate that either :

1. the attribute is not logically related to the others and is in violation of the universal relation assumption;
or
2. an important relationship has been omitted.

When such notification is raised the size estimator is obviously questionable.

INPUT : The set of attributes $R = \{ a_1, a_2, a_3 \dots a_n \}$
 The set of dependencies $D = \{ d_1, d_2, d_3 \dots d_k \}$
 which are valid in R

OUTPUT: The maximum estimate of relation size in records ($\#R$) and
 bytes ($\#B$) as well as a listing of all unused attributes.

METHOD:

procedure FIND_SIZE (R, D)

```

  procedure FIND_SIZE_FOR_GIVEN_ROOT (R, D, ROOT)
  begin
    accumulated multiplicity    <- 0
    more_added                  <- true
    #records, #bytes             <- 0
    all dependencies D           <- not used
    all attributes in R          <- not used
    selected set                 <- ROOT
    while more_added do
      begin
        more_added <- false
        for all valid dependencies di if not already
          covered by selected set [ie for  $X \rightarrow Y$  with
             $XY \in$  selected set] do
          begin
            more added <- true
            selected set <- selected set + di.RS
            accumulated multiplicity <- accumulated
              multiplicity * di.MULT
            all attributes  $i \in$  di.RS <- used
          end
        end
        Size <- 0
        for all used attributes ai
          Size <- Size + ai.Size
        return accumulated multiplicity, Size and attributes
          not used
      end
    end
  begin
    #R, #B <- 0
    for all possible roots ri do
      #R, #B <- max [#R, #B or
        FIND_SIZE_FOR_GIVEN_ROOT (R, D, ri)]
    report unused attributes
    return #R, #B
  end.

```

Algorithm to determine the estimated size of a given relation R
 Figure 2-10

The algorithm is described in Fig. 2-10. The performance of the algorithm follows primarily from the number of loops performed within the local procedure. If we assume that the next dependency which can add new attributes to the selected set is located halfway through the remaining dependencies then the total number of searches is the sum:

$$R \sum_{i=0}^D \left\lceil \frac{D-i}{2} \right\rceil = \left\lceil \frac{D}{2} \right\rceil + \left\lceil \frac{D-1}{2} \right\rceil + \left\lceil \frac{D-2}{2} \right\rceil + \dots + 2 + 2 + 1 + 1$$

where D is the number of dependencies
and R is the number of roots to be tested

Considering D even, the quantity in brackets is twice the arithmetic series from 1 to D/2

$$\text{or: } 2 \left(1 + 2 + \dots + \frac{D}{2} \right) = 2 \left(\frac{D}{2} \left(\frac{D}{2} + 1 \right) \right)$$

The overall complexity is therefore:

$$O(|R| \frac{D^2}{2} (D + D)) \text{ which is } O(|R| D^2)$$

EXAMPLE 2.5 Fig. 2-11 demonstrates the algorithm with a sample university database. Using CLASS as a root, we can only add TEXT to the selected set, and terminate without using all attributes. None of the other single attributes ultimately use all attributes either so we then consider compound roots. For the compound root CLASS_SECT the user is asked to estimate the domain size. For 10 classes each with an average of 3 sections we enter the value 30. We now examine each dependency and adjust the multiplicity value. We select in the order D1 to D6 giving a final multiplicity

value of $30 \times 5 \times 2 \times 2 \times 3 \times 2 = 3600$. Given 30 class section combinations as our root, we estimate that we need $3600 \times 30 = 108,000$ records for the resulting table which will require $108,000 \times 126 \text{ bytes} = 13.6 \text{ M bytes}$ of storage.

Consider

R(CLASS SECT STUD MAJ EX YEAR INST RANK SAL TEXT ROOM DAY)

with attributes of type (C - character, N - number)
and size in bytes

C5 C1 C25 C10 N2 N5 C25 C10 N5 C25 C10 C3

For a total of 126 bytes per tuple

with the following dependencies:

D1:CLASS,SECT-->STUD,MAJ,YR,	30	Each section of each class has 30 students, each with 1 MAJ and 1 YR
D2:CLASS,SECT,STUD-->EXAM,	5	Each student has 5 exam results on file
D3:CLASS-->TEXT,	2	Each class uses 2 texts
D4:CLASS,SECT-->INST,RANK,SAL,	2	Each section has 2 inst.
D5:CLASS,SECT-->DAY,	3	Each section meets on 3 days
D6:CLASS,SECT,DAY-->ROOM,	2	Each section meets each day in 2 rooms

Example 2.5 - 1NF size of a given relation
Figure 2-11

2.4 QUERY COST ALGORITHM

We begin the discussion of the query cost algorithm by describing a typical physical schema implementation and state several assumptions used in the design. Two possible approaches to the query processor are then compared and the details of the final algorithm are presented.

2.4.1 IMPLEMENTATION DETAILS

We assume a single processor based relational database similar to IBM's System R described in [SCHK79] [BLAS77] and [CHAM81], physically organized in the following manner. A tuple is stored as a series of sequential fields in secondary storage with no data compression. The fields are stored within distinct areas or pages of fixed size with the disc page being the minimum unit of transfer between main and secondary memory. The adjusted total number of tuples stored per page, TP, is therefore:

$$TP = \frac{PS}{TS \times BF} \quad \text{where } \begin{array}{l} PS = \text{assigned disc page size} \\ TS = \text{tuple size in bytes} \\ BF = \text{blocking factor} \end{array}$$

We arbitrarily define PS to be 512 bytes. The blocking factor is the percentage of usable space per page considering dynamic storage of the tuples and is set realistically to 60 %. A blocking factor of this order will be a DBMS implementation feature to allow for the efficient insertion, deletion and update

of individual tuples or for future modifications such as the addition of more attributes per tuple. Each unique tuple identifier (TID) completely determines the location of a tuple including both the address of the page in memory and the offset from the top of the page.

Index tables on single or multiple attributes are stored, in lexicographic or numeric sorted order, as balanced binary trees with the TID's located in the leaves. Access to all TID's is therefore $O(\log_2 n)$ where n is the number of tuples. The size of a TID is fixed can be realistically set to 4 bytes. As a result we can improve the blocking factor to 80% for index pages setting L , the number of TIDs per page, to 100.

We will also assume that the DBMS controls sufficient area within memory to manipulate two index and two tuple pages simultaneously with extra workspace available for the join or projection products. The amount of core memory available as tuple workspace is limited to 20K bytes in this example.

Tuples are either stored randomly throughout the segment of memory assigned to the database or in sequential clusters, ordered on one or more attributes. Random storage implies that ' m ' accesses will be required to retrieve ' m ' individual tuples (assuming the size of the database, in disc pages, is much greater than m). Clustering on the other hand significantly

reduces the number of accesses required. Retrieving the m tuples now requires only:

$$\frac{m}{TP} \text{ accesses,}$$

where TP is the adjusted number of tuples per page.

In order to simplify the analysis we also assume that indices on joining attributes are always available. This follows from the results obtained by [BLAS77] which show that the creation of such indices are cost effective in all cases. The alternative is to assume a full search of the memory segment which would unfairly bias the final cost estimate. Furthermore we assume that tuples are clustered about the attributes in the left side of the dependency used in the decomposition which created the scheme. This is reasonable because the left side attributes are always prime and are therefore important to ensure lossless joins. Intermediate results produced by projections and joins will certainly be clustered in a realistic system. As a result of these assumptions we consider only clustered storage which simplifies the overall cost estimation process. Fig. 2-12 summarizes the physical arrangements discussed so far.

Summary of Physical Storage Details

Figure 2-12

We now consider the join and project operations and estimate their total cost. A join of m tuples from relation $R1$ with n tuples from relation $R2$ will involve the following costs:

1. to retrieve the index pages for both relations;
2. to rewrite a new index for the combined relation;
3. to retrieve all tuples from both relations; and
4. to rewrite the new relation to disc.

The costs for the index operations will be:

$$2 \left(\frac{m+n}{L} \right) \text{ where } L \text{ is the number of index entries per page adjusted by the blocking factor.}$$

while the cost for the join will be:

$$2 \left(\frac{m}{TP1} + \frac{n}{TP2} \right) \text{ where } TP1 \text{ and } TP2 \text{ are the adjusted number of tuples of } R1 \text{ and } R2 \text{ respectively, per page}$$

The cost to project a relation is dependent upon both its final size and whether a key of the original relation remains intact in the projected version. The presence of the original key in a projection guarantees that there will be no duplication and so a sort is not required in this case. If the product is less than the allowable workspace then the sort for duplicate tuples can be conducted concurrently with the join or retrieval operation and no further work is needed. On the other hand if the final size exceeds the available workspace then we must include the cost to perform a subsequent sort. Following [KNUT73] we consider a 2 way external merge-sort with a cost proportional to

$$2 \left(\frac{m}{TP} (\log_2 (\frac{m}{TP} - 1)) \right) \text{ accesses}$$

where m is the size of the relation in tuples.

Since each page can be sorted prior to being written to disc, the first sort of the data can be done before the actual merge is begun and for this reason we deduct one complete pass from the expected number of required accesses.

2.4.2 QUERY PROCESSING STRATEGIES

A typical query in a relational calculus based language such as SEQUEL would be given as:

```
SELECT  attribute list
FROM    relation list
WHERE   predicate list
```

Such queries specify all source relations explicitly and the query processor can then optimize it based on the available indices and predicates. In contrast, a query from the design aids profile must simply be given as:

```
SELECT  attribute list
WHERE   predicate list
```

omitting the source relations because they are not known beforehand. The query profile does not consider predicates and so these are omitted as well. Query processors which do not require a 'FROM' clause have been studied in [SCHE77] [TANA79] [LOZI80] as a means to improve the usability of query languages.

The problem is therefore to compose a strategy of projections and lossless joins to retrieve the required attributes from any given schema.

It has been shown elsewhere that the projection and join operations have commutative properties that assist the development of an efficient query strategy. Specifically we note that the join operation is fully commutative such that :

$$A * (B * C) = (A * B) * C.$$

The project is also commutative over the join provided that we do not remove the necessary lossless join attributes and so we may write:

$$(A * B)[C] = A[C_1] * B[C_2]; \text{ where } C_1 \cup C_2 = C.$$

We therefore use an efficient though not necessarily optimal strategy which reduces the size of intermediate results by projecting out unnecessary attributes at the earliest opportunity. Proofs of these properties may be found in [TOTH80] [ULL80].

Two possible approaches to this problem are now presented. The first approach discussed in [SCHE77] uses a non-directed graph representation for the schema in which each vertex is a scheme and an edge is defined between two vertices if the intersection of the attributes in each vertex contains the left side of an FD or MVD valid in either scheme. The lossless join condition is thereby maintained. The strategy is then determined

by a sequence of joins forming a path through the graph, visiting nodes which contain the target attributes, until all have been collected. An alternate approach is to use the decomposition tree as the fundamental data structure and seek the required attributes from the children of each node in a recursive manner. We note that because of the form of the decomposition, the only attributes common to each child are those in the left side of the dependency used to decompose (called the junction root). The remaining attributes are therefore disjoint. We can maintain lossless joins by requiring that the junction roots be included in the products returned from either child.

EXAMPLE 2.6 Further description of these methods is best done by example. We return to the university database where we define the following attributes: CLASS(denoted by a), SECTION(b), STUDENT(c), MAJOR(d), EXAM_RESULTS(e), YEAR(f), INSTRUCTOR(g), RANK(h), SALARY(i), TEXT(j), ROOM(k) and DAY(l). Consider the following schema where the underlined attributes are those in the LHS of a valid FD or MVD:

1. CLASS SECTION STUDENT EXAM_RESULTS (abce);
2. STUDENT MAJOR YEAR (cdf);
3. INSTRUCTOR RANK SALARY (ghi);
4. CLASS SECTION INSTRUCTOR (abg);
5. CLASS TEXT (aj); and
6. CLASS SECTION DAY ROOM (abkl).

We define the graphical and decomposition tree data structures for the above schema and demonstrate each approach for the query

SELECT INSTRUCTOR, STUDENT, MAJOR (ie cdg)

in Figures 2-13 and 2-14.

To recover the query cdg from the given schema using the graph approach we must determine the join sequence which forms a path visiting vertices containing all of the query attributes. In this case the path R4-R1-R2 would suffice. The approach used in [SCHE77] is to consider the least number of joins. We join all vertices to all neighbours with connecting edges producing new clusters. These clusters are then joined to all neighbours and so on until a cluster contains all of the required attributes. When such a cluster is found the query strategy is then defined strictly as a series of required joins with no consideration for projecting out unnecessary attributes. In this case it would take two complete iterations to find the strategy:

$$((R4 * R1) * (R1 * R2)) [\text{cdg}].$$

Graph representation Approach
Figure 2-13

In the decomposition approach at each non-leaf node we seek a target set of attributes from the children below. To recover cdg from the tree above we first note that cd are in the left sub-tree and g is in the right and that we also need ab in order to join the products losslessly. We therefore request $abcd$ from the left child and abg from the right. In the same manner, the left child requests abc from its left child and cd from the right. At the leaf nodes $R1$ and $R2$ we can immediately project out e and f (which are unnecessary) returning abc and cd which are losslessly joinable on c (the junction root). In the right subtree of R the request for abg is passed directly to $R4$ which is returned intact. Finally, abg and $abcd$ are joined on ab and projected on cdg completing the query as:

$$((R1[abc] * R2[cd]) * R4) [cdg].$$

3NF decomposition tree Approach

Figure 2-14

We prefer the latter approach for the following reasons:

1. the decomposition tree data structure is already used in the decomposition algorithm and therefore the creation of an alternate data structure is not needed;
2. a recursive tree based algorithm is conceptually simpler;
3. because the algorithm keeps track of its required target list at each level, projections can be easily determined at the lowest levels - this removes the unnecessary attributes and gives a more efficient algorithm.

We now present a verbal description of the tree walking algorithm. At each node in the tree the algorithm selects its course of action based on the disposition of the attributes in the target set it is to return to its parent. A leaf node simply projects its contents onto the target set and returns the appropriate cost. Non-leaf nodes must determine whether the required attributes are strictly on one side, split up or, in the case of a redundant decomposition, in both children. Taking each of the choices separately we consider the first case. If the attributes are strictly on one side then the node simply requests the target set and the cost to recover it from the appropriate side. If the attributes are split with some located in both children, then we must consider that the subproducts from each side must be joinable in a lossless fashion. In this case we add the junction root to the target set and seek the new target set from either side. The total cost for the operation is therefore the costs to recover the target sets from the children, join the

subproducts and then project the result onto the original target set. In the last case, with redundancy, we seek the original target set from both sides and take the choice with the least cost. We cannot claim that this method determines the optimal method but we do claim that it represents a reasonable estimate.

CHAPTER 3

SCHEMA DECOMPOSITION ALGORITHMS

3.1 INTRODUCTION

This chapter describes the algorithmic approach to schema design using the decomposition method. We briefly discuss the Bernstein synthetic algorithm [BERN76] and contrast it with two decomposition algorithms found in [ZANI81] and [LIEN81]. We then conclude the chapter by describing the limitations of the decomposition approach and the compromise solution implemented in the design aid.

In chapter 1 we defined three normal forms of principal interest - 3NF, BCNF, and 4NF and stated that the aim of the design process was to create a non-redundant schema composed of schemes in the highest form of normalization possible while still maintaining the representation of both data and dependencies.

Bernstein's research concludes that 3NF is the highest normal form which can be achieved while maintaining both goals simultaneously. There are schemes which cannot be normalized to BCNF or 4NF, or higher, which will still maintain the total separation of dependencies and independence of data that we originally sought. Furthermore, Beeri and Bernstein's research [BEER79] reveals that determining whether a scheme is in BCNF or 4NF is in the class of NP complete problems because the decision requires information drawn from the complete closure of dependencies. Clearly we must therefore adopt a practical approach to the design method and examine the tradeoffs concerned.

3.2 THE BERNSTEIN SYNTHESIS ALGORITHM

We begin this section by describing the Bernstein algorithm (detailed in Fig. 3-1 [BERN76]) and its use for a particular example using the techniques discussed so far. Our aim is to demonstrate both the utility and the limitations of the synthetic approach which can then be compared to two current decomposition techniques.

- (1) Reduce all dependencies in F to fully functional form.
- (2) Find a minimal covering F' for F .
- (3) Partition dependencies in F' with common left sides into separate groups.
- (4) Merge equivalent keys. For each bijection $X \rightarrow Y$ and $Y \rightarrow X$ merge group X and group Y and store $X \rightarrow Y$ and $Y \rightarrow X$ in group J temporarily. For each attribute a in Y remove $X \rightarrow a$ from the X group if present. Do the same for each b in X with $Y \rightarrow b$ in the Y group.
- (5) Eliminate any induced transitive dependencies by again reducing F to a minimal cover. Replace all dependencies in J into their appropriate groups.
- (6) For each group construct a relation including all of the attributes in the group (the left sides of each dependency in the group form the candidate keys).

