8-1989

An Efficient Implementation of Logical Design for Relational Databases

Shakeel Ishaque
Western Michigan University

AN EFFICIENT IMPLEMENTATION OF LOGICAL DESIGN FOR RELATIONAL DATABASES

by

Shakeel Ishaque

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
August 1989

AN EFFICIENT IMPLEMENTATION OF LOGICAL DESIGN FOR RELATIONAL DATABASES

Shakeel Ishaque, M.S.

Western Michigan University, 1989

This work presents efficient methodology for the design of relational databases and an implementation of a design tool. A set of algorithms and supporting theory are discussed. Improvements are made on existing decomposition approaches.

The Dependency Preserving Normal Fom (DPNF), which is stronger than 3NF, is presented. It guarantees a decomposition with lossless join property and at the same time preserves all of the functional dependencies.

An algorithm to obtain DPNF decomposition is presented. The algorithm computes DPNF decomposition in polynomial time. It converts supplied functional dependencies to annular cover and uses reduced annular cover to compute closures and decomposition. The algorithm finds the keys for every decomposed scheme and the original scheme. The document contains a PASCAL implementation of the algorithm.

ACKNOWLEDGEMENTS

I would like to thank Dr. Motzkin for guiding me through this project. Regardless of her busy schedule she provided sufficient time to discuss problems and to convey her scholarly advice with possible solutions. I would also like to thank her for letting me do CS 710 with her while I was still taking CS 643. I would like to thank Dr. Boals and Dr. Williams for reading and providing valuable comments and suggestions to improve this work. Special thanks to Mr. Mani, a very good friend and colleague, for his initial contributions while working on a CS 710 project and for pursuading me to do this work.

Shakeel Ishaque

Order Number 1337927

An efficient implementation of logical design for relational databases

Ishaque, Shakeel, M.S.

Western Michigan University, 1989

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

This work presents efficient methodology for the design of relational databases and an implementation of a design tool. A set of algorithms and supporting theory are discussed. Many of the algorithms, theorems, lemmas, and proofs were outlined by Dr. Motzkin. The algorithms have been implemented with PASCAL programs.

There are basically three approaches for the design of relational database: analytical approach, which was introduced in (Codd 1971), synthetic approach, introduced in (Bernstein 1976) and integrated approach introduced in (Beeri, et al. 1986). In the integrated approach, the design process is similar to the synthetic approach but multivalued dependencies are considered.

The design method described here also uses an integrated approach to efficiently design the database but this integrated approach is different from (Beeri, et al. 1986) as pointed out below. The database designer defines a set of attributes and a collection of data dependencies which may or may not be universal. This work does not deal with the controversy of universal scheme ((Fagin, et al. 1982), (Kent 1981), (Kent 1983), (Ullman 1983)). It tries to design a database either if a universal scheme is given or a collection of relation schemes are given as input. If several schemes are given as input then it looks upon the dependencies of each scheme and applies the design approach to each scheme. The problem addressed is how one can derive a database scheme with certain desirable properties from the information provided by the database designer. Among the properties considered are those of lossless join property, preservation of dependencies, and normal forms such as 3NF, 4NF, BCNF.

A new type of normal form called Dependency Preserving Normal Form (DPNF), is presented which is stronger than 3NF. A simple algorithmic approach to obtain non redundant DPNF database schemes with near minimum relation schemes is

presented, in addition it resolves some drawbacks of earlier methods. The approach used by (Beeri, et al. 1986) is to first decompose the database on the basis of multi valued dependencies (MVDs) and then on the basis of functional dependencies (FDs) whereas the algorithm presented here first uses FDs to obtain 3NF database scheme and then incorporate into it some of the algorithms given in ((Beeri 1980), (Beeri, et al. 1986), (Yuan 1987)) using MVDs to obtain DPNF. For the definitions of MVD and FD refer to chapter II. After decomposing on the basis of FDs, sometimes, new MVDs appear; that is why we decompose first on the basis of FDs and then MVDs.

The algorithm considered here is simplified and somewhat faster than the earlier approaches, and as noted above, it achieves a DPNF database. The algorithm to obtain a near optimal design is also presented. The complexities using this approach are calculated and compared to previous techniques. The algorithm is described in detail in chapter III.

CHAPTER II

TERMINOLOGY AND NOTATIONS

This chapter will introduce the concepts and terms used in this document. The terms and notations that are mentioned in this document, but not defined here, are used in the same way as in (Maier 1983).

FUNCTIONAL DEPENDENCY: Let r be a relation on scheme R and let X, Y be two subsets of R. Relation R satisfies the functional dependency X ---> Y if for any two tuples t1 and t2 whenever t1(X) = t2(X) then t1(Y)=t2(Y).