The Bernstein 3NF Synthesis Algorithm in Simplified Form
Figure 3-1

EXAMPLE 3.1 In the sample database (described in Fig. 3-2) we record information about a company's employees (uniquely identified by their social insurance numbers - SIN# and their employee numbers - EMP#), their salary, the department in which they work, the projects that they are working on, and the workstation that they use for each project (such as word processing unit 1 or milling machine 5). The designer has noted the functional dependencies given in Fig. 3-2. In this particular company the designer has also noticed that no two employees with the same name work in the same department and so he wishes to define dependency d_7 . This is not a very strong constraint and is included only for illustrative purposes.

Entity-Relationship Description
of Example 3.1
Figure 3-2 (a)

Attributes :

SIN#	EMP#	EMP_NAME	WORK_STATION	SALARY
DEPT#	DEPT_NAME		DEPT_LOCATION	PROJ

GIVEN

- d1: EMP#->SIN# EMP_NAME
- d2: SIN#->EMP# EMP_NAME
- d3: EMP# EMP_NAME->DEPT
- d4: SIN#->DEPT
- d5: EMP# PROJ->WORKSTATION
- d6: SIN# WORKSTATION->PROJ
- d7: EMP_NAME DEPT->SALARY
- d8: EMP#->SALARY
- d9: DEPT#->DEPT_NAME DEPT_LOCATION

Dependency Description of Example 3.1
Figure 3-2 (b)

Following step 1 of the algorithm, we begin by reducing all dependencies to fully functional form. In this example only d3 requires modification. Using the membership algorithm we can show that the dependency $EMP\# \rightarrow DEPT$ F+ because of $d1: EMP\# \rightarrow SIN\#$ and $d4: SIN\# \rightarrow DEPT$. This allows us to remove EMP_NAME from the left side of d3. In Step 2 we then consider each dependency in the arbitrary order d9 to d1 testing for membership in the closure defined by the remaining dependencies. In this manner we find that d8 is in fact redundant because it can be inferred using d3, d4 and then by transitivity with d7. The same is also true for d4. The minimal cover is therefore the original cover less d8 and d4. We then group the remaining dependencies with common left sides as shown in Fig. 3-3(a). Step 4 is meant to collect those attributes which are logically equivalent into the same group as we would expect them to be. $EMP\#$ and $SIN\#$ are both equivalent and so groups one and two are united with the dependencies $EMP\# \rightarrow SIN\#$ and $SIN\# \rightarrow EMP\#$ temporarily moved to group J. Not quite so obvious however is the equivalence of groups 3 and 5 - however rewriting d3 and d4 as:

1. $EMP\# PROJ \rightarrow EMP\# WORK\ STATION$
2. $SIN\# WORK\ STATION \rightarrow SIN\# PROJECT$

and noting the equivalence of $SIN\#$ and $EMP\#$ we now see that these groups also need to be united. Since the merging of groups may re-introduce the possibility of new transitive

dependencies we must again reduce to a minimal cover. In this case we already have a minimal cover and we simply return dependencies d1-d4 to group two. This leaves the groups as shown in Fig. 3-3(b).

SIN#->EMP# EMP_NAME EMP#->SIN# EMP_NAME SIN# WORKSTATION->PROJ
 EMP#->DEPT

(1)

(2)

(3)

EMP_NAME DEPT->SALARY

EMP# PROJ->WORKSTATION

(4)

(5)

DEPT#->DEPT_NAME DEPT_LOCATION

(6)

Initial Grouping
 Figure 3-3(a)

SIN#->EMP# EMP_NAME SIN# WORKSTATION->PROJ EMP_NAME DEPT->SALARY
 EMP#->SIN# EMP_NAME EMP# PROJ->WORKSTATION
 EMP#->DEPT

(1)

(2)

(3)

DEPT#->DEPT_NAME DEPT_LOCATION

(4)

Final Grouping
 Figure 3-3(b)

Dependency Grouping - Example 3.1
 Figure 3-3

2

Concluding the algorithm in Step 6 we form the relations R1, R2, R3 and R4 as shown in Fig. 3-4 with the candidate keys underlined in each.

```

R1 ( SIN# EMP# EMP_NAME DEPT )
R2 ( EMP_NAME DEPT SALARY )
R3 ( DEPT DEPT_NAME DEPT_LOCATION )
R4 ( EMP# SIN# PROJ# WORK_STATION )

```

```

-----
-----
-----
-----
-----
-----

```

Final Schema Design - Example 3.1

Figure 3-4

The final design is indeed in 3NF, preserves all dependencies and has no data or dependency redundancies; however, there remains a potential problem between relations R1 and R4. By strict definition, scheme R4 is not in BCNF because the determinants EMP# and SIN# are not candidate keys - they are candidate keys only in scheme R1. Using these attributes as candidate keys in scheme R1, we ensure that no two tuples are to be allowed to have the same SIN# or EMP#. Scheme R4's candidate keys, however, indicate that no two tuples are allowed to have the same:

1. EMP# and PROJ; or
2. SIN# and PROJ; or
3. EMP# and WORK_STATION; or
4. SIN# and WORK_STATION.

This does not preclude the possibility of entering a tuple with a different SIN# and EMP# combination which does not match the information found in R1. Ultimately, through an inadvertent error, an employee may end up having an incorrect SIN# and EMP# combination in the database even though the series of transactions are valid given the existing dependency constraints - a potentially serious anomaly. In addition, by ignoring the obvious non-functional dependencies we leave the design relatively inefficient. For example if we seek to know all of the projects for a particular department using this schema we must find all employees in the department and then, through them, all of the projects currently underway. Projects which have no current workers but are allocated against particular departments cannot be entered - again a serious deficiency. A partial solution to this problem was also presented by Bernstein [BERN76] using the dummy attribute $\emptyset$. For example we express the dependency $EMP\# \twoheadrightarrow PROJ$ as $EMP\# PROJ \rightarrow \emptyset$, using a unique $\emptyset$ attribute for each such dependency. However this solution will lead to the creation of many separate schemes, each covering a unique nonfunctional relationship and inducing redundancy into the final design.

Such problems are symptomatic of the inability of the synthesis algorithm to cope with the higher normal forms without direct intervention from the designer. As Bernstein points out in [BERN76], the final class of acceptable 3NF products produced by this approach is really quite small and the choice of which schemes are to be assigned to cover the given set of FDs is driven by logical and not practical considerations. Therefore if we wish to take advantage of the improved freedom from anomalies offered by the higher normal forms or to adapt the schema design to reflect more efficient considerations of non-functional relationships, then we must go beyond the basic synthesis approach to achieve this goal. This has led to alternate algorithmic design methods which we discuss in the next section.

3.3 THE ZANIOLO AND LIEN DECOMPOSITION ALGORITHMS

3.3.1 INTRODUCTION

The decomposition method was first proposed in [FAGI79a] to address the limitations of the synthetic approach by allowing the designer to consider both functional and multivalued dependencies with the aim of achieving a schema composed of 4NF schemes. Working from the same initial universal relation we recursively decompose schemes which are not in 4NF into sub-schemes which are. We immediately encounter two problems with this method -

firstly, to determine if a given scheme is in 4NF and then, secondly, to select the best dependency to use at each step of the decomposition. The selection of which dependency to use is particularly acute. While in the synthetic approach we were able to discard redundant attributes or dependencies without affecting the overall FD representation, the effective partitioning of attributes through the decomposition process can result in the loss of dependencies and the introduction of inter-relational constraints.

Two different approaches can be found in the algorithms proposed in [ZANI81] and [LIEN81] and are discussed in this section. The first alternative, followed in the Zaniolo algorithm, emphasizes the preservation of representation at all stages of the decomposition while the second, as used in the Lien algorithm aims to reduce the scheme to its most normalized form without consideration for the lost dependencies. We describe the approach in [ZANI81] as having 'representation priority' while [LIEN81] has 'normalization priority'. These alternatives present the designer with a difficult decision - to accept the anomalies in an unnormalized relation or to ignore the lost dependencies and normalize to the greatest degree possible. We conclude this section with a discussion of the implications of each.

3.3.2 A REPRESENTATION PRIORITY APPROACH

We have already introduced the concept of 'complete relatability' proposed in the Zaniolo algorithm. At each stage of the decomposition, the decomposing dependency is chosen from those that preserve the dependency representation of the original scheme. That is, the closure of the dependencies in the sub-schemes must be the same as the closure of the original scheme. In addition, the authors of [ZANI81] introduce a novel way of expressing the dependency structure of the initial scheme, which forms an integral part of their algorithm. We must therefore begin this discussion by briefly introducing the concept of an 'elementary dependency'.

A dependency is said to be 'elementary' if the following conditions are satisfied:

1. the left and right sides are disjoint; and
2. no other dependency exists with its left and right sides completely contained in the left and right sides of another.

The latter condition implies that we consider only the minimal elements of the partial ordering of dependencies where:

for: $d1: A \rightarrow B$ (or $A \twoheadrightarrow B$) and $d2: C \rightarrow D$ (or $C \twoheadrightarrow D$)
 $d1 < d2$ implies that $A \subseteq C$ and $B \subseteq D$.

The redundant decomposition shown in Example 1.4 demonstrated the effects of using other than minimal elements. If we group the

elementary dependencies with common left sides, as in the synthesis algorithm, then each member of those groups with more than one element is called 'multiple'. Hence in the following group of dependencies:

```

d1 : abc -> de
d2 : a ->-> e
d3 : a ->-> f
d4 : b ->-> g

```

d4 is an 'elementary' MVD while d2 and d3 are 'multiple' elementary MVDs.

Elementary dependencies are shown to have beneficial properties for decomposing the initial schema. A theorem proven in [ZANI81] shows that the set of elementary FD's (denoted by F') forms a minimal cover for the FD closure of the original scheme. Using the multiple elementary MVD's and removing one element from each group is then shown to form a minimal MVD cover (given as GM').

It is therefore implicit in the Zaniolo algorithm that the designer must generate all of the elementary dependencies as the initial input to the algorithm. As we shall see in Example 3.2, this is not a trivial task. Let us now follow the complete algorithm shown in simplified form in Fig. 3-5 for a database similar to the one presented as Example 3.1.

A REPRESENTATION PRIORITY ALGORITHM

```

    procedure DECOMPOSE (R); to decompose scheme R
    begin
Step 1      DETERMINE (F', GM'); the elementary FDs and
            multiple elementary MVDs for scheme R
Step 2      If GM =  $\emptyset$  THEN SCHEME R IS IN 4NF
            else
Step 3      begin
            found  $\leftarrow$  false
            for each MVD X $\rightarrow$ Y in GM while NOT (found) do
            begin
Step 4      split scheme (R, {into}, R1, {and}, R2)
            determine (F', GM')
                    R1      R1

            determine (F', GM')
                    R2      R2
Step 5      {test for complete relatability}
            if (F'+ = (F' + F')+) and
                    R1      R1
            (GM'* = (GM' + GM')*) then
                    R1      R2
            begin
Step 6      decompose (R1);
            decompose (R2);
            found  $\leftarrow$  true ;
            end
            end
            if NOT(found) then
            output 'CANNOT DECOMPOSE THIS SCHEME'
            end
    end.

```

A Representation Priority Algorithm by Zaniolo
Figure 3-5

EXAMPLE 3.2 Consider the scheme:

R (EMP#, SIN#, EMP_NAME, EMP_ADDRESS, DEPT#, PROJECT)

where the designer has noted the following set of dependencies D:

1. d1: EMP# -> SIN#
2. d2: SIN# -> EMP#
3. d3: EMP# -> EMP_NAME EMP_ADDRESS
4. d4: EMP# EMP_NAME -> DEPT
5. d5: EMP# -> DEPT
6. d6: EMP# ->-> PROJ
7. d7: DEPT ->-> PROJ
8. d8: DEPT ->-> EMP# SIN# EMP_NAME EMP_ADDRESS
9. d9: SIN# -> DEPT

First of all, since dependency d5 is contained within d4, the latter is not elementary and may be deleted. The set of remaining dependencies do not constitute the complete set of elementary dependencies required for step one. For example we have not considered the equivalent FDs for d3 and d6 using SIN# instead of EMP#, nor have we included the MVD counterparts for all FDs. Using the notation d' for an MVD counterpart of an FD and a second identifying digit for the substitution of an equivalent left side attribute in an original dependency we can therefore add:

1. d31': EMP# →→ EMP_NAME
2. d32': EMP# →→ EMP_ADDRESS
3. d33': SIN# →→ EMP_NAME
4. d34': SIN# →→ EMP_ADDRESS
5. d61 : SIN# →→ PROJ

as well as the remaining MVD counterparts. After arbitrarily deleting one member from each MVD group with a common left side we are left with the minimal FD and MVD covers:

$$F' = \{d1, d2, d5, d9\}$$

and

$$GM' = \{d1', d2', d31', d31', d34', d5', d6, d61, d7, d9'\}.$$

Clearly GM' is not null, indicating that the scheme is not normalized, and so we proceed to step three.

Consider the choice of d7 as the first decomposition giving:

R1 (DEPT, PROJ)

R2 (DEPT, SIN#, EMP#, EMP_NAME, EMP_ADDRESS)

Using the projectivity property for both FDs and MVDs the following dependencies are found to hold for R1 and R2;

1. $F'_{R1} = \emptyset$
2. $GM'_{R1} = \{d7\}$
3. $F'_{R2} = \{d1, d2, d5, d9\}$
4. $GM'_{R2} = \{d1', d2', d31', d33', d34', d5', d7, d9'\}$

Testing the complete relatability conditions (CRC) in Step four reveals that d7 is a valid decomposition because the only missing dependencies are d6 and d61 which can be derived from the FD and MVD cover formed by:

$$(F'_{R1} \cup F'_{R2}) \text{ and } (GM'_{R1} \cup GM'_{R2})$$

For example, d61 can be inferred using:

1. $d2' : \text{SIN\#} \rightarrow \text{EMP\#}$
2. $d33' : \text{SIN\#} \rightarrow \text{EMP_NAME}$
3. $d34' : \text{SIN\#} \rightarrow \text{EMP_ADDRESS}$
4. $d9' : \text{SIN\#} \rightarrow \text{DEPT}$

Since, by the union rule

$$\text{SIN\#} \rightarrow \text{EMP\# EMP_NAME EMP_ADDRESS DEPT}$$

we can infer by complementation that

$$\text{SIN\#} \rightarrow \text{PROJ.}$$

In fact, d7, d33', and d34' are the only valid decomposition choices at this stage. Arbitrarily selecting d7 we divide the scheme into R1 and R2 and then decompose each sub-scheme in step six. For R1 we find GM' is now null; therefore no further decomposition is required and GM' is an atomic

scheme for our final design. The candidate key for R1 is DEPT# and PROJ (all key) since the left side DEPT# does not uniquely identify a tuple in the scheme. Continuing the process for R2 in this manner with an arbitrary choices for each decomposition finally results in the following possible schema:

1. R1 (DEPT# PROJ)
2. R2 (SIN# DEPT)
3. R3 (SIN# EMP_ADDRESS)
4. R4 (SIN# EMP#)
5. R5 (SIN# EMP_NAME)

which can be represented by the decomposition tree shown in Fig. 3-6.

Final Decomposition Tree for Example 3.2
Figure 3-6

We can immediately see that all schemes in this design are in 4NF and are therefore free of the anomalies suffered by the similar synthesis example. It also handles the relationship between departments, employees and projects in a much more efficient manner. On the other hand, the design seems to be overnormalized with the separate schemes R2, R3, R4 and R5 containing information about a single entity which could be merged into a single 4NF scheme. These small binary schemes are not very practical for a final design and so further work will be needed to consolidate them into more meaningful ones.

The reason for this behaviour lies in the method chosen to determine if a scheme requires further decomposition by using the groups of multiple MVDs. We recall that the complementation property for MVDs states that if a given MVD $X \twoheadrightarrow Y$ holds in the scheme $R(X,Y,Z)$ (X,Y,Z disjoint) then $X \twoheadrightarrow Z$ must hold as well. By this assumption, if we eliminate all attributes contained in Z from R then only one MVD with left side X will remain. By deleting groups with only one element, we remove $X \twoheadrightarrow Y$ from further consideration.

This method does not handle FDs with more than one attribute on the right side very well. For example the dependency $a \twoheadrightarrow bcde$ is transformed into the multiple MVD group $a \twoheadrightarrow b$, $a \twoheadrightarrow c$, $a \twoheadrightarrow d$, and $a \twoheadrightarrow e$ (from which, one is to be deleted) which will lead us to decompose the scheme $abcde$ unnecessarily. It would

seem natural and reasonable to avoid this behaviour in the design process.

In summary this approach makes effective use of the representation principal and can resolve certain anomalies not previously handled by the simple synthesis algorithm. Practically, though, the proliferation of elementary dependencies, even for this small example, shows great potential to be a source of errors or, in large designs, to be totally unmanageable. Finally the criteria chosen to determine if further decomposition is required does not follow from the basic definition of fourth normal form and should be modified. The Lien approach, which we shall now describe, addresses these very problems in a completely different manner.

3.3.3 A NORMALIZATION PRIORITY APPROACH

In contrast to the previous algorithm, the approach taken by Lien is much simpler and does not consider dependency representation in its selection of the dependency with which to decompose. We therefore highlight two important aspects of Lien's approach - the method used to determine if decomposition is necessary and the selection procedure for the best dependency on which to decompose. An example is then presented to show the effect of ignoring dependency representation in a practical situation.

Realistically, to determine if a scheme is in BCNF or 4NF the designer must always analyze the valid FD or MVDs which hold in the scheme using the criteria for each normal form. A scheme which is not 4NF can therefore be considered to be 'decomposable' based on the MVD which violates the 4NF condition. More specifically a scheme is decomposable if there exists an MVD (which is not the direct counterpart of a FD) such that:

1. $X \subseteq R'$; the left side X is completely contained in R'
2. $Y \cap R' \neq \emptyset$; some part of Y is contained, and
3. $R' - (X \cup Y) \neq \emptyset$; other attributes exist as well.

Hence a scheme $R'(a,b,c,d)$ is said to be decomposable by $a \twoheadrightarrow cd$.

Lien's approach follows the practical designer's method - simply scan the existing dependencies and check each against the decomposability criteria. In this manner those schemes which are decomposable can then be dealt with ultimately yielding a final design of non-decomposable schemes. This final design may not be in 4NF, however, if the original cover implies certain dependencies which are not explicitly listed.

EXAMPLE 3.3 Consider the scheme:

R(EMP#, PROJ, WORK_SITE, STARTING_TIME, QUITTING_TIME)

with the reasonable set of dependencies:

1. d1: EMP# $\rightarrow$ -> PROJ
2. d2: CONTRACTOR $\rightarrow$ -> WORK_SITE
3. d3: PROJ WORK_SITE $\rightarrow$ -> STARTING_TIME
4. d4: PROJ WORK_SITE $\rightarrow$ -> QUITTING_TIME.

Hence an employee may work on many projects, each of which can involve several work sites. The combination of a particular PROJ and WORK_SITE determines at least one established starting and quitting time. Furthermore let us assume that the relationship between the employee and the contractor cannot be explicitly defined by the designer.

One possible decomposition results in the schema:

R1 (EMP# PROJ)

R2 (CONTRACTOR WORKSITE), and

R3 (EMP# CONTRACTOR STARTING TIME QUITTING TIME)

which is no longer decomposable with the given set of dependencies. R3 is not in 4NF because the scheme can be further decomposed using the dependency:

EMP# CONTRACTOR $\rightarrow$ -> PROJ STARTING_TIME

which can be inferred from the original set.

Significantly, the concept of 'elementary dependencies' discussed in the previous section does not solve this problem

either. If we were to add the inferred dependency to the proposed list of elementary dependencies, we would immediately see that d_1 is completely contained within it and thus by partial ordering, the inferred dependency would be discarded. Hence the Zaniolo algorithm does not always result in 4NF designs either.

Lien defines a '4NF cover' as an initial cover which gives enough dependency information to ensure that such problems do not arise. Practically, therefore, to avoid the need to exhaustively determine the dependency basis for each possible combination of attributes, we must assume that the given set of dependencies forms a 4NF cover. Since a minimal cover reduces the possibility of inherent redundancy in the final design, Lien then employs the method used in the synthesis algorithm to reduce the initial 4NF cover to a minimal one. This has the significant advantage of not requiring the consideration of the many elementary dependencies which hold for the database.

The remaining dependencies can still result in redundancies if they are used in the wrong order and so Lien proposes a two step treatment which is very similar to the Zaniolo approach. In the first step we consider dependencies which have left sides which are themselves decomposable by others in the cover by removing the redundant attributes. For example, the dependency

$$d_1: abcd \rightarrow e$$

has a left side which can be decomposed using

$d2:a \rightarrow \rightarrow def.$

By the theorem proven in [LIEN81] $d1$ can be rewritten as

$ad \rightarrow \rightarrow e.$

In the general case, for:

$Z \rightarrow \rightarrow W$

where Z is decomposable using $X \rightarrow \rightarrow Y$ it can be shown that

1. $W \cap Y = W$ or $\emptyset$; and therefore

2. $X \cup (Y \cap Z) \rightarrow \rightarrow W \cap Y$

The reduction technique is therefore useful for the case when

$W \cap Y = W.$

Lien's analysis closely follows the approach discussed in [ZANI81] using the properties of elementary dependencies.

When all such problems have been resolved the reduced dependencies are then ordered by left sides in a similar manner to that used in [ZANI81]; that is for two groups $g1$ and $g2$ with left sides $g1.LS$ and $g2.LS$ the partial order $g1 < g2$ implies $g1.LS \subseteq g2.LS$. The remaining minimal dependencies are then used in arbitrary order to effect the decomposition.

It is our opinion that this latter step in fact renders the reduction technique totally unnecessary. Consider any two dependencies $Z \rightarrow \rightarrow W$ and $X \rightarrow \rightarrow Y$ where the conditions stated above are satisfied - that is Z is decomposable using $X \rightarrow \rightarrow Y$ and $W \cap Y = W$. We follow the Lien approach and replace the dependency by its reduced form $X \cup (Y \cap Z) \rightarrow \rightarrow W$. Since the left side of

any reduced dependency still contains the left side X of the original decomposing dependency $(X \twoheadrightarrow Y)$ it therefore must in all cases be a non-minimal dependency. Furthermore, by the decomposition property, when we use the minimal dependency $X \twoheadrightarrow Y$ on any initial scheme, we partition the attributes such that the only attributes in common are those in the left side X . Hence the reduced dependency $X \cup (Y \cap Z) \twoheadrightarrow W$ is no longer useful for any further decompositions. We claim, therefore, that the partial order of the minimal cover dependencies meets all of the requirements of the Lien approach for all practical schemes.

Using the simplified description of the algorithm in Fig. 3-7, let us now use this technique for the following example adapted from [ULL81].

PROCEDURE DECOMPOSE (R); to decompose scheme R.

```

begin
  if R is NOT in 4NF then
    begin
      Step 1    reduce the MVD cover D to a minimal cover D'
      Step 2    partition D' into groups with common left sides
                and partially order the dependencies in D' with
                common left sides such that for
                g1:X->->Y and g2:Z->->W
                g1 < g2 implies X  $\subseteq$  Z;
      Step 3    with the minimal member of an arbitrary group do
                begin
                  split R into R1 and R2;
                  decompose (R1);
                  decompose (R2);
                end;
    end;
  end.

```

A Normalization Priority Algorithm by Lien
Figure 3-7

EXAMPLE 3.4 We consider an investment database as shown in Fig. 3-8, composed of the attributes Investor (a), Broker (b), Stock (c), Dividend (d), Quantity Held (e), and Broker's Phone No (f) with the following dependency cover D:

1. d1: a $\twoheadrightarrow$ b; an investor has only 1 Broker
2. d2: c $\rightarrow$ d; a stock declares only 1 dividend
3. d3: b $\twoheadrightarrow$ f; a broker can be reached at several numbers, and
4. d4: ac $\rightarrow$ e; the investor holds a fixed quantity of each stock.

Example 3.4 - ER Representation

Figure 3-8

Let us first assume, for the purpose of the decomposition, that all of the dependencies are multivalued and so we use the MVD counterparts for d_1 , d_2 , and d_4 . Applying the membership algorithm for each dependency of D we find that none are redundant and conclude step one with the original cover D . As discussed, we shall ignore the reduction technique proposed in the Lien algorithm. Partially ordering the dependencies in step two reduces D to three groups which are:

1. $a \twoheadrightarrow b$, $ac \rightarrow e$
2. $c \twoheadrightarrow d$; and
3. $b \twoheadrightarrow f$.

The arbitrary selection of dependencies in step three can yield some very different results. For example, using the dependencies in the order one to three, we get the following design:

1. R_1 (INVESTOR (a), BROKER (b))
2. R_2 (INVESTOR (a), STOCK (c), QUANTITY_HELD (e))
3. R_3 (STOCK (c), DIVIDEND (d)), and
4. R_4 (INVESTOR (a), STOCK (c), BROKERS_PH# (f))

In selecting this order we have lost the relationship between the Broker and his phone numbers which is far more important than the relationship between the investor, his

stock, and the broker's phone number given in R4. Although this schema meets Lien's goal of non-redundancy,⁴ it is clearly not a good design. Choosing instead the order three, two and then one, however, gives the schema;

1. R1 (BROKER (b), BROKERS_PH# (f))
2. R2 (STOCK (c), DIVIDEND (d))
3. R3 (INVESTOR (a), BROKER (b)) and finally
4. R4 (INVESTOR (a), STOCK (c), QUANTITY_HELD (e))

which supports all dependencies and is a much more efficient design.

Conceptually, Lien's approach is much simpler than the previous algorithm, and can be easily implemented using the basic algorithms discussed so far. The arbitrary choice of dependency in the last step can lead to some very different, and in some cases unsatisfactory, results, leaving us to conclude that the ~~basic~~ Lien approach is, by itself, not an effective technique.

3.3.4 SUMMARY

As the examples in this section have shown, there are many issues to be resolved prior to the implementation of a universally applicable schema design algorithm. Clearly we must draw from the advantages offered by each to prepare a useful and efficient compromise solution.

The maintenance of dependency representation as embodied in the Zaniolo algorithm seems to be a more important design objective than strict normalization. Although no general rule can be formulated, it is our opinion that the dependency constraints represent important semantic rules which should remain intact even when a scheme is left relatively unnormalized. The imposition of inter-relational constraints is somewhat akin to the act of patching an unsatisfactory program structure rather than re-examining the entire problem. Furthermore, current workers are recognizing that normalization, even to 4NF and 5NF is not sufficient to prevent all anomalies and are studying semantic data models as a means to augment these shortcomings. The Zaniolo approach offers the added advantage of recognizing whenever a dependency is not supported, ultimately giving the designer a role in the decomposition process.

Adapting the spirit of the Zaniolo algorithm to the simpler techniques used by Lien would therefore seem to be an effective compromise solution and was chosen for implementation in the design aid.

3.4 COMPROMISE SCHEMA DECOMPOSITION ALGORITHM

3.4.1 INTRODUCTION

In this section we describe the schema decomposition algorithm which was implemented in the design aid. A compromise of the two algorithms previously discussed, it is capable of decomposing a wider range of initial schemes. Furthermore, by considering the expected query cost estimates, the algorithm removes the various degrees of arbitrary selection found in the two previous algorithms. We defer examples of its use until chapter five where they are described in detail.

3.4.2 DESCRIPTION

In Fig. 3-9 we present the simplified details of the schema decomposition algorithm - the last algorithm to be described in this work. Since each step of the algorithm has been introduced in previous sections we stress only the tradeoffs considered for the final approach in this description.

PROGRAM SCHEMA DESIGN

```

        procedure decompose (Scheme)
        begin
Step 2      if Scheme is NOT in 4NF then
              begin
Step 3        find current minimal MVD cover;
              place cover in partial order;
              for each minimal MVD in cover do
                begin
Step 4          test equality of closures before and after split;
Step 5          IF test = OK then FIND COST FOR QUERY PROFILE
                  using current schema;
                end
Step 6          IF at least 1 test was OK then
                  for MVD with least cost and valid test do
                    begin
                      split scheme into scheme 1 and scheme 2;
                      decompose (scheme 1);
                      decompose (scheme 2);
                    end
                ELSE
Step 7          advise - scheme has no valid decomposition;
              end
              else advise - scheme is already in 4NF;
        end

        begin
Step 1      read (attributes, dependencies, queries);
            decompose (universal relation);
        end.

```

The Composite Decomposition Algorithm
Figure 3-9

We begin the algorithm with the designer interactively entering the database description including the attribute dependency and query profile details in step one. Each attribute is fully described using an appropriate name such as Employee_Number, its type (character or numeric), the maximum

length in bytes of any given value and the estimated domain size. Once entered, each attribute is assigned a distinct identifier - the letters 'A' to 'Z', then 'a' to 'z' - so that further reference is much easier. Next, the designer enters all of the dependency information including the FDs, MVDs and EMVDs along with their estimated multiplicity factors as appropriate. Lastly the query profile is entered referring to the attributes by their respective identifiers. Entering the EMVDs at the beginning of the decomposition is unique to this algorithm but was chosen to avoid the need to constantly interact with the user at each stage - which would increase both the time and expense of a design session. We now recursively call the procedure to decompose the initial scheme.

At the beginning of each call to Decompose in step two, we must determine if we have any work to do - that is, to determine whether the scheme is decomposable and hence not in 4NF. We examine each member of the set of dependencies currently valid for the scheme to determine if it can be used for a decomposition. By our assumption that the input forms a 4NF cover we can return a valid answer after considering only the given set of dependencies.

Proceeding with step three, we form the minimal cover in the manner used in the synthesis algorithm. The new set of dependencies is called MIN_DEC (for the minimal decomposing set).

A brief point of explanation is required here. In order to avoid the profusion of dependencies which results from the creation of MVD counterparts for each FD - especially those with more than one attribute on the right side - another set of detailed dependencies is automatically generated though this set is never seen by the user.

The partial ordering in step three directly follows the Lien algorithm. Only the minimal elements in each group are then considered as possible candidates for subsequent decompositions. Using the Zaniolo approach we then determine the valid dependencies for each resultant sub-scheme and check to see if the FD and MVD covers remain intact. For each valid decomposition we then determine the approximate cost to retrieve the query profile in step five.

Assuming that at least one choice of dependency retains the original closure, we choose the one with the least cost and recursively decompose each sub-scheme. Should none of the choices be successful, then we advise the designer of this fact and give him the choice of continuing the decomposition or leaving the scheme in its current form. If the designer wishes to continue the decomposition with the loss of one or more dependencies then for each possible choice we advise which dependencies are not maintained and the expected query profile cost.

Since a finite number of decompositions are possible, the algorithm must eventually terminate yielding an efficient schema with each scheme in 4NF unless specifically noted. Estimation of the overall time complexity is not possible because of the many factors involved.

By following the partial order imposed in step three and noting that the final schemes are no longer decomposable (as previously discussed), we can strengthen the properties of the final design by directly applying the proof presented in [LIEN81]. It can therefore be shown that the compromise algorithm produces a non-redundant relational schema in all cases.

CHAPTER 4

SOFTWARE DESCRIPTION

4.1 INTRODUCTION

This chapter gives the functional specification of the software written to implement the conceptual schema design aid. We begin by describing the overall system architecture and the standard interfaces between the main modules in the first section. In the next section we demonstrate a complete user session showing the various user options, prompts and error messages. Finally in the succeeding six sections we examine each module separately referring to the detailed summary of the significant routines provided in Appendix A.

For simplicity, we shall use the parameter passing modes used in ADA rather than the pass by value and pass by reference terms used in Pascal. This has the advantage of clearly

identifying the logical direction of the data without being overly concerned with how the data item is actually passed. We refer to a parameter passed to a routine as input as an 'IN' parameter while parameters which are products of the routine are referred to as 'OUT'. An IN_OUT parameter is therefore passed with one value (the input) and returned with another (the output).

4.2 SYSTEM ARCHITECTURE

The Relational Database Design Aid (RDD) is an interactive software tool to assist in the creation of efficient conceptual schemae, considering both semantic constraints and query processing efficiency.

RDD is composed of approximately 3,500 lines of Pascal code as used on the Honeywell Level 66 under the CP6 operating system. With the exception of two extensions (the INCLUDE directive and underscores in variable names - both easily removed) the complete system is fully transportable to any Pascal implementation supporting variable length SETS.