CLOSURE OF A SET OF FUNCTIONAL DEPENDENCIES: Let F be a set of FDs over a relation scheme R. The closure of F, written F^+, is the smallest set containing F such that Armstrong's axioms (Armstrong, 1974) can not be applied to the set to yield a FD not in the set.

TRANSITIVE DEPENDENCY: Given a relation scheme R and X, Y subsets of R. Let A be an attribute in R and F be a set of FDs. A is transitively dependent on X in R if there is Y such that X ---> Y, Y ---> A and Y -/-> X under F and A is not in XY.

PRIME ATTRIBUTE: Given a relation scheme R, an attribute A in R, and a set of FDs F over R. Attribute A is prime in R with respect to F if A is contained in some key of R. Otherwise A is nonprime in R.

THIRD NORMAL FORM: A relation scheme R is in third normal form (3NF) with respect to set of FDs F if all of its attributes are atomic and no nonprime attribute of R is transitively dependent on a key of R. A database is in third normal form if every relation scheme in it is in third normal form.

BOYCE-CODD NORMAL FORM: A relation scheme R is in Boyce-Codd normal form (BCNF) with respect to set of FDs F if all of its attributes are atomic and no attribute of R is transitively dependent on any key of R. A database is in Boyce-Codd normal form if every relation scheme in it is in Boyce-Codd normal form.

MULTIVALUED DEPENDENCY: Let r be a relation on scheme R and let X, Y, Z be three disjoint subsets of R such that R = X U Y U Z. We say that Y is multivalued dependent on X, if there exist tuples t_1, t_2 , with t_1 having X and Z values x1, z1 and t_2 having X and Y values x1,y1 then there exist a tuple t_3 with X, Y, and Z values x1,y1,z1. The tuples t_1, t_2, t_3 do not need to be distinct.

Notation: We denote X --->---> Y to mean that Y is multivalued dependent on X. The abbreviation MVD is also used.

STRICT MULTIVALUED DEPENDENCY: A MVD that is not a FD is called a strict MVD. That is If X -->--> Y but X-/-> Y then X-->--> Y is a SMVD and is denoted by X-->-->--> Y.

DIRECT DEPENDENCY: Let X be a subset of R and A an attribute of R. Let X ---> A be in F^+. We say that A is directly dependent on X (or X directly implies A) if A is non transitively dependent on X. It is denoted by X ===> A.

A set of attributes Y is directly dependent on X if each attribute of Y is directly dependent on X.

Example 2.1: Let F={ A ---> B, B ---> C, A ---> D, D ---> E, A ---> C } be a set of FDs over R(ABCDE).

FD A ---> B is a direct dependency whereas A ---> C is not a direct dependency.

CONTAINED FUNCTIONAL DEPENDENCY: Let R be a relation scheme and F be the set of FDs over R. Let R_i be a subset of R. A FD of F^+, X ---> Y,is said to be

contained in a relation scheme R_i if XUY is a subset of R_i.

Example 2.2: Consider example 2.1.

Let R1(ABD) and R2(BC) be a decomposition of R then A ---> B is a contained FD in R1 while D ---> C is not contained in either R1 or R2.

LOST FUNCTIONAL DEPENDENCY: Let $R_1, R_2, \ldots, R_n$ be a decomposition of a relation scheme R and let G be a set of FDs over R. Let F be a set of all functional dependencies contained in any of R_i. The FD, X ---> Y of G^+, is said to be a lost FD if it is not in F^+.

Example 2.3: Let R(ABC) be a relation scheme with a set of FDs {AB ---> C, C---> B}. Let R1(AC) and R2(CB) be a decomposition of R then AB ---> C is a lost FD. Note that decomposition has lossless join property.

On the other hand let R(XYZ) be a relation scheme with set of FDs {Y---> X, X ---> YZ}. The decompositon of R into R1(XY) and R2(XZ) results in no lost FDs.

KEY BASED DECOMPOSITION: Let R (XYZ) be a relation scheme. Let X, Y, and Z be three sets of attributes where X is a key of R. A decompositon of R into R1(XY) and R2(XZ) is a key based decompositon.

Example 2.4: Let F ={A ---> B, A ---> C, B ---> D} be a set of FDs over R(ABCD).

A decomposition of R into R1(ABD) and R2(AC) is called a key based decomposition whereas a decomposition of R into R1(ABC) and R2(BD) is not a key based decomposition.

DEPENDENCY PRESERVING NORMAL FORM: Let R be a relation scheme, F be a set of FDs over R then a decomposition of R into $R_1, R_2, \ldots, R_n$ is in Dependency Preserving Normal Form if the following conditions are met:

D1. Every scheme R_i is in 3NF.

D2. There are no lost functional dependencies, of F.

D3. The decomposition has lossless join property.

D4. There are no non-trivial functional and multivalued dependencies on the basis of which a relation scheme can be decomposed, without violating conditions D2 or D3, except for a key based decomposition of it.

DPNF is stronger than 3NF because it guarantees a decomposition with lossless join property, whereas the conventional way of synthesis sometimes results in a decomposition with no lossless join property. DPNF yields decomposition with fewer schemes by removing redundant schemes. Unlike 3NF, DPNF guarantees preservation of FDs. Also, it decomposes as much as possible by removing prime attributes from relations as long as no FDs are lost. DPNF also decomposes on the basis of some MVDs.

Example 2.5: Consider R = {ABCDEF} and F = { AB --->D, D ---> AF, C ---> B, CD ---> E, B ---> C, E ---> A }.

Conventional approach for 3NF decomposition is R1={ABCDE} with Key ={AB, CD}, R2={ADF} with Key={D}, R3={CB} wiht Key={C, B} and R4={EA} with Key ={ E}.

The presented approach will produce R1={ABDE} with Key={AB}, R2={ADF} with Key={D} and R3={CB} with Key={C, B}.

MINIMAL DEPENDENCY PRESERVING NORMAL FORM: Let R be a relation scheme, F be a set of FDs over R then a decomposition of R into R_1, R_2, ... ,R_n is in minimal dependency preserving normal form if the following conditions are met:

1. The decomposition is in DPNF.

2. There is no decomposition of R which is in DPNF and has fewer relation schemes.

Example 2.6: Let F = { AB ---> CDIEH, CDI ---> ABE, EK ---> H, HI ---> EJ, AE ---> AH, H ---> IK, D ---> EHK, AH ---> AE } be a set of FD over R(ABCDEHIJK). The minimal DPNF decomposition for R may be:

R1(ABCD) with Keys={AB, and CD}

R2(EHIJK) with keys={EK, and H}

R3(AEH) with keys={AE}

R4(DEK) with key={D}

It is clear that R1, R2, R3, and R4 are in 3NF and there are no lost FDs. It has been proved in (Biskup, et al. 1979) that if all FDs are preserved then presence of a key of R guarantees a lossless join property. AB is key of scheme R therefore the above decomposition has lossless join property. The above decomposition cannot be decomposed any further because doing so will result in lost FDs. Hence, the above decomposition is in DPNF.

Combining any of the above relation schemes will result in a decomposition which is not in 3NF. Therefore, there is not a DPNF decomposition of R with fewer than four relations. Hence, the above decomposition is in minimal DPNF.

The above decomposition is in DPNF and there is not any DPNF decomposition for R with fewer relation schemes.

OPTIMAL DEPENDENCY PRESERVING NORMAL FORM: Let R be a relation scheme, F be a set of FDs over R then a decomposition of R into R_1, R_2, ... ,R_n is in optimal dependency preserving normal form if the following conditions are met:

1. The decomposition is in DPNF.

2. There is no decomposition of R which is in minimal DPNF and has fewer attributes in it.

Example 2.7: Consider set of FDs, F and relation scheme of example 2.6.

The decomposition of example 2.6 is optimal but not minimal that is because R3 can have only attribute H instead of EK. Therefore the minimal DPNF decomposition for R is:

R1(ABCD) with Keys={AB, and CD}

R2(EHIJK) with keys={EK, and H}

R3(AEH) with keys ={AE}

R4(DH) with key={D}

The decomposition of example 2.6 is in minimal DPNF i.e. there is not any DPNF decomposition for R with fewer relations in it. Therefore by replacing the sets with equivalent sets having less number of attributes in them, will result in a decomposition with fewest possible attributes. In the above decomposition EK is replaced with its equivalent set H to yield an optimal DPNF decomposition.

REDUNDANT SCHEME: A scheme R_i is said to be redundant in a database if it can be removed without losing the DPNF properties.

Example 2.8: Let F={A ---> BE, BD ---> C, C ---> D, E --->C} be a set of FDs over R(ABCDE).

A decomposition of R into R1(ABE), R2(BCD), R3(EC), and R4(CD) has a redundant scheme R3.

NON REDUNDANT DATABASE: A database is non redundant if it does not contain any redundant scheme.

Example 2.9: Consider example 2.8.

A decomposition of R into R1(ABE), R2(BCD), and R3(EC) has no redundant schemes therefore it is a nonredundant database.