The six separate compilation unit modules and two supporting files which comprise the complete RDD system are organized by primary function as shown in Fig. 4-1. A design session is initiated and controlled by the Main Module which directs the

acceptance of the database description from the Input Module and then directs the Decomposition Module to carry out the subsequent design. The three supporting modules, Redundancy, Size, and Cost, are responsible for the determination of, respectively, the membership problem for FD and MVDs, the estimated 1NF scheme size for any given combination of attributes and finally the overall cost in secondary storage accesses for the current schema design. Together they implement the schema design algorithm shown in Fig. 3-9. All of the modules share a common data structure declaration and procedures library as indicated in Fig 4-1.

Relational Database Design Aid (RDD) - System Architecture
Figure 4-1

While the initial concept of the design aid was to provide the user with a broad range of interactive services (such as the ability to edit the database description, add new dependencies and so on) it became apparent in the course of development that the design process requires a greater degree of off-line review and analysis than was originally thought. As a result, we have considered a design session to be the treatment of only one application at a time. Instead, the user can save an application, edit the simple text file off-line and then recall the Database Description for a subsequent session thus avoiding the need to repeatedly re-enter all of the information.

4.3 A TYPICAL USER SESSION

4.3.1 INTRODUCTION

To demonstrate a complete RDD design session we first introduce the following sample database application adapted from [ZANI81].

EXAMPLE 4.1 The field service department of a local computer manufacturer is establishing a database consisting of the following attributes:

1. Customer# (a); for 5,000 customers
2. Customer_Name (b)
3. Customer_Address (c)
4. Computer_Model (d); for 100 different models
5. Quantity_Held (e)
6. Date_Of_Installation (f)
7. Technician# (g); for 100 technicians
8. Technician_Name (h)

with the following dependencies:

1. d1:CUST# -> CUST_NAME CUST_ADDRESS
2. d2:CUST# ->-> MODEL QTY DATE_INSTALLED, 3 { each }
3. d3:CUST# ->-> TECH# TECH_NAME, 2
4. d4:CUST# MODEL -> QTY DATE_INSTALLED
5. d5:TECH# -> TECH_NAME
6. d6:TECH# ->-> CUST# CUST_NAME CUST_ADDRESS
MODEL QTY DATE_INSTALLED, 6

and let us assume that the monthly status report requires a review for the attributes:

1. CUST# CUST_NAME CUST_ADDRESS MODEL QTY;
for billing purposes;
2. TECH# MODEL QTY DATE_INSTALLED;
to review technician workload; and
3. TECH# TECH_NAME CUST# CUST_NAME;
for expense account recovery purposes.

Let us now use RDD to prepare a useful design. {

4.3.2 ENTERING THE DATABASE DESCRIPTION

For the current implementation a command file has been prepared to prompt the user for initial information prior to the commencement of the actual RDD session. This heading is shown in the examples in Chapter 5 and is self-explanatory. Immediately upon program execution RDD will declare itself with the heading:

RDD Vers. 7.5 /

and request the user to select either an interactive session (from the terminal) or the recall of a previous application from the predeclared text file with the prompt:

Interactive Input ? (Y/N):.

In this case we are entering the application for the first time and select an interactive session. This session is presented with a series of sequential figures containing both correct and incorrect description data along with the options available to the user. We show the entry of the customer attributes in Fig. 4-2(a) and the system response to missing or incomplete entries.

	ATTRIBUTE	phase
*Customer#	N10,5000	
*Customer_Name	c25,5000	
*Customer_Address	5000	
*** Missing Attribute type (C or N). Size and Domain ?		
*C50	5000	

Response to Missing Information
Figure 4-2(a)

The missing information forced the user to simply reenter only the last two details. Fig. 4-2(b) shows the response to unwanted characters in the input.

```
*Com(puter_model, c25,100
^ *** Illegal Character ignored
^ *** Illegal Character ignored
^ *** Illegal Character ignored
```

Illegal Characters in Input Figure 4-2(b)

Input data is always checked against the predefined limits with the following prompts for out of limit values (Fig. 4-2(c)):

```
*Quantity_Held,n2222222222222222,500
*** Max Field Size = 256, Size and Domain ?
*7333333333333333333,3
*** Max Field Size = 256, Size and Domain ?
*2,10
```

Numeric Values Beyond Limits Figure 4-2(c)

In this case the attribute Quantity Held is strictly a relationship attribute and has no real domain size. The value of 10 is strictly arbitrary.

Duplicate entries are also prevented as shown in Fig. 4-2(d).

*Customer#.n10,5000

*** Attribute already defined .. Please reenter

Duplicate Entry Checks Figure 4-2(d)

At any time, the user can call for a summary of the currently defined attributes by entering a '?' as shown in Fig. 4-2(e).

ID	ATTRIBUTE NAME	TYPE	SIZE	DOMAIN SIZE
A	Customer#	Number	10	5000
B	Customer_Name	Char	25	5000
C	Customer_Address	Char	50	5000
D	Computer_Model	Char	25	100
E	Quantity_Held	Number	2	10

User Help - Attribute Summary Figure 4-2(e)

Completing the entry of the last attributes we terminate the phase with a period resulting in the summary shown in Fig. 4-2(f):

*Date_Of_Installation,n6,10

*Technician#,n5,100

*Technician_Name,c50,100

*,

ID	ATTRIBUTE NAME	TYPE	SIZE	DOMAIN SIZE
A	Customer#	Number	10	5000
B	Customer_Name	Char	25	5000
C	Customer_Address	Char	50	5000
D	Computer_Model	Char	25	100
E	Quantity_Held	Number	2	10
F	Date_Of_Installation	Number	6	10
G	Technician#	Number	5	100
H	Technician_Name	Char	50	100

DEPENDENCY Phase

The Complete Attribute Summary Figure 4-2(f)

Dependencies are entered in either the basic FD or MVD form or the EMVD notation using the short attribute identifiers assigned above. The case of the identifiers is quite significant as shown in Fig. 4-3(a).

*a->bc

*** This Attribute has not been defined .. Please reenter

Attribute Identifier's - Case Significance Figure 4-3(a)

All possible error conditions are considered and dealt with as Fig 4-3(b) shows:

```

#A>BC,5
^ *** -> assumed
*** Multiplicity for all FD's is 1.
#A->>>DEF,3
^ *** Too many ">" .. MVD assumed
#G : H : ,6
^ *** No attributes given .. Please reenter
#G : H : ABCDEF,6
*.

```

Dependency Input Error Handling Figure 4-3(b)

A summary of the currently defined dependencies is available by entering a '?'. A summary of the attributes is requested using '??'. When all of the dependencies have been entered the summary shown in Fig. 4-3(c) is presented. All dependencies which are redundant are flagged with a '*'.

```

***** Dependency Resume
1  A -> BC      1
* 2  A -> -> DEF 3
* 3  A -> -> GH   2
4  AD -> EF     1
5  G -> H       1
E1  G : ABCDEF, 6 : H, 1

```

QUERY PROFILE phase

Final Dependency Summary Figure 4-3(c)

Finally we enter the query profile and conclude the description phase as shown in Fig. 4-4.

QUERY PROFILE phase

***** Query Summary
 - Query 1 Attributes
 Customer#
 Customer_Name
 Customer_Address
 Computer_Model
 Quantity_Used

- Query 2 Attributes
 Computer_Model
 Quantity_Used
 Date_Of_Installation
 Technician#

- Query 3 Attributes
 Customer#
 Customer_Name
 Technician#
 Technician_Name

DATA BASE DESCRIPTION COMPLETE

Query Entry and Summary
 Figure 4-4

4.3.3 EVALUATING THE OUTPUT

Immediately after the conclusion of the Input Phase, the user is asked to select whether a copy of the application is to be made and to choose the desired level of output for the subsequent final design phase. The prompt is shown in Fig. 4-5(a).

Do you wish to retain a record of the input ? (Y/N) :

*N

Please select the required output format,

0 - Basic only, 2 - Query Cost details, 4 - Query Cost and Processing Detail

User Prompt - Start of Design Phase
 Figure 4-5(a)

Considering the basic level of '0' the RDD output for this design will be as shown in Fig. 4-5(b).

```

(1) ***** DECOMPOSE  ABCDEFGH 8 RECORDS 30000 8 BYTES/REC 173 Total Size 8.804 M Bytes
      ADEF. ABCGH. Schema_Size = 3.536 MBytes OVERALL COST - 88775. ACC.
(2)      AGH. ABCDEF. Schema_Size = 4.161 MBytes OVERALL COST - 131312. ACC.
      ABCDEFG. GH. Schema_Size = 6.313 MBytes OVERALL COST - 128535. ACC.

***** DECOMPOSE  ADEF 8 RECORDS 19000 8 BYTES/REC 43 Total Size 1.152 M Bytes
(3) ... ATOMIC RELATION ... ADEF

***** DECOMPOSE  ABCGH 8 RECORDS 10000 8 BYTES/REC 140 Total Size 2.385 M Bytes
(4) ADEF. AGH. ABC. Schema_Size = 3.020 MBytes OVERALL COST - 46631. ACC.

***** DECOMPOSE  AGH 8 RECORDS 10000 8 BYTES/REC 65 Total Size 1.135 M Bytes
(5) ADEF. AG. GH. ABC. Schema_Size = 2.197 MBytes OVERALL COST - 41782. ACC.

***** DECOMPOSE  AG 8 RECORDS 10000 8 BYTES/REC 15 Total Size 0.301 M Bytes
(6) ... ATOMIC RELATION ... AG

***** DECOMPOSE  GH 8 RECORDS 100 8 BYTES/REC 55 Total Size 0.010 M Bytes
(7) ... ATOMIC RELATION ... GH

***** DECOMPOSE  ABC 8 RECORDS 5000 8 BYTES/REC 85 Total Size 0.734 M Bytes
      ... ATOMIC RELATION ... ABC

FINAL SCHEMA DESIGN ---

      ADEF. AG. GH. ABC. Schema Size = 2.197 MBytes
RDD SESSION COMPLETE.

```

Example 4.1 - Basic Design Output
Figure 4-5(b)

Using the annotated output shown in Fig. 4-5(b) we can analyze the results of the decomposition as follows: first each call made to the Decomposition Module is highlighted with the heading DECOMPOSE. Flag (1) shows that the initial schema with the attributes ABCDEFGH has an estimated 30,000 records comprising some 8.8 megabytes of data. Below this we see at Flag (2) that only three possible initial schemas formed by decomposing with:

1. $A \twoheadrightarrow DEF$; or
2. $A \twoheadrightarrow GH$; or
3. $G \twoheadrightarrow H$.

maintain all original dependencies.

The least expensive selection is the first resulting in the new schema - ADEF and ABCGH. Decomposing each of these sub-schemes in the next phase we first consider ADEF at Flag (3) which shows that the scheme ADEF is in fact in 4NF and cannot be decomposed any further. Next, at Flag (4) we consider ABCGH and find that it has only one possible decomposition and so the cost estimate is not used. Decomposing ABCGH into AGH and ABC we again consider each of these sub-schemes. Only one possible choice exists for AGH as shown at Flag (5) and the resulting products are both 4NF (Flags (6) and (7)). Returning to ABC we now find that it is also 4NF and so the design has been completed. The final design is summarized in Fig. 4-5(c). Note that we have reduced the overall size of the schema to 2.2 megabytes from 8.8 megabytes and that the final query cost is far cheaper than the cost for previous alternate schema designs.

R1 (CUST#, CUST_NAME, CUST_ADDRESS)
 R2 (CUST#, MODEL, QTY, DATE_OF_INSTALLATION)
 R3 (CUST#, TECH#)
 R4 (TECH#, TECH NAME)

Example 4.1 - Final Schema Design
 Figure 4-5(c)

With a selected output level of '2' we have more detailed information to consider. For the schema GH, AG, ADEF, ABC (shown at Flag (5)) the cost breakdown for the query profile is shown in Fig. 4-6.

ADEF, AG, GH, ABC. Scheme Size = 2.197 MBytes
 Query for
 ABCDE
 TOTAL QUERY COST 29999. ACCESSES

 Query for
 DEFG
 TOTAL QUERY COST 3287. ACCESSES

 Query for
 ABGH
 TOTAL QUERY COST 8495. ACCESSES
 OVERALL COST - 41782. ACC.

Sample Output at Level 2
 Figure 4-6

Finally at the highest output level of '4', we see the breakdown of costs for the single query DEFG, again from the same

schema at Flag (5), in Fig. 4-7. At this level we are presented with the complete query processing strategy and the detailed costs for each step.

```

Query for
  D E F G
----- PROJECT  A D E F INTO    A D E F
With Cost of      0.
----- PROJECT  A G INTO      A G
With Cost of      0.
----- JOIN    A D E F AND    A G INTO D E F G
      External Sort Required
With cost of      3287.
TOTAL QUERY COST  3287. ACCESSES

```

Sample Output at Level 4
Figure 4-7

Not shown in the previous discussion are the requests to the user to supply the estimated domain size of compound roots or the situation that results when none of the choices preserve all of the original dependencies. These are presented in the actual examples in Chapter 5.

4.4 MAIN MODULE

4.4.1 INTRODUCTION

The Main Module is the overall controller for the design aid and performs the following primary functions:

1. to initialize the main data structure;
2. to determine and assign the source of the database description (stored text file or interactively from the terminal);
3. to direct the input of description data;
4. to direct the decomposition; and
5. to record the result and save the application if required.

Access to the main module occurs immediately with execution of the RDD system.

In the next two sections we describe the common environment of the RDD system including the data structure and library routines.

4.4.2 THE RDD DATA STRUCTURE

In this section we describe the major components of the RDD data structure.

4.4.2.1 ATTRIBUTES -

Since we must often perform the boolean operations of union, intersection and difference when considering collections of attributes a bit vector representation is by far the most efficient data structure. The Pascal SET construct was therefore the logical choice for the representation of sets of attributes. This imposes a restriction on the system since the range of allowable set sizes is implementation dependent and must be fixed at compile time. We therefore define:

```
Attributes = 1..Max_Number_of_Attributes  
            { sub range of integer }
```

```
Attribute_Set = SET OF Attributes
```

and set the maximum limit to 52 (well below most implementation limits of 512 elements). This limit will cover most reasonable examples and allows the use of the letters 'A' to 'Z' and 'a' to 'z' as the short attribute identifiers. An array called the `Attribute_Character_Set` is used to cross refer between the attributes and short identifiers.

We use an array of records called the `Attribute_List`, indexed by the integer attribute value, to store the full attribute description as follows:

```

Attribute_List = ARRAY[1..Max_Number_Of_Attributes] OF
    RECORD;
        Name : PACKED_ARRAY[1..Max_Name_Length]
              OF CHAR;
        Domain_Size : INTEGER;
        Attribute_Type : (Char, Number);
        Length : 1..Max_Attribute_Size;
    END

```

Finally to facilitate passing the attribute information between modules we introduce the Attribute_Group, defined as follows:

```

Attribute_Group = RECORD
    Attribute : Attribute_List;
    Attribute_Count :
        1..Max_Number_of_Attributes;
END

```

Pascal SETS support all basic set operations using the notation summarized in Fig. 4-8. For the sample sets A and B we have:

	<u>Set Notation</u>	<u>Pascal Notation</u>	<u>Meaning</u>
1.	$a \in B$	$a \text{ IN } B$	a is an element of B
2.	$A \supset B$	$A \Rightarrow B$	B is contained in A
3.	$A \subseteq B$	$A \leq B$	A is contained in B
4.	$A \cup B$	$A + B$	A union B
5.	$A \cap B$	$A * B$	A intersect B
6.	$A - (A \cap B)$	$A - B$	Set Difference
7.	$A = B$	$A = B$	Set Equality

Pascal Set Operations
Figure 4-8

This Pascal implementation does not support the determination of

the proper subset condition " $A \subset B$ ". Whenever this condition is to be tested we must determine if both $A \subseteq B$ and $A = B$ are valid.

Special procedures to determine the cardinality of a given set (CARD) and to list the set elements (WRITE_SET) were included in Procedures Library.

4.4.2.2 DEPENDENCIES -

There are two representations for dependencies. Basic FD and MVDs of the form:

$X \rightarrow Y$ or $X \twoheadrightarrow Y$, multiplicity

are stored as records defined as follows:

```
Dependencies = RECORD
    Left Side, Right Side : Attribute Set
    Dependency Type       : (FD, MVD)
    Multiplicity          : Integer
END.
```

The collection of all dependencies is then formed as an array of dependencies called the Dependency_List indexed by a sequential integer value simply indicating the order of entry. This is then nested within another record structure (to simplify the passing of parameters) as:

```
Dependency_Group = RECORD
    Dependency_List:
        ARRAY[1..Max_Number_Of_Dependencies]
        OF Dependencies;
    Dependency_Count: 1..Max_Number_Of
        Dependencies;
END
```

EMVDs on the other hand, require more elaborate storage. We use a linked list of stacks, with each stack identified by the left side of the EMVD and the elements in the stack by their right sides. The context for the EMVD is stored in the base of each stack.