EQUIVALENCE CLASSES: Let F be a set of FDs and $e_F(X)$ be the set containing those sets of attributes appearing on the left sides of any functional dependency of F which are equivalent to X. Let $E_F(X)$ be the set of all functional dependencies whose left sides are in $e_F(X)$. The equivalence classes e_F and E_F is defined as :

e_F = { $e_F(X)$: $e_F(X) \neq e_F(Y)$ where X, Y is the set of attributes that appeared on left side with $X \neq Y$ }

E_F = { $E_F(X)$: $e_F(X) \neq e_F(Y)$ where X, Y is the set of attributes that appeared on left side with $X \neq Y$ }

Example 2.10: Let F={A ---> B, B ---> A, C ---> B, D ---> B, CD ---> E} be a set of FDs over R(ABCDE).

e_F={ $e_F(A)= e_F(B)$={A,B}, $e_F(C)$ ={C}, $e_F(D)$ ={D}, $e_F(CD)$ ={CD} } and

E_F={ $E_F(A) = E_F(B)$ = { A ---> B, B ---> A }

$E_F(C)$ = { C ---> B }

$E_F(D)$ = { D ---> B }

$E_F(CD) = \{ CD \text{ ---> } E \} \}$.

COMPOUND FUNCTIONAL DEPENDENCY: Let R be a relation scheme and $X_1, X_2, \ldots, X_n$ be subsets of R, where X_i and X_j are equivalent for i, j = 1...n, such that $n \geq 1$. Let Y be the set of attributes which may or may not be empty. The compound functional dependency is of the form $(X_1, X_2, \ldots, X_n)$ ---> Y, such that intersection of Y with any X_i is empty and X_i ---> Y is in F^+ but Y -/-> X_i. Each of the X_i is called a left set of the CFD_X, Y is called the right set of CFD_X.

Example 2.11: Let AB ---> D and D ---> ABCF.

Since AB and D are equivalent i.e AB derives D and vice versa.

Therefore it will result in the following CFD: (AB, D) ---> CF.

ANNULAR COVER: The annular cover is a set of compound functional dependencies such that the set of all the projections of X_i ---> Y form a cover. If two left sets X_i, and X_j are equivalent then both of them are in same CFD. The closure of the annular cover is the closure of all the projections X_i ---> Y.

Example 2.12: Let F= { A ---> B, B ---> AE, C ---> D, D ---> CF, AC ---> BD, BD ---> ACF} be a set of FDs over R(ABCDEF).

The annular cover, G, for the above FDs may be G={(A,B) ---> E, (C,D) ---> F, (AC,BD) ---> F}.

NONREDUNDANT ANNULAR COVER: An annular cover G is nonredundant if no CFD can be removed from G without altering the closure of G.

Example 2.13: Consider example 2.12.

In the annular cover G described in example 2.12 CFD (AC,BD) ---> F is redundant therefore the set G of CFD is not a non redundant annular cover. But without CFD, (AC, BD) ---> F, the set G is a nonredundant annular cover.

REDUCED ANNULAR COVER: Let G be a nonredundant annular cover. A CFD of

G

$(X_1, X_2, \ldots, X_n)$ ---> Y is reduced if the following are satisfied;

1) No X_i can be shifted to the right set without changing the closure of G.

2) No attribute of any of X_i can be shifted to the right set without changing the closure of G.

3) No attribute from the left or the right sets of any CFD can be removed without changing the closure of G.

If all of the CFDs in G are reduced then G is a reduced annular cover.

Example 2.14: Consider a set of FDs { AB ---> CD, CD ---> ABEFGH, BEF ---> AD, A ---> C, B ---> D, E ---> F, H ---> G }

After converting these FDs into CFDs we get the annular cover as: { (AB, CD, BEF) --->GH, (A)---> C, (B) ---> D, (E) ---> F, (H) ---> G }

But the above annular cover is not reduced. After shifting left sets to the right set we get

{ (CD, BEF) ---> AGH, (A)---> C, (B) ---> D, (E) ---> F, (H) --->G }

Still the attribute F in the set BEF is extraneous and can be discarded i.e there is a need to check if any attribute is extraneous in any of the left sets. The new annular cover, after discarding all extraneous attributes of left sets is :

{(CD, BE) ---> AGH, (A)---> C, (B) ---> D, (E) ---> F, (H) --->G }

Now check if there are any extraneous attributes in the right set. Obviously G is extraneous in the first CFD. Therefore the reduced annular cover is:

{(CD, BE) ---> AH, (A)---> C, (B) ---> D, (E) --->F, (H) --->G }.

KEY: Let R(A1, A2, . . ., An) be a scheme and {B1, B2, . . ., Bm} be a subset of R. K(B1, B2, ... , Bm) is called a key of R if ti(B1, B2, ... Bm) ≠ tj(B1, B2, ... Bm) whenever i ≠ j.

Example 2.15: Let F={AB ---> DE, E ---> ABC} be a set of FDs over R(ABCDE). {AB}, {E}, {ABC}, {ABE} are all some of the keys of R.

REDUNDANT KEY: A key is called a redundant key if a proper subset of it is also a key.

Example 2.16: Consider example 2.15.

{ABC}, {ABE} are both redundant keys of R whereas {AB} and {E} are not redundant keys of R.

REDUCED KEY: A key is said to be reduced if no proper subset of it is a key.

Example 2.17: Consider example 2.15.

{AB}, and {E} are the only two reduced keys of R.

MINIMAL KEY: A key, K, of a relation scheme r(R) is said to be MINIMAL if there is no key having smaller number of attributes than K. Note that a minimal key is always a reduced key.

Example 2.18: Consider example 2.15.

{E} is the only minimal key of R.

KERNEL: A kernel is a set of attributes which do not appear on the right side of any of the functional dependencies of a reduced cover of F.

Note that an attribute that does not appear on the right side of any FD in a reduced cover of F, does not appear on the right side of any FD in any reduced cover of F.

Note that kernel is a part of every key. This is due to the fact that no set of attributes non-trivialy imply the kernel or parts of the kernel.

Example 2.19: Let F={A ---> B, B ---> DE, C ---> B} be a set of FDs over R(ABCDEH).

ACH is the kernel of R. This is because A and C appear only on the left side of FDs whereas H did not take part in any FD.

NOTATIONS

The symbols below are used in this paper with the following meanings.

1. := - Assignment operator.

2. = - Equality Comparison.
3. U - Union of two sets.
4. ⊆ - Subset of.
5. - - Difference of two sets.
6. - - - > - Functional dependency also denoted as FD.
7. ==> - Directly implied FD.
8. CFD_X - Compound FD with left set Equivalent to X.
9. { } - Empty set.
10. ≠ - Not equal to comparison.
11. card(S) - Cardinality of the set S.
12. - - > - - > - Multivalued dependency. Also denoted as MVD.
13. - - > - - > - - > - Strict Multivalued dependency also denoted as SMVD.

CHAPTER III

OUTLINE AND DETAILED STEPS OF THE ALGORITHM

This work presents a design algorithm which accepts relation schemes, which may or may not be universal, and sets of functional dependencies among the attributes of each scheme. Some attributes of the scheme may not even take part in any functional dependency. This algorithm is shown to be computed in polynomial time with respect to the length of input. The objective of this algorithm is to construct DPNF database schemes with the properties D1, D2, D3, and D4. The algorithm is designed using top down organization. The top layer of the algorithm is as follows:

ALGORITHM MakeDPNF:

FOR i = 1 to number of input schemes DO

STEP 0 : Read in the relation scheme U_i and its functional dependencies, F.

STEP 1 : Compute a reduced annular cover of the set of functional dependencies.

STEP 2 : Determine a set of relation schemes of the database satisfying properties D1, D2, D4 and also compute some of the reduced keys of each schemes so constructed.

STEP 3 : Check whether any of the left sets is a key of U_i.

STEP 4 : If yes, then do step 8 or else do step 5.

STEP 5 : Determine a reduced key, K, of the original scheme U.

STEP 6 : Add a new scheme composed of the attributes of K.

STEP 7: Remove strict multivalued dependencies from K, to obtain DPNF decomposition.

STEP 8: Remove redundant relation schemes.

Computing the Reduced Annular Cover

The step 1 consists of following substeps:

STEP 1.1 : Determine the equivalence classes e_F and E_F.

STEP 1.2 : Determine the annular cover.

STEP 1.3: Determine reduced annular cover.

Determine Equivalence Classes

The equivalence classes e_F and E_F are used to construct the compound functional dependencies from the original functional dependencies.

The algorithm LINDERIVES determines if two sets of attribute X and Y form an FD:

X ---> Y in F^+. This algorithm is used in part to determine the equivalence classes e_F and E_F.

```
Algorithm  LinDerives( F,  X,  Y )

 Input :      F  -   Set of functional dependencies, denoted by LS[i] --->
 RS[i].
              X, Y -  Attribute Set.
 Variables :  R   -   Set containing all the attributes.
              p   -   Number of functional dependencies.
               LS[i] - Array of left sets where i = 1 ... p.
               RS[i] - Array of right sets where i = 1 ... p.
 Output :  return true  if X derives Y otherwise return false.
 Body   :
    for each attribute attr in R do
        A[attr] := 0
    for i = 1 to p do
        count[i] := 0
        for each attribute attr in LS[i] do
            A[attr] := A[attr] U { i }
            count[i] := count[i] + 1
        end
```

```
end
Old := X;  New := X
if (Y ⊆ Old) then return(true)
While ( New ≠ { } ) do
    Choose an attribute attr from New
    New := New - {attr}
    for each i in A[attr] do
        count[i] := count[i] - 1
        if (count[i] = 0) then
            for each attribute ele in RS[i] do
                if not(ele in Old) then
                    Old := Old U {ele}
                    New := New U {ele}
                    if (Y ⊆ Old) then return(true)
                end
    end
end
return(false)
```

The LinDerives algorithm is a modification of the algorithm LINCLOSURE in (Beeri, et al. 1979). Unlike algorithm LINCLOSURE, LinDerives() does not necessarily compute the actual closure of X, therefore LinDerives is faster while it has the same complexity. The algorithm is clearly correct. It is guaranteed to halt as the number of attributes and number of functional dependencies are finite. The complexity of the above algorithm in the worst case has been proved to be O(ap) in (Beeri, et al. 1979), where **p** is the number of FDs and **a** is the number of attributes in scheme.