Hence, EMVDs of the form

A : B,3 ; C,4 (meaning $A \rightarrow B,3$ and $A \rightarrow C,4$ in $R(ABC)$)

and

F : G,2 ; H,3

would be represented in the data structure as shown in Fig 4-9:

Dependency Processing Data Structure
Fig 4-9

A similar data structure called the `SLOT_RACK` is used to process the dependencies in the Decomposition module. The library procedure `ENTER` is used to add a new dependency to the appropriate stack.

4.4.2.3 THE SCHEMA REPRESENTATION -

The global schema is implemented as a binary decomposition tree as discussed in Chapter 2. Each node of the tree represents a single scheme and uses the common record type:

Scheme : RECORD

```

Scheme_Attributes   : Attribute_Set;
Left_Son, Right_Son : Scheme_Pointer;
Root                : Attribute_Set
Size_Records {Size of the table in records}
Size_Bytes   {Size of a tuple in bytes}
              : Integer;
Total_Size   : Real;
              {Total Size of the table
               plus index,   in bytes}

```

END

4.4.2.4 DATA LISTS -

There are two additional lists of information commonly passed between the modules: the Query List and the Compound Root List. The query list is a linked list of records containing the attributes for each query. Compound roots, the left side of a dependency containing more than one attribute (which are used to estimate the size of a given scheme), are stored in a linked list of records (containing the Left Side attributes and the estimated size) called the Compound Root List.

4.4.3 LIBRARY ROUTINES

Since all modules perform operations on sets and with the system data structure a set of common routines were placed in the Procedures Library for common access. A summary of the significant routines is given in Appendix A.

4.5 INPUT MODULE

The Input Module is responsible for the collection and verification of all database description information. We have incorporated a high degree of 'user friendly' options to make allowance for entry errors or to facilitate the recall of previously entered information for user reference. Calls to the module use the entry;

```
PROCEDURE INPUT PHASE (IN OUT In_File, Out_File:TEXT,
                       IN Output_Level :Integer,
                       OUT Atts       :Attribute_Group;
                       OUT Deps       :Dependency_Group;
                       OUT EMVD_Header :EMVD_Rack_Pointer;
                       OUT Query_Header :Query_List_Pointer)
```

This module is then broken down into three distinct phases namely:

1. Input Attributes;
2. Input Dependencies; and
3. Input Queries

which are simply called as routines, and are described in the following sections. In all cases these routines follow character input driven Finite State Machine (FSM) implementations based on the appropriate input syntax.

4.5.1 PROCEDURE INPUT ATTRIBUTES

```
(IN OUT In_File, Out_File:TEXT; OUT Atts: Attribute_Group)
```

This routine is called to collect the attribute descriptions from the assigned file. It checks the input for duplicate names, invalid characters, missing details and sizes which are outside the assigned limits.

An attribute is formally defined using BNF syntax as:

Attribute Entry = <Attribute Name>,(1) [C | N](2) <LENGTH>,<Domain Size>(3)

We begin the routine by initializing an input line buffer, attribute name buffer and establishing the various character classes for each acceptable symbol. The formal FSM is therefore defined with three states handled by the separate routines named:

1. Handle Case in Attribute {Name}
2. Handle Case in Type;
3. Handle Case in Domain {Size}

which then parse the given input line.

Errors are flagged and reported to the user as they are detected and in some cases corrective assumptions are made and noted. A user help feature to summarize the current attributes is available by typing a '?'. The phase is terminated by entering a '.'.

4.5.2 PROCEDURE INPUT DEPENDENCIES

```
(IN OUT In_File, Out_File : TEXT,
OUT Dependency : Dependency_Group;
OUT EMVD_Header : EMVD_Pointer)
```

This routine parses the description of all types of dependencies and enters the information into the appropriate data structure. It checks the input for such errors as valid form, undefined attributes, invalid characters and missing details. We define a dependency using the following syntax:

1. `Dependency_Entry = <LeftSide>{<Type><RightSide>} ;`
`[:<RightSide>[:<RightSide>]*]}`
2. `<Left Side> = <Attribute ID>+`
`{ one or more of A, B, C... }`
3. `<Type> = -> | ->->`
4. `<Right Side> = <AttributeID>+ [, <Multiplicity>]?`
`{ such as A, 3 or B }`

There are six routines to handle the formal states of this FSM, namely:

1. Handle Case in Left {Side};
2. Handle Case in Dependency;
3. Handle Case in Right {Side};
4. Handle Case in Multiplicity;
5. Handle Case in EMVD Attributes; and
6. Handle Case in EMVD multiplicity

To assist the user, there are two help features available.

To obtain a current list of dependencies the user may enter '!' and to recall the list of attributes the user enters '??'. The phase is also terminated with a '.' to proceed to the INPUT_QUERIES phase.

4.5.3 PROCEDURE INPUT QUERIES

```
(IN_OUT In_File, Out_File : TEXT;
  OUT Query_Header       : Query_List_Pointer)
```

This routine is the last of the input phases and is called to collect the attribute identifiers for each query in the profile. A query is simply given as:

Query = <Attribute_ID> +

We check for duplicate or undefined attributes. The user may recall the attribute details by entering a '?' and again terminates the phase with a '.' returning control to the main module.

4.6 DECOMPOSITION MODULE

The Decomposition Module directly implements the schema design algorithm as shown in Fig 3-9 and is therefore the core of the design aid. It is called recursively to decompose each scheme choosing the least expensive option which retains the original closure or warning the designer of any lost dependency constraints. Calls to the module use the entry:

```

PROCEDURE DECOMPOSE (IN OUT In_File, Out_File : TEXT;
                        IN OUT Root      : Scheme_Pointer ;
                        IN Scheme_To_Be_Decomposed: Scheme_Pointer;
                        IN Output_Level   : Integer;
                        IN Atts           : Attribute_Group;
                        IN Deps           : Dependency_Group;
                        IN EMVD_Header    : EMVD_Rack_Pointer;
                        IN Query_Header   : Query_List_Pointer;
                        IN Compound_Root   : Compound_Root_Pointer).

```

This passes the module a pointer to the root of the schema decomposition tree and to the specific leaf which is the scheme to be decomposed. The remaining parameters are passed so that they can be used by the supporting modules.

4.7 DEPENDENCY REDUNDANCY MODULE

The Redundancy Module implements both the FD and MVD membership algorithms discussed in Chapter 2. We use the module by passing it the list of dependencies which are valid for the scheme of interest and by designating the test dependency. The module then sorts the list selecting those of the same type (FD or MVD) and executes the membership algorithms as discussed. If the dependency is supported by the cover then the module returns TRUE otherwise it returns FALSE. Calls to the module use the entry:

```

FUNCTION REDUNDANCY CHECK(IN OUT In_File, Out_File:TEXT;
                            IN Dependencies : Dependency_Group;
                            IN Test_Dep     : Integer;
                            IN Attributes   : Attribute_Group)
    RETURNS Booleap.

```

4.8 QUERY COST MODULE

The Query Cost Module determines the number of secondary storage accesses needed to recover the query profile attributes from the current schema design. Calls to the module use the entry:

```

PROCEDURE ASSESS QUERY COSTS(IN OUT In_File, Out_File:TEXT;
                             IN Root           : Scheme_Pointer ;
                             IN Dependencies    : Dependency_Group;
                             IN Attributes      : Attribute_List;
                             IN EMVD_Header    : EMVD_Rack_Pointer;
                             IN Query_Header   : Query_List_Pointer;
                             IN Compound_Root_Header :
                                     Compound_Root_List;
                             IN Output_Level    : Integer;
                             OUT Overall_Cost  : Real).

```

The module then considers each query separately by walking the tree collecting the target attributes as discussed in section 2.4.2. Depending on the output level selected by the user, the module reports:

1. Total costs only (at level 0)
2. Cost by Query plus Total Cost (at level 2); and
3. Cost for each operation (at level 4 or higher)

4.9 SCHEME SIZE MODULE

The Scheme Size module estimates the 1NF size of a given set of attributes based on the valid dependencies which hold for the scheme and follows the description given in section 2.3. Calls

to the module use the entry:

```

PROCEDURE EVALUATE TABLE SIZE(IN OUT In_File, Out_File:TEXT;
                                IN OUT Table      : Scheme_Pointer;
                                IN Output_Level  : Integer;
                                IN Attributes    : Attribute_List;
                                IN Dependencies  : Dependency_List;
                                IN EMVD_Header   : EMVD_List_Pointer;
                                IN Compound_Root :
                                                Compound_Root_List)

```

The first action of the module is to determine the currently valid set of dependencies using the library routine Find_Valid_Dependencies. The procedure then checks each possible root considering single attributes first and then the compound roots. If all attributes are not used in the size determination then the user is advised with the message:

```

      '*** 3 unused attributes EMP, PROJ, DEPT'.

```

When this situation occurs, the user must add further dependency information, possibly using an EMVD, to provide the module with a relationship between these attributes and the others in the scheme. At specified levels of output the module provides a message of the scheme size giving the size in tuples, bytes per tuple and total size (table plus index). In all cases the information is returned to the calling module in the appropriate fields of the Table record.

CHAPTER 5

DATABASE DESIGN SESSIONS USING RDD

5.1 INTRODUCTION

In this chapter we present five sample applications of RDD which cover the full range of its capabilities over the class of examples found in the current literature. In each case we provide a statement of the aim of the example, a description of the database with both its dependency structure and ER representation, the sample RDD output and a brief discussion of the final design. We close this chapter with a discussion of the limitations of both the decomposition approach and the schema design aid.

5.2 EXAMPLE 5.1

5.2.1 AIM

This example demonstrates a successful (in the sense that both data and dependency information is fully represented in the final schema) and non-redundant design for a non-trivial example with many possible decompositions.

5.2.2 DESCRIPTION

We consider an office database as shown in the description and ER representation of Fig. 5-1. We have assumed that 5,000 employees work for a company comprised of 10 departments with 100 projects currently underway. The dependency structure indicates that employees work in only one department and take part in all of their department's projects. To test the system's response to multiple candidate keys we have indicated that both EMP# and SIN# uniquely identify each employee. This is a simplified version of Example 3.1.

Example 5.1 - ER Representation
Figure 5-1 (a)

!XER RDD

*** RDD LOAD ROUTINE ***

Do you wish to LINK ? (Y/N) N
 Name of Input_Record File (leave blank if not required) : EXAMPLE.1
 Name of Output_Record File :
 Do you wish to make a copy of this session (Dribble) ? (Y/N) N

R D D vers 7.5

Interactive Input ? (Y/N) :
 *N

ID	ATTRIBUTE NAME	TYPE	SIZE	ATTRIBUTE DOMAIN SIZE	phase
A	Employee#	Number	6	5000	
B	SINH	Number	9	5000	
C	Employee_Name	Char	25	5000	
D	Employee_Address	Char	25	5000	
E	Department#	Number	3	20	
F	Project#	Number	3	100	

DEPENDENCY phase

***** Dependency Resume

1	A -> B	1
* 2	A -> E	1
3	E -> -> F	2
* 4	A -> -> F	2
* 5	B -> C D	1
* 6	B -> -> F	2
* 7	A -> C D	1
8	B -> A	1
* 9	B -> E	1

QUERY PROFILE phase

***** Query Summary
 - Query 1 Attributes
 Employee#
 Employee_Name
 Employee_Address
 Department#

DATA BASE DESCRIPTION COMPLETE

Example 5.1 - Data Base Description
 Figure 5-1 (b)

5.2.3 RESULTS

The basic design output for this application is seen in Fig. 5-2 and yields the following non-redundant 4NF schema:

1. R1(EMP#, EMP_NAME, EMP_ADDRESS);
2. R2(EMP#, SIN#);
3. R3(DEPT#, PROJ#);
4. R4(EMP#, DEPT#).

This is a very effective design for the given data base. In comparison with the Zaniolo approach (Example 3.2) we note that this final design is already logically consolidated. The designer may, in order to totally reduce the schema to its minimum form, choose to merge R1 and R4 although the existing design allows the flexibility to associate an employee with more than one department which may be worth retaining. We see also that, since our standard query emphasizes the use of EMP# as the most frequently used primary key, the final design associates EMP# with the other attributes - SIN# would also be logically correct but more costly. The final design reduces the overall data base size (from 1.235 Mbytes to .744 Mbytes) and results in the lowest total query cost (1784 accesses) considering all of the intermediate schema designs.

Despite the maintenance of representation inherent to the algorithm we have not explicitly retained all of the original

dependencies. The dependency $EMP\# \rightarrow \rightarrow PROJ$ is derivable from the final cover but is not present in the final design. While this is an acceptable design trade-off it has the potential of causing the algorithm to miss some join dependencies. We offer no solution to this problem.

```

***** DECOMPOSE  ABCDEF # RECORDS 10000 # BYTES/REC 71 Total Size 1.235 M Bytes

ACD. AB EF. Scheme_Size = 0.893 MBytes OVERALL COST - 2706. ACC.
EF. ABCDE. Scheme_Size = 0.593 MBytes OVERALL COST - 11200. ACC.
AB. BCDEF. Scheme_Size = 1.285 MBytes OVERALL COST - 14268. ACC.

***** DECOMPOSE  A,C,D # RECORDS 5000 # BYTES/REC 56 Total Size 0.492 M Bytes

... ATOMIC RELATION ... ACD

***** DECOMPOSE  A,B,E F # RECORDS 10000 # BYTES/REC 21 Total Size 0.401 M Bytes

ACD. AB. AE F. Scheme_Size = 0.894 MBytes OVERALL COST - 2413. ACC.
ACD. EF. A,B E. Scheme_Size = 0.668 MBytes OVERALL COST - 3225. ACC.
ACD. AB. B,E F. Scheme_Size = 0.944 MBytes OVERALL COST - 4177. ACC.

***** DECOMPOSE  AB # RECORDS 5000 # BYTES/REC 15 Total Size 0.151 M Bytes

... ATOMIC RELATION ... AB

***** DECOMPOSE  A,E F # RECORDS 10000 # BYTES/REC 12 Total Size 0.251 M Bytes

ACD. AB. EF. AE. Scheme_Size = 0.744 MBytes OVERALL COST - 1784. ACC.

***** DECOMPOSE  EF # RECORDS 40 # BYTES/REC 6 Total Size 0.001 M Bytes

... ATOMIC RELATION ... EF

***** DECOMPOSE  AE # RECORDS 5000 # BYTES/REC 9 Total Size 0.101 M Bytes

... ATOMIC RELATION ... AE

FINAL SCHEMA DESIGN ---
ACD. AB. EF. AE. Scheme Size = 0.744 MBytes

RDD SESSION COMPLETE.

```

Example 5.1 - Sample Design Output
Figure 5-2

5.3 EXAMPLE 5.2

5.3.1 AIM

Expanding upon Example 5.1 we recreate the data base initially presented as Example 3.1 and show an unsuccessful attempt to design a fully representative schema. We also demonstrate the system for an increased number of attributes, dependencies and queries.

5.3.2 DESCRIPTION

The data base description is shown in Fig. 5-3. The ER representation has already been given in Fig. 3-2(a). Only functional dependencies are used to describe the logical correspondences although several multivalued dependencies could have been given. The problem encountered by the algorithm in this example is that in order to prevent a scheme from introducing the type of error seen in Example 3.1 we must lose some of the original dependency information. This potential loss is then brought to the attention of the designer.

!XER ROD

*** R D D LOAD ROUTINE ***

Do you wish to LINK ? (Y/N) N

Name of Input_Record File (leave blank if not required) : EXAMPLE_2

Name of Output_Record File :

Do you wish to make a copy of this session (Dribble) ? (Y/N) N

R D D vers 7.5

Interactive Input ? (Y/N) :

nN

ID	ATTRIBUTE NAME	TYPE	SIZE	ATTRIBUTE DOMAIN SIZE	phase
A	SINH	Number	9	5000	
B	Employee#	Number	6	5000	
C	Employee_Name	Char	25	5000	
D	Department#	Char	10	20	
E	Department_Name	Char	25	20	
F	Location	Char	25	10	
G	Work_Station	Char	5	2500	
H	Salary	Number	5	100	
I	Project	Char	25	100	

DEPENDENCY phase

***** Dependency Resume

1	A -> B C	1
2	B -> A C	1
* 3	B C -> D	1
* 4	A -> D	1
5	B I -> G	1
6	A G -> I	1
7	C D -> H	1
* 8	B -> H	1
9	D -> E F	1

QUERY PROFILE phase

***** Query Summary

- Query 1 Attributes

SINH

Employee#

Employee_Name

Department#

- Query 2 Attributes

Employee#

Employee_Name

Work_Station

Salary

Project

DATA BASE DESCRIPTION COMPLETE

Example 5.2 - Data Base Description
Figure 5-3

5.3.3 RESULTS

The sample output for this application is given in Fig. 5-4. We can see in this example that the algorithm is frequently using dependencies with compound roots and so must ask the user to estimate their domain sizes. Considering EMP# EMP_NAME (BC) with 5,000 employees, each with one name, we therefore estimate 5,000 as the domain size. Similarly if one employee works on an average of two projects then the domain size of BI (or similarly for AG) will be $5,000 \times 2 = 10,000$.