The algorithm to determine equivalence classes uses the algorithm Equivalence. The algorithm Equivalence() finds out if two sets of attributes X and Y are equivalent.

```
Algorithm  Equivalence(F,  X ,  Y)

 Input  :  F  -  set of functional dependencies.
 Output :  If  X and Y are equivalent then  true  else  false.
 Body   :
    if (LinDerives(F, X, Y) and LinDerives(F, Y, X )) then return(true)
    else   return(false).
    end.
```

This algorithm is clearly correct. The complexity of this algorithm is based on

the complexity of algorithm LinDerives i.e., O(ap).

The above algorithm finds out if two set are equivalent or not, whereas the algorithm Determi_e&E() determines the equivalence classes for all left sets of the set of FDs F. The algorithm to construct equivalence classes e_F and E_F is as follows:

```
Algorithm Determi_e&E( F, eF, EF)

 Input  : F  - Cover of functional dependencies.
 Output : eF - Sets of  equivalent attributes.
          EF - Sets of FDs of F with equivalent left sets.
 Body   :
   G := F
   Until  G is empty do
      Select an FD X ---> Y from G
      eF(X)  :=  { X }
      EF(X) := { X ---> Y}
      G := G - { X ---> Y }
      for each FD W ---> U in G do
         if  (Equivalence(F, X, W)) then
               eF(X) := eF(X) U { W }
               EF(X) := EF(X) U { W ---> U }
                G := G - { W ---> U }
         end
      end
   end
```

The above algorithm is indeed correct, by definition of equivalent classes. The algorithm is guaranteed to halt because at least one functional dependency is removed every iteration of the until loop. For Ith iteration of the until loop, the for loop will be executed (p - i) + 1 times in the worst case, where 1<= i <= p where p is number of FDs in F. The complexity of the algorithm to determine $e_F(X)$ and $E_F(X)$ for all left hand sides of functional dependencies of a cover is $O(ap^3)$.

Example 3.1: Consider example 2.6. The set of equivalence classes e_F and E_F are given below.

e_F={ e_F(AB)= e_F(CDI) = {AB, CDI}, e_F(EK)= e_F(H)= e_F(HI) = {EK, H, HI},

$e_F(D) = \{D\}$, $e_F(AE) = e_F(AH) = \{AE, AH\}$ } and

E_F={ $E_F(AB) = E_F(CDI) = \{$ AB ---> CDIEH, CDI ---> ABE }

$E_F(EK) = E_F(H) = E_F(HI) = \{$ EK ---> H, HI ---> EJ, H ---> IK }

$E_F(D) = \{$ D ---> EHK }

$E_F(AE) = E_F(AH) = \{$ AE ---> AH, AH ---> AE } }.

Determine Annular Cover

The algorithm to determine the compound functional dependency and annular cover is given below.

Algorithm Compound (e_F, E_F, G)

```
Input  : eF - Sets of equivalent attributes.
         EF - Sets of FDs with equivalent left sets.
Output : G  - Set of compound functional dependencies.
Body   :
  G := { }
  for each member eF( W ) of eF do
      left_sets := left_attr := right_attr := { };
      for each FD V ---> X in EF( W ) do
          left _attr  := left _attr U V
          right _attr := right _attr U X
          Add V to the left_sets of the cfd C.
      end
      right _attr := right _attr - left _attr
      right _set := right_attr
      G := G U cfd (left_sets) ---> right_set
  end
```

The algorithm is clearly correct and guaranteed to halt. The complexity of this algorithm is O(p). The set of compound functional dependencies obtained is the annular cover of the original set of functional dependencies.

Compute Reduced and NonRedundant Annular Cover

In order to compute reduced annular cover; first, shiftable members of left set are shifted to right; second, remove the extraneous attributes in every member of left set; third step is to discard extraneous attributes of right set.

In order for us to construct a reduced annular cover we have to determine the closure of a set of attributes using annular cover. The algorithm presented below accepts two sets of attributes X , Y and an annular cover AnnCov. It returns a value of true if X ---> Y is in closure of AnnCov.

```
Algorithm Annderive( X, Y, AnnCov )

    Input  :  X is a set of attributes.
              AnnCov is the Annular Cover.
              Y is a set of attributes.
    Output :  if X derives Y then return TRUE else return FALSE
    Body:
    IF (X ⊆ Y)
    then   return(TRUE).
    FOR each CFD, C, of AnnCov do
        FOR each attribute A of C do
            Right_set[C] := Right_set[C] U {A}.
        FOR each left set Xi of C do
            count[C,i] := 0
            FOR every attribute A of Xi do
                add (C,i) to list[A]
                count[C,i] := count[C,i] + 1
    Newcl := X.
    Oldcl := X.
    While Oldcl ≠ {} do
        choose an attribute A of Oldcl
        For each (C,i) in List[A] do
            Count[C,i]  := Count[C,i] -1.
            if  Count[C,i] = 0  then
                Exten := Right_set[C] - Newcl.
                Newcl := Newcl + Exten
                Oldcl := Oldc + Exten
                if ( Y ⊆ Newcl )
            then  return( TRUE).
```

```
        Oldcl := Oldcl - {A}
    return(FALSE)
    end
```

The algorithm Annderive() is based on the correctness of Linclosure. The complexity of the above algorithm is O(ap) i.e the complexity is linear to the input length.

The example 2.6 is traced through the entire algorithm step by step.

Example 3.2 : The annular cover for equivalence classes formed in Example 3.1 is:

{ (AB, CDI) ---> EH, (EK,H, HI) ---> J, (D) ---> EHK, (AE, AH) ---> {} }.

The algorithm to rigth shift sets of shiftable left sets is described below:

Algorithm Shiftleftset(AnnCov)

```
Input   :  AnnCov is the Annular Cover.
Output  :  AnnCov with all shiftable Left sets shifted to right set.
Body:
   For each CFD (X1, . . ., Xn) ---> Y in AnnCov do
      If n > 1 then
         i = 1
         Count := n
         While (i <= n)  and  (Count >1) do
           NC := AnnCov - CFDX
           LeftSets := LeftSets of CFDX - Xi
           RightSet := RightSet of CFDX U Xi
           J := the first Set in LeftSets
           NC := NC U { LeftSets ---> RightSet }
           If AnnDerives (Xi, J, NC) then
               CFDX := LeftSets ---> RightSet
               AnnCov := NC
               Count := Count -1
           i= i +1
      AnnCov = NC
   end
```

Example 3.3: The annular cover after Shiftleftset() is performed on the annular cover in Example 3.2 is:

{ (AB, CDI) ---> EH, (EK, H) ---> IJ, (D) ---> EHK, (AE) ---> H } The attribute set HI is shifted in second CFD and AH is shifted in fourth CFD from left set to the right set.

The algorithm for removing extraneous attributes in left sets is as follows:

```
Algorithm Removeleftattr( AnnCov )

 Input  : AnnCov is the Annular Cover.
 Output : AnnCov with no extraneous attributes in Left sets.
 Body:
   For each CFD (X1, . . ., Xn) ---> Y in AnnCov do
      For each Xi of CFD do
         For each attribute A of Xi do
           If Annderive(Xi-A, A, AnnCov) then
               AnnCov := AnnCov - {(X1, . . ., Xn) ---> Y}
               AnnCov := AnnCov U {(X1, . . ., Xi-A, . . ., Xn) ---> Y}
   end
```

The worst complexity of Removeleftattr() algorithm is $O(a^2p^2)$.

Example 3.4: The annular cover after Removeleftattr() is performed on the annular cover in Example 3.3 is given below:

{ (AB, CD) ---> EH, (EK, H) ---> IJ, (D) ---> EHK, (AE) ---> H } The attribute I is removed in second CFD.

The algorithm to discard the extraneous attributes in the right set is:

```
Algorithm Rightreduce( AnnCov )

 Input  :  AnnCov is the Annular Cover.
 Output :  AnnCov with all extraneous attributes of right set discarded.
 Body:
   For each CFD (X1, . . ., Xn) ---> Y in AnnCov do
      For each attribute A of Y do
         NC :=  AnnCov - {(X1, . . ., Xn) ---> Y}
         NC :=NC U {(X1, . . ., Xn) ---> Y - A}
           If Annderive(X1,A,NC) then
               AnnCov := NC
   end
```

The above algorithm is clearly correct and guaranteed to halt as the number of left sets and the elements of right set are finite. The complexity of the above algorithm is $O(a^2p^2)$.

Example 3.5: The annular cover after Rightreduce() is performed on the annular cover in Example 3.4 is given below:

{ (AB, CD) ---> {}, (EK, H) ---> IJ, (D) ---> EK, (AE) ---> H } The attribute sets EH was extraneous in the first CFD, and so was attribute H in third CFD.

The algorithm that controls the sequence of steps to transform a set of FD's to a reduced annular cover is:

```
Algorithm ReduceAnn( AnnCov )

 Input  :  AnnCov is the Annular Cover.
 Output :  Return reduced Annular cover of AnnCov.
 Body:
    Determi_e&E( F, eF, EF)
    Compound(eF, EF, AnnCov)
    shiftleftset(AnnCov)
    Removeleftattr(AnnCov)
    rightreduce(AnnCov)
    remove CFDs of the form (X1) ---> {}
    end
```

The Complexity of the entire process of transforming a set of FD's to a reduced nonredundant annular cover is $O(max(ap^3, a^2p^2))$.