After separating DEF (DEPT#, DEPT_NAME, LOCATION), CDH (EMP_NAME, DEPT#, SALARY) and AD (SIN#, EMP#) the system notifies the user that although the remaining scheme (ABCGI) is not in 4NF it cannot be normalized without the loss of at least one dependency for each possible decomposition. We have the choice of losing the association of either SIN# or EMP# with WORK_STATION and PROJ. Since our query profile stresses the relationship between these attributes and EMP# it would seem desirable for a better design to include this dependency and relinquish the other. This is reflected in the estimated query costs because of the smaller number of accesses required for the second choice. Note, however, that neither choice will cause us to form schemes with the same problem found in the final design for Example 3.1. Making the selection to retain the dependency BI->G leaves us with the final 4NF design:

1. R1(DEPT#, DEPT_NAME, LOCATION);
2. R2(EMP NAME, DEPT#, SALARY);
3. R3(SIN#, DEPT#);
4. R4(SIN#, EMP#, EMP_NAME);
5. R5(EMP#, PROJ, WORK_STATION).

The final design is again substantially less than the original scheme (1.555 MBytes versus 2.301 MBytes), although some of the schema designs in their unnormalized forms had less expensive query costs. Careful inspection will also reveal that the dependency EMP# WORK_STATION->PROJ which is equivalent to the one which was lost in the decomposition can be incorporated into the final schema (leaving R5 with two candidate keys) thereby retaining the original dependency information without introducing any redundancy.


```

***** DECOMPOSE  ABCDEFGHI
!!!! Compound Root - How many unique values of BC do you estimate ?
#5000

!!!! Compound Root - How many unique values of BI do you estimate ?
#10000
# RECORDS 10000 # BYTES/REC 135 Total Size 2.301 M Bytes

DEF, ABCD GHI. Scheme_Size = 1.470 MBytes OVERALL COST - 39474. ACC.

***** DECOMPOSE DEF # RECORDS 20 # BYTES/REC 60 Total Size 0.002 M Bytes

... ATOMIC RELATION ... DEF

***** DECOMPOSE ABCD GHI # RECORDS 10000 # BYTES/REC 85 Total Size 1.468 M Bytes

!!!! Compound Root - How many unique values of CD do you estimate ?
#5000
DEF, CDH, ABCD GI. Scheme_Size = 1.746 MBytes OVERALL COST - 69927. ACC.

***** DECOMPOSE CDH # RECORDS 5000 # BYTES/REC 40 Total Size 0.359 M Bytes

... ATOMIC RELATION ... CDH

***** DECOMPOSE ABCD GI # RECORDS 10000 # BYTES/REC 80 Total Size 1.385 M Bytes

DEF, CDH, AD, ABC GI. Scheme_Size = 1.763 MBytes OVERALL COST - 63387. ACC.

***** DECOMPOSE AD # RECORDS 5000 # BYTES/REC 19 Total Size 0.184 M Bytes

... ATOMIC RELATION ... AD

***** DECOMPOSE ABC GI # RECORDS 10000 # BYTES/REC 70 Total Size 1.218 M Bytes

CANNOT DECOMPOSE .. ABC GI WITHOUT LOSS OF DEPENDENCIES.

!!!! Compound Root - How many unique values of AG do you estimate ?
#10000
DEF, CDH, AD, ABC, AGI. Scheme_Size = 1.605 MBytes OVERALL COST - 64108. ACC.
- CRC Test Failure with dependency BI -> G
- CRC Test Failure with dependency BI -> -> G
DEF, CDH, AD, ABC, BGI. Scheme_Size = 1.555 MBytes OVERALL COST - 64010. ACC.
- CRC Test Failure with dependency AG -> I
- CRC Test Failure with dependency AG -> -> I
Do you wish to further decompose this scheme ? (Y/N)
nY
Please enter choice for the dependency to use next
0 (minimum cost value) or 1 to 2
#0

***** DECOMPOSE ABC # RECORDS 5000 # BYTES/REC 40 Total Size 0.359 M Bytes

... ATOMIC RELATION ... ABC

***** DECOMPOSE BGI # RECORDS 10000 # BYTES/REC 36 Total Size 0.651 M Bytes

... ATOMIC RELATION ... BGI

FINAL SCHEMA DESIGN ---

DEF, CDH, AD, ABC, BGI. Scheme Size = 1.555 MBytes

RDD SESSION COMPLETE.

```

Example 5.2 - Sample Design Output
Figure 5-4

POOR PRINT
Epreuve illisible

5.4 EXAMPLE 5.3

5.4.1 AIM

In this example we increase the size of the database under study and follow a successful design for the major example presented in [ZANI81]. We show that the final product of the RDD approach gives a more cohesive schema design than the original Zaniolo algorithm.

5.4.2 DESCRIPTION

Consider the database for a vehicle registration and traffic control department in a major urban centre with the complete 'elementary' dependency description and ER representation shown in Fig. 5-5. Furthermore we will add extra queries to cover the broad range of expected demands on the database.

We can see in Fig. 5-5 that we have a poor crop of drivers in this town (3 violations per driver) although they are reasonably well off (2 cars per owner). Note that this set of dependencies also implies that all drivers are owners of the vehicles that they drive.

Example 5.3 - ER Representation
Figure 5-5 (a)

'RED RDD

*** RDD LOAD ROUTINE ***

Do you wish to LINK ? (Y/N) N
 Name of Input_Record File (leave blank if not required) : EXAMPLE_3
 Name of Output_Record File :
 Do you wish to make a copy of this session (Debbble) ? (Y/N) N

R D D vers 7.5

Interactive Input ? (Y/N) :
 ON

ID	ATTRIBUTE NAME	TYPE	SIZE	ATTRIBUTE DOMAIN SIZE	Phase
A	Vehicle_Licensed	Number	8	25000	
B	Make	Char	25	4	
C	Model	Char	25	100	
D	Year	Number	8	25	
E	Value	Number	4	1000	
F	Owner_Details	Char	50	25000	
G	Driver_Licensed	Number	8	25000	
H	Violation	Number	8	75000	
I	Date_Of_Violation	Number	8	75000	

DEPENDENCY Phase

***** Dependency Resume

1	A ->	BCDEFG	1
2	A -> ->	H I	3
3	BCD ->	E	1
4	BCD -> ->	AFGHI	6
5	F ->	G	1
6	G -> ->	H I	3
7	F -> ->	H I	3
8	F -> ->	ABCDE	2
9	G ->	F	1
10	G -> ->	ABCDE	2

QUERY PROFILE Phase

***** Query Summary
 - Query 1 Attributes
 Vehicle_Licensed
 Owner_Details
 Driver_Licensed

- Query 2 Attributes
 Make
 Model
 Year
 Value

- Query 3 Attributes
 Vehicle_Licensed
 Make
 Model
 Year

- Query 4 Attributes
 Owner_Details
 Driver_Licensed
 Violation
 Date_Of_Violation

DATA BASE DESCRIPTION COMPLETE

Example 5.3 - Data Base Description
 Figure 5-5 (b)

POOR PRINT
 Epreuve illisible

5.4.3 RESULTS

The design output for this example is given in Fig. 5-6 and yields the following non-redundant 4NF schema:

1. R1(VEHICLE LICENSE, DRIVER_LICENSE, MAKE, MODEL, YEAR);
2. R2(MAKE, MODEL, YEAR, VALUE);
3. R3(DRIVER LICENSE, VIOLATION#, DATE); and
4. R4(OWNER DETAILS, DRIVER LICENSE).

Again we have a sizeable reduction in the overall space required for the data base as well as the total number of query accesses. In contrast, this same example in [ZANI81] results in a similar schema but with 7 relations including primitive schemes such as (VEHICLE LICENSE, MAKE), (VEHICLE LICENSE, MODEL) and (VEHICLE LICENSE, YEAR). Clearly the design above is much more useful and efficient as a final product. It does have some significant limitations based on the set of initial dependencies. For example we have no way of recording the vehicle information for a violation in which the driver was not the owner of the vehicle. This demonstrates that elementary dependencies do not necessarily provide a complete logical picture of the database. Note also that while the union of dependency covers of the final schema maintain the original closure, again, not all dependencies are explicitly retained. For example we have lost the relationship between Owner_Details and Violation# as well as the one between Owner_Details and the Make, Model and Year of his

vehicle which can only be found by using his Driver_License# in exactly the same manner as SIN# and EMP# in Example 5.2 .

```

***** DECOMPOSE  ABCDEFGHI # RECORDS  75000 # BYTES/REC  146 Total Size  18.634 M Bytes

AHI, ABCDEFG, Scheme_Size =  8.929 MBytes
!!!! Compound Root - How many unique values of BCD do you estimate ?
$1000
OVERALL COST - 348160. ACC.
ABCD, FGHI, BCDE, Scheme_Size =  17.996 MBytes OVERALL COST - 529671. ACC.
ABCDEF, FGHI, Scheme_Size =  14.845 MBytes OVERALL COST - 202015. ACC.
ABCDEG, FGHI, Scheme_Size =  12.095 MBytes OVERALL COST - 140445. ACC.

***** DECOMPOSE  ABCDEG # RECORDS  25000 # BYTES/REC  80 Total Size  3.461 M Bytes

ABCDG, BCDE, FGHI, Scheme_Size =  12.957 MBytes OVERALL COST - 131082. ACC.

***** DECOMPOSE  ABCDG # RECORDS  25000 # BYTES/REC  74 Total Size  3.211 M Bytes

... ATOMIC RELATION ... ABCDG

***** DECOMPOSE  BCDE # RECORDS  1000 # BYTES/REC  64 Total Size  0.112 M Bytes

... ATOMIC RELATION ... BCDE

***** DECOMPOSE  FGHI # RECORDS  75000 # BYTES/REC  74 Total Size  9.634 M Bytes

ABCDG, BCDE, FG, FHI, Scheme_Size =  14.502 MBytes OVERALL COST - 142606. ACC.
ABCDG, BCDE, GHI, FG, Scheme_Size =  9.252 MBytes OVERALL COST - 132352. ACC.
ABCDG, BCDE, FG, GHI, Scheme_Size =  9.252 MBytes OVERALL COST - 132352. ACC.

***** DECOMPOSE  GHI # RECORDS  75000 # BYTES/REC  24 Total Size  3.384 M Bytes

... ATOMIC RELATION ... GHI

***** DECOMPOSE  FG # RECORDS  25000 # BYTES/REC  58 Total Size  2.545 M Bytes

... ATOMIC RELATION ... FG

FINAL SCHEMA DESIGN ---
ABCDG, BCDE, GHI, FG, Scheme Size =  9.252 MBytes

RDD SESSION COMPLETE.

```

Example 5.3 - Sample Design Output
Figure 5-6

POOR PRINT
Epreuve illisible

5.5 EXAMPLE 5.4

5.5.1 AIM

This example is somewhat of a classic in the literature having been introduced in the original paper by [FAGI77] as well as [DELO78] and [LIEN81] and was also used to discuss the alternative query processing strategies in Chapter 3. The queries are selected specifically to ensure that the attributes are gathered from across the schema and not simply from one constituent scheme. We need to introduce several EMVD's to describe the association of certain attributes in order for the size estimator algorithm to perform correctly.

5.5.2 DESCRIPTION

The database description is given as Fig. 5-7. Perhaps the only attributes requiring further description are MAJOR (say Computer Science or Mathematics) and YEAR_INDICATOR (such as First Year, Qualifying Year and so on). The EMVD's are also fairly straightforward. Consider

$$C : GHI, 8 \mid DF, 1$$

which indicates that a Student has, on average, eight instructors and one Major which can be modelled using MVD's only in the context CDFGHI. We cannot introduce the simple MVD

C $\rightarrow$ GHI , 8 for the entire relation because we will induce the "lossy join" errors previously discussed. Since we need to know this association for the size estimator algorithm, the EMVD can be used to speed up the design process by avoiding frequent interaction with the user. The selection of this extra information follows from the analysis of the 'unused attribute' error messages during previous design sessions.

Example 5.4 - ER Representation
Figure 5-7(a)

USER ADD

*** RDB LOAD ROUTINE ***

Do you wish to LINK ? (Y/N) N
 Name of Input_Record File (leave blank if not required) : EXAMPLE.4
 Name of Output_Record File :
 Do you wish to make a copy of this session (Dribble) ? (Y/N) N

RDB vers 7.5

Interactive Input ? (Y/N) :
 on

ID	ATTRIBUTE NAME	TYPE	SIZE	ATTRIBUTE DOMAIN SIZE	Phase
A	Class	Char	5	10	
B	Section	Char	1	30	
C	Student	Char	25	300	
D	Major	Char	10	30	
E	Examinations	Number	2	100	
F	Year_Indicator	Char	5	4	
G	Instructor	Char	25	60	
H	Rank	Char	10	5	
I	Salary	Number	5	15	
J	Text	Char	25	20	
K	Room	Char	10	30	
L	Day	Char	3	5	

DEPENDENCY Phase

```

***** Dependency Resume
# 1 AB->-> CDEF      150
# 2 ABC->-> E        5
# 3 A->-> J          2
# 4 AB->-> GHI       2
# 5 AB->-> KL        3
# 6 ABL->-> K        2
# 7 C->-> DF         1
# 8 G->-> HI         1
E1 C : GHI, 8 : DF, 1
E2 C : E, 40 : DF, 1
E3 C : AB, 8 : DF, 1
E4 A : BC, 90 : J, 1
E5 AB : E, 450

```

QUERY PROFILE Phase

```

***** Query Summary
- Query 1 Attributes
Class
Section
Student

- Query 2 Attributes
Student
Major
Instructor

- Query 3 Attributes
Class
Section
Instructor

- Query 4 Attributes
Student
Major
Examinations

```

DATA BASE DESCRIPTION COMPLETE

Example 5.4 - Data Base Description
 Figure 5-7(b)

5.5.3 RESULTS

The sample output is given as Fig. 5-8. The resulting final design is given below:

1. R1(CLASS,TEXT);
2. R2(CLASS,SECTION,DAY,ROOM);
3. R3(STUDENT,MAJOR,YEAR_INDICATOR);
4. R4(CLASS,SECTION,STUDENT,EXAMS);
5. R5(INSTRUCTOR,RANK,SALARY);
6. R6(CLASS,SECTION,INSTRUCTOR).

Overall this is a very efficient design, matching the best solutions offered by similar algorithms in the literature. The action of the algorithm in selecting a non-redundant and reasonable design from the many possible choices available is quite evident.

```

***** DECOMPOSE  ABCDEFGHIJKL
!!!! Compound Root - How many unique values of AB do you estimate?
#30
# RECORDS 54000 # BYTES/REC 126 Total Size 11.616 M Bytes

AJ, ABCDEFGHIKL, Scheme_Size = 4.684 MBytes OVERALL COST - 57680, ACC.
CDF, ABCEGHIJKL, Scheme_Size = 10.288 MBytes OVERALL COST - 98629, ACC.
GHI, ABCDEFGJKL, Scheme_Size = 10.271 MBytes OVERALL COST - 101299, ACC.

***** DECOMPOSE  AJ # RECORDS 20 # BYTES/REC 30 Total Size 0.001 M Bytes

... ATOMIC RELATION ... AJ

***** DECOMPOSE  ABCDEFGHIKL # RECORDS 27000 # BYTES/REC 101 Total Size 4.683 M Bytes

AJ, ABCDEF, ABGHIKL, Scheme_Size = 0.403 MBytes OVERALL COST - 26513, ACC.
AJ, ABGHI, ABCDEFKL, Scheme_Size = 1.448 MBytes OVERALL COST - 32662, ACC.
AJ, ABKL, ABCDEFGHI, Scheme_Size = 1.370 MBytes OVERALL COST - 31764, ACC.
AJ, CDF, ABCEGHIKL, Scheme_Size = 4.031 MBytes OVERALL COST - 49737, ACC.
AJ, GHI, ABCDEFGKL, Scheme_Size = 4.014 MBytes OVERALL COST - 52406, ACC.

***** DECOMPOSE  ABCDEF # RECORDS 4500 # BYTES/REC 48 Total Size 0.383 M Bytes

AJ, CDF, ABCE, ABGHIKL, Scheme_Size = 0.312 MBytes OVERALL COST - 21454, ACC.

***** DECOMPOSE  CDF # RECORDS 300 # BYTES/REC 40 Total Size 0.022 M Bytes

... ATOMIC RELATION ... CDF

***** DECOMPOSE  ABCE # RECORDS 4500 # BYTES/REC 33 Total Size 0.271 M Bytes

... ATOMIC RELATION ... ABCE

***** DECOMPOSE  ABGHIKL # RECORDS 180 # BYTES/REC 59 Total Size 0.019 M Bytes

AJ, CDF, ABCE, ABKL, ABGHI, Scheme_Size = 0.301 MBytes OVERALL COST - 21401, ACC.
AJ, CDF, ABCE, ABGHI, ABKL, Scheme_Size = 0.301 MBytes OVERALL COST - 21401, ACC.
AJ, CDF, ABCE, GHI, ABGKL, Scheme_Size = 0.312 MBytes OVERALL COST - 17422, ACC.

***** DECOMPOSE  GHI # RECORDS 40 # BYTES/REC 40 Total Size 0.004 M Bytes

... ATOMIC RELATION ... GHI

***** DECOMPOSE  ABGKL # RECORDS 180 # BYTES/REC 44 Total Size 0.014 M Bytes

AJ, CDF, ABCE, GHI, ABKL, ABG, Scheme_Size = 0.304 MBytes OVERALL COST - 21373, ACC.

AJ, CDF, ABCE, GHI, ABG, ABKL, Scheme_Size = 0.304 MBytes OVERALL COST - 21373, ACC.

***** DECOMPOSE  ABKL # RECORDS 90 # BYTES/REC 19 Total Size 0.003 M Bytes

... ATOMIC RELATION ... ABKL

***** DECOMPOSE  ABG # RECORDS 60 # BYTES/REC 31 Total Size 0.003 M Bytes

... ATOMIC RELATION ... ABG

FINAL SCHEMA DESIGN ---
AJ, CDF, ABCE, GHI, ABKL, ABG, Scheme Size = 0.304 MBytes

RDD SESSION COMPLETE.

```

Example 5.4 - Sample Design Output
Figure 5-8

5.6 EXAMPLE 5.5

5.6.1 AIM

We present an example database given in [ULL80] which cannot be decomposed without the loss of some dependency information, to show the usefulness of the cost estimate in choosing between possible designs. This example demonstrates that RDD is more capable of resolving a class of problem not handled by other algorithms.

5.6.2 DESCRIPTION

The description is given in Fig. 5-9 and we refer the reader to the ER representation given in Fig. 5-7(a). A small number of EMVD's are again used to add information for size estimation purposes. The compound roots reflect the associations given in Example 5.4 - such as 500 students each with 10 classes (hence the domain for AE is 5,000) or 100 classes each with 2 rooms (CD = 200).

!XEQ RDD

*** RDD LOAD ROUTINE ***

Do you wish to LINK ? (Y/N) N

Name of Input_Record File (leave blank if not required) : EXAMPLE.5

Name of Output_Record File :

Do you wish to make a copy of this session (Dribble) ? (Y/N) N

R D D vers 7.5

Interactive Input ? (Y/N) :

nN

ID	ATTRIBUTE NAME	TYPE	SIZE	ATTRIBUTE DOMAIN SIZE	phase
A	Course	Char	5	100	
B	Teacher	Char	25	125	
C	Hour	Number	4	24	
D	Room	Number	4	50	
E	Student	Char	25	500	
F	Grade	Number	2	100	

DEPENDENCY				phase
1	A	→	B	1
2	C	→	A	1
3	B	→	D	1
4	A	→	F	1
5	C	→	D	1
E1	A	:	EF, 30	BC, 3
E2	CD	:	B, 1	EF, 30

***** Dependency Resume

1 A → B 1
 2 C → A 1
 3 B → D 1
 4 A → F 1
 5 C → D 1
 E1 A : EF, 30 : BC, 3
 E2 CD : B, 1 : EF, 30

QUERY PROFILE phase

***** Query Summary

- Query 1 Attributes

Course

Hour

Room

- Query 2 Attributes

Course

Student

Grade

- Query 3 Attributes

Course

Teacher

DATA BASE DESCRIPTION COMPLETE

Example 5.5 - Data Base Description
 Figure 5-9

5.6.3 RESULTS

The sample design output is shown in Fig. 5-10. Following the initial establishment of the size of the universal relation RDD reports that none of the decomposition choices are successful and lists the possible options available. We see that the cost estimate accurately reflects the importance of the lost constraints with respect to the given query profile. Hence in this example the loss of BC $\rightarrow$ D (its MVD counterpart BC $\twoheadrightarrow$ D is also not supported as indicated) represents the best choice. Selecting on this basis we continue the decomposition finally resulting in the efficient 4NF schema:

1. R1(COURSE, TEACHER);
2. R2(COURSE, STUDENT, GRADE);
3. R3(COURSE, HOOR, ROOM); and
4. R4(HOOR, ROOM, STUDENT).

Again, should the designer choose to induce some redundancy into the design, the lost dependency may be re-introduced. The important thing to note is that this approach recognizes and reports this loss of information to the user.

```

***** DECOMPOSE  ABCDEF
!!!! Compound Root - How many unique values of  CD do you estimate ?
#200

!!!! Compound Root - How many unique values of  BC do you estimate ?
#500

!!!! Compound Root - How many unique values of  AE do you estimate ?
#3000

!!!! Compound Root - How many unique values of  CE do you estimate ?
#900

# RECORDS  900 # BYTES/REC  65 Total Size  0.102 M Bytes

CANNOT DECOMPOSE .. ABCDEF WITHOUT LOSS OF DEPENDENCIES.
AB, ACDEF. Scheme_Size = 0.070 Mbytes OVERALL COST - 2934, ACC.
-- CRC Test Failure with dependency BC -> D
-- CRC Test Failure with dependency BC -> -> D
ACD, BCDEF. Scheme_Size = 0.636 Mbytes OVERALL COST - 47093, ACC.
-- CRC Test Failure with dependency A -> B
-- CRC Test Failure with dependency AE -> F
-- CRC Test Failure with dependency A -> -> B
-- CRC Test Failure with dependency AE -> -> F
BCD, ABCEF. Scheme_Size = 0.973 Mbytes OVERALL COST - 6403, ACC.
-- CRC Test Failure with dependency CD -> A
-- CRC Test Failure with dependency CE -> B
-- CRC Test Failure with dependency CD -> -> A
-- CRC Test Failure with dependency CE -> -> B
CDE, ABCEF. Scheme_Size = 1.322 Mbytes OVERALL COST - 10796, ACC.
-- CRC Test Failure with dependency CD -> A
-- CRC Test Failure with dependency BC -> B
-- CRC Test Failure with dependency CD -> -> A
-- CRC Test Failure with dependency BC -> -> D
Do you wish to further decompose this scheme? (Y/N)
Y
Please enter choice for the dependency to use next
0 (minimum cost value) or 1 to 4
00

```

```

***** DECOMPOSE  AB # RECORDS 100 # BYTES/REC 30 Total Size  0.036 M Bytes
... ATOMIC RELATION ... AB

***** DECOMPOSE  ACDEF # RECORDS 900 # BYTES/REC 40 Total Size  0.065 M Bytes
AB, AEF, ACDE. Scheme_Size = 0.242 Mbytes OVERALL COST - 127, ACC.

***** DECOMPOSE  AEF # RECORDS 3000 # BYTES/REC 22 Total Size  0.175 M Bytes
... ATOMIC RELATION ... AEF

***** DECOMPOSE  ACDE # RECORDS 900 # BYTES/REC 38 Total Size  0.062 M Bytes
AB, AEF, ACD, CDE. Scheme_Size = 0.547 Mbytes OVERALL COST - 0, ACC.

***** DECOMPOSE  ACD # RECORDS 200 # BYTES/REC 12 Total Size  0.005 M Bytes
... ATOMIC RELATION ... ACD

***** DECOMPOSE  CDE # RECORDS 6000 # BYTES/REC 33 Total Size  0.361 M Bytes
... ATOMIC RELATION ... CDE

FINAL SCHEMA DESIGN ---
AB, AEF, ACD, CDE. Scheme Size = 0.547 Mbytes
RDB SESSION COMPLETE.

```

Example 5.5 - Sample Design Output
Figure 5-10

5.7 LIMITATIONS OF THE DECOMPOSITION APPROACH

In the last section it was shown that the decomposition approach can be applied algorithmically for non-trivial examples provided that the universal relation and uniqueness assumptions hold and that the inherent semantic constraints can be modelled using either functional or multivalued dependencies. Unfortunately, extending these examples to include significantly more information causes problems which force us to recognize the artificiality of our original assumptions. We shall use the following example for illustration:

EXAMPLE 5.6 Consider the database requirements for an airforce base on which several flying squadrons operate different types (fleets) of aircraft. Let us assume that current organization policy dictates that the airbase is responsible for the maintenance of the aircraft fleets using separate maintenance units for each type. We must therefore record information related to the number of each type of aircraft, their performance records, their constituent component histories and flight data. For each flying squadron and maintenance unit we must also record data such as personnel details, qualifications and daily activities (flying or repairing). Other typical relationship information to be recorded includes such details as which aircraft are currently assigned to which squadron, which aircrew flew which missions and the date, duration of job

and technicians involved in the maintenance actions for a specific aircraft. We present a small portion of the ER representation for this database in Fig. 5-11.

Example 5.6 - A Realistic Database Problem
Figure 5-11.

The first problem is to model the various correspondences using either functional or multivalued dependencies. Functional dependencies, as always, are the least difficult and we can immediately cite the FD's `Maintenance_Unit_Name->Aircraft_Fleet` or `Squadron_No->Aircraft_Fleet` - meaning that only one type of aircraft is either fixed or flown by each maintenance unit or flying squadron. The multivalued dependency is simply not adequate to properly represent the majority of the remaining correspondences. For example, although each airbase has several fleets of aircraft this cannot be expressed as the MVD `Airbase->->Fleet` because then the maintenance units will be associated through the airbase (using the complementation property) with all types of aircraft flown from the Base - which is contrary to the FD defined above. The 'cliques' of entities formed by the numerous relationships which are commonly found in most realistic database structures severely restrict possible uses of the MVD.

We now examine the universal relation and uniqueness assumptions. Consider the relationships of Fleet, Squadron and Aircrew. We have defined that Squadrons fly only one type of aircraft and so it is reasonable to assume that all Aircrew on each Squadron are qualified on the type of aircraft flown by the Squadron. Realistically though we would like to maintain a record of all aircraft types that the Aircrew are qualified on - not just the one that they are currently flying. In order to

incorporate this reasonable criteria we violate the uniqueness assumption in that the transitive correspondence of the Aircrew through their Squadron to the type of aircraft flown by the Squadron will not yield the same tuples as the direct relationship between Aircrew and Fleet. Considering other similar situations within this database it is clear that we cannot form a universal relation and therefore the decomposition method is logically if not practically incapable of producing a reasonable design.

We must therefore conclude that the decomposition method is rigidly bound in the scope of its application and the tools (such as the FD or MVD) which it can use. The creativity of the designer is still an essential part of the design process.

CHAPTER 6

CONCLUSIONS AND SUGGESTIONS
FOR FURTHER RESEARCH

6.1 THESIS CONTRIBUTIONS

The major contributions of this work can be summarized as follows:

1. the development of an original and improved schema decomposition algorithm which can handle a broader class of database structures than those in the current literature;
2. the development of an improved algorithm for the determination of the multivalued dependency basis; and
3. the specification and implementation of an automated schema design aid which embodies the schema decomposition algorithm and provides realistic and useable designs for a range of examples which fully represent those presented in the current literature.

6.1.1 SCHEMA DESIGN ALGORITHM

In this work we have developed an improved conceptual schema design algorithm, based on the decomposition method, which can be used to effectively handle a broad class of dependency structures. We have discussed the theoretical basis for semantic data dependencies as a tool for the logical reduction of relational schemes to a series of normal forms. In so doing we defined that a 'good' schema design is one which represents our original database considering both the data and the dependencies in a practical manner.

In Chapter 2 we then examined a series of fundamental algorithms which we used as the basic tools of the design aid. We first described the membership algorithms for both functional and multivalued dependencies and their uses in practical situations. We then assumed a database model in which the relationships between the attributes were uniform and developed an algorithm to estimate the 1NF size of any given scheme based on the valid dependencies and domain sizes. The Scheme Size algorithm performed very well in databases with many dependencies linking the attributes. For some examples in Chapter 5, however, it was necessary to introduce more specific EMVD's when the MVD was not capable of expressing the correspondences needed for some attribute combinations. This algorithm is clearly limited when used with database structures which are not 'well connected' in

this sense. Lastly, we developed both the model of a practical relational database implementation and a query processing strategy based on the decomposition tree approach which we used to estimate the costs in secondary storage costs to respond to the anticipated query profile.

We then examined three currently proposed design algorithms considering their approach in terms of complexity, dependency capabilities and practical results for a series of examples in Chapter 3 and concluded that a compromise approach was necessary. Using the examples presented in Chapter 5, the improved algorithm was then shown to be a better and more practical approach. The algorithm consistently gave realistic final designs over the full range of examples, avoiding poor design choices and overnormalized results. Furthermore, we showed that the query cost estimator was effective in guiding the algorithm in the selection of reasonable schema designs when confronted with schemes which could not be decomposed without the loss of dependency information. Considering that many dependency structures do not have fully representative schema designs this information can be used to cope with situations beyond the current limits of normalization theory.

To put the decomposition approach into perspective we then examined a realistic database example. We concluded that the approach is very limited in its application because of the nature

of the multivalued dependency and the assumption of the universal relation. Within these practical limits, however, we have shown that the design aid does perform an effective and valid role in the design process.

6.1.2 DEPENDENCY BASIS ALGORITHM

We have demonstrated and proved an algorithm of less time complexity than the one proposed in [BEER80] for the derivation of the multivalued dependency basis. As the examples in Chapter 2 have demonstrated, our binary tree approach is also conceptually far simpler than the original. As a result, a designer can use the algorithm to compute a dependency basis with pen and paper in short order. This capability is extremely valuable when confirming the results of the design aid.

6.1.3 IMPLEMENTATION OF A SCHEMA DESIGN AID

In this work we have also detailed the functional specification for an implementation of a schema design aid using structured programming techniques. The Relational Database Design Aid (RDD) was then described, in detail, including the particulars of the principal routines, giving the reader an appreciation of both the control and data flow of the system. An attempt was made throughout the development of RDD to use dynamic

data structures whenever possible to allow it to accept large database descriptions while remaining within approximately 128 KBytes of CPU memory. It has been successfully tested with up to 30 attributes and 25 dependencies.

We have acknowledged that the lack of features to modify the database description (for example, to add new attributes or dependencies or adjust multiplicity factors) limits RDD to the single treatment of one application. The modular nature of the input routines and data structure would facilitate these improvements although they were not implemented. A more graphic form of output would also improve the final product.

The feasibility of a commercial database design aid is clearly worth considering despite the limitations of current normalization theory. Given the improvements suggested above, such a tool could be of great value when used in the support of a relational database implementation. We suggest two possible uses. First, it could interpret the assignment of indices as the declaration of semantic dependencies in order to examine the overall database structure and assess the quality of the existing design. Alternatively, in the same manner as in this work, the design aid could be used to create the initial schema.

~

6.2 SUGGESTIONS FOR FURTHER RESEARCH

This research has revealed several areas for further research. We make the following suggestions:

1. The method of maintaining dependency representation embodied in the decomposition algorithm does not always detect the existence of join dependencies due to the complementation property of multivalued dependencies (MVD). An algorithm, given in [AH079], can be used to determine whether a lossless join can be found for any given set of attributes and dependencies with time complexity $O(n^4)$ where n is the number of attributes. Extending the existing decomposition algorithm to test for this lossless condition would ensure that all cases of join dependencies would be handled.
2. The MVD is not easily applied to model the one-to-one or one-to-many correspondences found in common situations without 'lossy join' errors. This suggests that a new — general form dependency (GD) should be pursued.
3. Since a main advantage of a relational database implementation is flexibility, the idea of a fixed conceptual schema design is perhaps inefficient. This suggests the development of an on-line monitor to study

the effectiveness of the current schema and to make recommendations for new and more efficient schema configurations, given actual query profile data (perhaps covering some of the proposals outlined in ([BUNE79] [SCHK79] or [YAO79])).

4. Various alternate conceptual and physical schema storage models have recently been introduced for relational implementations ([CODD79] [DELO78] or [HAMM79]) which propose the use of pre-joined relations, attribute partitioning and physical link pointers. Further schema design algorithms should consider these options and their query profile performance as well.
5. Such alternate schema storage methods will make normalization, in the sense described herein, more difficult to achieve and will increase the need for on-line semantic integrity checks. It follows that the formulation of a system design methodology to cope with semantic integrity in relational implementations would be of great help in these circumstances.

6. A free form query processing strategy (that is, excluding the source relations - the FROM clause in SEQUEL) has significant potential to improve both the ease of use and optimization potential of existing relational implementations. Such a feature would have the added benefit of disallowing any possible 'lossy joins' which the user might inadvertently use in the formulation of complex queries.

APPENDIX A

A SUMMARY OF SIGNIFICANT MODULE ROUTINES

A.1 PROCEDURES LIBRARY

FUNCTION CARD (IN SET : Attribute_Set) RETURNS

Cardinality:INTEGER

Returns the cardinality of any set of attributes.

PROCEDURE WRITE SET (IN OUT FILE:TEXT; IN SET:Attribute_Set)

Writes the short identifiers (A, B, C...) for each element of the set to the appropriate file.

PROCEDURE WRITE DEPENDENCY (IN D:Dependencies;
IN OUT FILE:TEXT);

Writes a dependency in the form X->->Y, M to the appropriate file.

PROCEDURE WRITE EMVD (IN EMVD_Header : EMVD_Rack_Pointer;
IN OUT FILE:TEXT);

3 3
OF / DE

Writes an EMVD of the form A: B,3 | C,4 | D,5 ... to the appropriate file.

PROCEDURE ENTER (IN OUT RACK:SLOT_RACK, IN D:Dependencies)

Checks for a duplicate entry and if none exist, adds the dependency to its appropriate slot (stack).

PROCEDURE FIND VALID DEPENDENCIES (IN SET;Attribute_Set,
IN EMVD_Header
: EMVD_RACK_Pointer;
IN Original_Set
: Dependency_Group;
OUT Current_Set
: Dependency_Group).

Any set of attributes A will have a specific set of valid dependencies considering both the projection of the original dependency set entered in the database description and EMVDs which are valid in the context of the set A. This procedure determines the valid set and its size and returns it to the calling routine.

PROCEDURE WRAPUP (IN Attribute:Attribute_List;
IN Dependency:Dependency_List;
IN EMVD_Header:EMVD_Rack_Pointer,
IN Query_Header:Query_Pointer;
IN OUT FILE:TEXT)

Writes a summary of the database description to the appropriate file for future reuse.

A.2 INPUT MODULE

PROCEDURE DEPENDENCY RESUME (IN OUT File:TEXT
IN Dependency : Dependency_Group,
IN EMVD_Header : EMVD_Rack_Pointer)

This procedure summarizes the current dependency information entered so far. For both FD and MVDs it advises the user whether each dependency is considered redundant. (using the Redundancy Module) by marking redundant dependencies with a '1'.

A.3 SCHEME DECOMPOSITION MODULE

PROCEDURE DETERMINE (IN Valid_Dependencies : Dependency_Group;
IN Scheme_Attributes : Attribute_Set;
IN Step_No: INTEGER;
OUT F, GM, DEC: Dependency_Group)

This procedure sorts the dependencies, valid for the given set of scheme attributes, into two groups: the functional group F and the multivalued group GM. When called from step 2 of the algorithm, the multivalued group is partially ordered and returned as the decomposing dependency group DEC.

Each of the dependencies is processed sequentially using the Slot Rack for storage and to check for duplicates. To

support the MVD relatability test we must include all of the primitive MVD counterparts for FDs with more than one right side attribute. Hence, given the FD:

$a \twoheadrightarrow bcd$

GM will contain $a \twoheadrightarrow b$, $a \twoheadrightarrow c$, and $a \twoheadrightarrow d$. To avoid the overnormalized designs seen in the Zaniolo approach we do not consider these primitive MVD's for possible decomposition. The user is unaware of these primitive MVD's as well.

PROCEDURE REDUCE TO MINIMAL COVER (IN Dec,
OUT Min_Dec: Dependency_Group)

This procedure simply checks each member of DEC for redundancy against the remaining dependencies and discards those that are redundant returning the minimal cover MIN_DEC.

FUNCTION FOUR NF STATUS (IN Scheme_Attributes : Attribute_Set;
IN F, DEC : Dependency_Group) :
RETURNS Boolean.

This function returns TRUE if the scheme is in 4NF given the valid set of dependencies F and DEC such that all non-trivial MVDs $X \twoheadrightarrow Y$ in DEC are covered by FDs $X \rightarrow Y$ in F.

PROCEDURE DIVIDE SCHEMA (IN In_Scheme: Scheme;
OUT SubScheme 1, SubScheme 2 : Scheme;
IN Dividing_Dependency : Dependency)

This procedure uses the dividing dependency $A \twoheadrightarrow B$ to split the scheme (ABC) into (AB) and (AC) creating two new scheme records for the decomposition tree.

PROCEDURE MERGE(IN OUT Group_1,
 IN Group_2: Dependency_Group);

This procedure merges Group 2 into Group 1 and returns the new Group 1.

FUNCTION CRC TEST (IN Original_Cover, SubScheme_Cover
 : Dependency_Group) RETURNS Boolean.

This function tests whether the original cover (either FD or MVD) is still supported by testing each member of the original cover for membership in the closure of the union of the sub-scheme covers. It returns TRUE if the closures are identical.

PROCEDURE REPORT ERROR(IN OUT In_File, Out_File: TEXT
 OUT Go_Ahead: Boolean;
 OUT Choice: INTEGER)

When none of the dependencies in DEC maintain the original cover through decomposition, this procedure is called to advise the user of the problem. The user is shown all non-trivial decompositions highlighting both the dependencies not supported by the design and the appropriate query cost. He is asked whether to continue with the decomposition (Go_Ahead : = TRUE or FALSE) and his choice

of which dependency to use ('0' for the cheapest or the number of a particular dependency).

A.4 DEPENDENCY REDUNDANCY MODULE

```
PROCEDURE SELECT (IN Object_Type: (FD, MVD);
                   IN Dependencies,
                   OUT Common_Set : Dependency_Group)
```

This procedure returns the set of all dependencies chosen from the currently valid set which are of the object type (with the test dependency in the last position).

```
FUNCTION FD CHECK(IN Common_Set : Dependency_Group)
                RETURNS Boolean
```

Using the FD membership algorithm as shown in Fig 2-2 this function determines if the test FD (always last) can be derived from the remaining dependencies and returns TRUE if so. The supporting procedure PICK is used to arbitrarily select an attribute from the set To_Be_Checked.

```
FUNCTION MVD CHECK(IN Common_Set : Dependency_Group)
                RETURNS Boolean.
```

Using the improved basis algorithm as shown in Fig 2-6, this function determines if the test MVD is a member of the closure of the remaining dependencies and returns TRUE if so. The supporting procedures SEEK and SEPARATE are used exactly as discussed in Chapter 2.

A.5 QUERY COST MODULE

PROCEDURE ASSESS COST FOR QUERY (IN Root: Scheme_Pointer ;
IN Target_Set : Attribute_Set;
OUT Subproduct: Scheme;
OUT Cost_For_Query : Real)

The procedure Assess_Query_Costs calls this procedure to determine the cost of each individual query.

FUNCTION ASSESS SITUATION (IN Node: Scheme;
IN Target_Set: Attribute_Set)
 RETURNS situation

At each node in the decomposition tree the algorithm must act on the disposition of the attributes of its target set. This function determines the disposition and returns:

1. Leaf; or
2. Left Heavy, Right Heavy if the target set is fully contained in the left or right children; or
3. Split if some of the attributes are on both sides;
or
4. Both if the target set is contained in both children.

PROCEDURE JOIN PROJECT (IN Scheme 1, Scheme 2: Scheme_Pointer;
OUT Final_Scheme : Scheme_Pointer;
IN Target_Set: Attribute_Set;
OUT Cost: Real)

This procedure joins Schemes 1 and 2 and projects the result onto the target set returning the final project and the associated cost. A check is made to see if a primary key of the original joined product remains in the final target set.

If not, and the final product exceeds the available work space, then the costs to perform a sort for duplicates of the final product is added to the query cost.

```
PROCEDURE PROJECT (IN Scheme 1: Scheme_Pointer;  
                   OUT Final_Scheme: Scheme_Pointer;  
                   IN Target_Set: Attribute_Set;  
                   OUT Cost: Real)
```

This procedure projects Scheme 1 onto the Target Set returning the Final Scheme result and the associated cost. As above, if a primary key is not present and the final product is too large for the work space, we include the costs for an external merge sort.

A.6 SCHEME SIZE MODULE

```
PROCEDURE SIZE TABLE(IN Scheme_Attributes : Attribute_Set;  
                     IN Root: Attribute_Set;  
                     OUT Details: Size_Estimates;  
                     OUT Attribute_Usage:  
                         ARRAY [1...Max_Number_Of_Attributes]  
                         OF (Used, Not_Used);  
                     OUT All_Used: Boolean);
```

This procedure is called to evaluate each possible root. It returns the Size_Estimates, whether all attributes were selected (All_Used:=TRUE) and final usage results for the attributes in the scheme.

PROCEDURE REPORT SIZE (IN Table:Scheme)

Given the appropriate output level selected, this routine reports the table size in tuples and bytes in the following form:

#Records nn #BYTES/REC nn Total Size nn MBytes

PROCEDURE REPORT UNUSED ATTRIBUTES(IN Attribute_Usage:
 ARRAY [1...Max_Number_Of_Attributes]
 OF (Used, Not_Used));

Reports the number and name of all attributes not covered during the estimation process.

APPENDIX R

REFERENCES

The following abbreviations are used:

IEEE	Institute of Electrical and Electronics Engineers
ACM	Association For Computing Machinery
SIGMOD	Special Interest Group for Management of Data
TODS	Transactions on Database Systems
COMPSAC	Computer Software and Applications Conference
IFIP	Institute for Information Processing

- [AHO79] Aho A.V., Beeri C., Ullman J.D. "The theory of joins in relational databases", ACM TODS 4, pp. 297-314.
- [ANSI75] Study Group Report, ANSI X3/SPARC/DBMS, American Standards Inst, Wash., DC. 1975.
- [ARMS74] Armstrong W.W. "Dependency structures of database relationships", Proc. IFIP Congr. 74, North Holland, New York, pp. 580-583.
- [AROR81] Arora S.K., Smith K.C. "Graphical Normal Forms Based on Root Dependencies in Relational Data Base Systems", Int. Journal of Computer and Information Sciences, Vol. 10, No. 4, August 1981, pp 235-260.
- [BEER77] Beeri C., Fagin R., Howard J.H. "A complete axiomatization for functional and multivalued dependencies in database relations", Proc. ACM SIGMOD, pp. 47-61.

- [BEER78] Beerl C., Bernstein P.A., Goodman N. "A sophisticate's introduction to database normalization theory", Proc. 4th Int. Conf. Very Large Data Bases, pp. 113-124.
- [BEER79] Beerl C., Bernstein P.A. "Computational problems related to the design of normal form relational schemas", ACM TODS 4, pp. 30-59.
- [BEEP80] Beerl C. "On the membership problem for functional and multivalued dependencies in relational databases", ACM TODS 5, pp. 241-259.
- [BERN76] Bernstein P.A. "Synthesizing third normal form relations from functional dependencies", ACM TODS 1, pp.277-298.
- [BLAS77] Blasgen M.K., Eswaren K.P. "Storage and access in relational databases", IBM Syst. Journal 16, pp.363-377.
- [BUBF77] Bubenko J.A., Jr. "IAM: An inferential abstract modelling approach to design of conceptual schemas", Proc. ACM SIGMOD, pp. 62-74.
- [BUNE79] Buneman O.P., Clemens F.K. "Efficiently monitoring relational databases", ACM TODS 4, pp. 368-382.
- [CHAM81] Chamberlin et al, "History and Evaluation of System R", Comm. ACM Vol. 24, Oct 1981, pp. 82-91.
- [CHEN76] Chen, P.P. "The Entity-Relationship Model - Toward a Unified View of Data", ACM TODS 1, pp. 9-36.
- [CODD70] Codd E.F. "A relational model for large shared data banks" Comm. ACM 13, pp. 377-387.
- [CODD79] Codd E.F. "Extending the relational model to capture more meaning", ACM TODS 4, pp. 397-424.
- [DATE80] Date C.J. Introduction to Data Base Systems , 3d Ed. Addison-Wesley, Reading, Mass., 1980.
- [DELO78] Delobel C. "Normalization and hierarchical dependencies in the relational data model", ACM TODS 3, pp. 201-222.
- [FAG77a] Fagin R. "The decomposition versus the synthetic approach to relational database design", Proc. 3rd Int. Conf. Very Large Data Bases, pp. 441-446.

- [FAG77b] Fagin R. "Multivalued dependencies and a new normal form for relational databases", ACM TODS 2, pp. 262-278.
- [FAGI81] Fagin R. "Normal Form for Relational Databases Based on Domains and Keys", ACM TODS 6, pp. 387-415.
- [GANE80] Gane C., Sarsens T. Structured Systems Analysis - Tools and Techniques, Prentice-Hall, Englewood Cliffs, N.J., 1980.
- [GOTL75] Gottlieb L.P. "Computing Joins of Relations", Proc. ACM SIGMOD 75, pp. 55-63.
- [HAMM76] Hammer M.M., McLeod D. "Framework for Data Base Semantic Integrity", Proc. ACM Conf. on Software Eng. 1976 pp. 498-504.
- [HAMM79] Hammer M.M., Niamir B. "A Heuristic Approach to Attribute Partitioning", Proc. ACM SIGMOD 1979, pp. 93-115.
- [IBM75] Data Base Design Aid General Information Manual GH-20-1626-C, International Business Machines Inc.
- [KENT81] Kent W. "Consequences of Assuming a Universal Relation", ACM TODS 6, pp. 539-556.
- [KNUT73] Knuth D.E. The Art of Computer Programming, Vol III, "Sorting and Searching", Addison-Wellesley, Reading, Mass, 1973.
- [LIEN81] Lien Y.F. "Hierarchical Schemata for Relational Databases" ACM TODS 6, pp. 48-69.
- [LOZI80] Lozinskii E.L. "Construction of Relations in Relational Databases", ACM TODS 5, pp. 208-224.
- [MELK80] Melkanoff M.A., Zaniolo C. "Decomposition of Relations and Synthesis of Entity Relationship Diagrams", Proc. Symposium ER Approach to Systems Analysis, North Holland, New York, 1980, pp. 277-284.
- [RISS77] Pissanen J. "Independent Components of Relations", ACM TODS 2, pp. 317-325.
- [SCHK75] Schkolnick M. "Secondary Index Optimization", Proc. ACM SIGMOD 1975, pp. 186-192.

- [SCHK79] Schkolnick M., Tiberio P. "Considerations in Developing a Design Tool for a Relational DBMS", Proc. Compsac 1979, pp. 228-235.
- [TANA79] Tanaka Y. "Logical Design of a Relational Schema and Integrity of a Database", Database Architecture, North Holland, New York, 1979, pp. 297-316.
- [TEIC75] Teichrow D., Bastarache M., PSL/PSA Users' Manual, ISDOS Working Paper No. 98, The University of Michigan, March 1975.
- [TOTH80] Toth, K.C. Distributed Database Architecture and Query Processing Strategies, PhD Dissertation, Carleton University, April 1980.
- [TSIC82] Tsichritzis D.C., Lochovsky F.H. Data Models, Prentice-Hall, Englewood Cliffs, N.J.
- [ULL80] Ullman J.D., Principles of Database Systems, Computer Science Press, Rockville, Maryland 1980, pp. 180-185.
- [WASS81] Wasserman A.I., Botnick K. "Annotated Bibliography on Data Design", SIGMOD RECORD Jan 1981, pp. 48-77.
- [YAO79] Yao S.B. "Optimization of Query Algorithms", ACM TODS 4, pp. 133-155.
- [ZANI81] Zaniolo C., Melkanoff M.M. "On The Design Of Relational Data Base Schemata", ACM TODS 6, 1981, pp. 1-47.

END

06109H84

FIN