Example 3.6: The annular cover after ReduceAnn is performed on the set of FDs F and relation scheme R of example 2.6 is given below.

The annular cover below is reduced.

{ (AB, CD) ---> {}, (EK, H) ---> IJ, (D) ---> EK, (AE)---> H }.

The annular cover so obtained may not be unique. This is because of the equivalence classes.

Example 3.7: The reduced annular cover after ReduceAnn is performed on the set of FDs,F and relation scheme R in Example 2.6 may result in the following

annular cover:

{ (AB, CD) ---> {}, (EK, H) ---> IJ, (D) ---> H, (AE) ---> {} } This is because EK and H are equivalent to each other and one can be substituted for the other.

Lemma 3.1: Let X_i be a left set in CFD_x and let A be in X_i then

a) A is shiftable to the right iff X_i -A ---> A.

b) A is extraneous iff (X_i -A) ---> A or CFD is redundant.

proof: part a is obvious.

part b:

Case 1: CFD has more than one left set. Let CFD = (X_1, X_2, X_3, ..., X_n) ---> Y.

If A is extraneous in say X_i then X_i -A becomes a left set which means (X_i-A) ---> X_2

But also X_2 ---> X_i

i.e X_2 ---> A

hence (X_i-A) ---> A.

Case 2: The CFD has only one left set X ---> Y.

assume that A is extraneous in X.

let X-A = X^-

then Annular Cover derives X^----> Y.

If X ---> Y is not redundant then let B Ú Y such that B is not extraneous and X ---> B.

Since A is extraneous then X^- ---> B.

But if X^- -/-> X then $(X^-)^+$ can be computed not using X ---> Y.

Hence X^- ---> B is derived from FDs other than X ---> Y.

But X ---> X^- .

Therefore B is extraneous which contradicts our assumption that it was not.

Hence, if A is extraneous in X then either X-A ---> A or every attribute of Y is extraneous.

Lemma 3.2: The algorithm above produces a reduced annular cover.

Proof: Determi_e&E() finds all the equivalence classes of left sets. Compound() takes the equivalence classes and forms CFDs out of FDs. Shiftleftset() shifts all those left sets which are extraneous on left to right set. Removeleftattr() removes all extraneous attributes of every left set. Whereas rightreduce() removes all extraneous attributes from right set.

Determine the Relation Schemes

After obtaining a reduced annular cover the relation schemes are determined. This is acomplished in two steps:

STEP 2.1 : Find the relation schemes of the database.

STEP 2.2 : Remove redundant relation schemes.

From the construction of the database it appears that the schemes obtained may not be distinct, i.e. some of them can be a subset of the other which is observed in ((Beeri, et al. 1979), (Maier 1983), (Bernstein 1976)). This can be illustrated by the following example.

Example 3.8: Let R = (C, S, Z) and the set of functional dependencies F = { Z ---> C, CS ---> Z }.

The set F is left reduced. Transforming it into annular cover will result in:

(Z) ---> C & (CS) ---> Z

Thus the database consists of the following schemes:

R1(C, Z) with K_{11} = { Z }

R2(C, S, Z) with K_{21} = { CS }

It is obvious that R1 is a subset of R2. Thus, there is no point in decomposing the scheme R into R1 and R2. Here preservation of functional dependencies is desired rather than the Boyce Codd Normal Form, in the line of the observations by (Beeri, et al. 1979).

The database can be constructed by transforming each compound functional dependency of the reduced annular cover into a relation scheme. The algorithm to

obtain schemes of database is as follows:

```
Algorithm  DataBase ( G, DB )

 Input  : G  -  Reduced annular cover.
 Output :  DB -  Set of relation schemes in the database.
           K  -  Two dimensional array where k_ij means j^th key for i^th
                 scheme. i = 1 ... no. of schemes and j = 1 ... num_keys(i).
         output the set of schemes along with contained FDs.
 Body  :
   i := j := m := 1
   for each CFD  in G do
       let the i_th CFD be (X1,  X2, ......,  Xn) ---> Y
       DB(j) := X1 U X2 U .... U Xn U Y
       Reduced Keys for DB(j) are X1, X2, . . . , Xn where n ≥ 1
          K_j1 := X1,  K_j2 := X2, . . . . K_jn := Xn
       F(j) = { X_1 ---> X_2 U Y,  X_2 ---> X_3,  . . . , X_n ---> X_1 }
       m := 1
       found =  false
       While (m< j) and (not found) DO
          if DB(j) ⊆ DB(m) then
            F(m) =  F(j) U F (m)
            found := true
          else  if  DB(m) ⊆ DB(j)  then
                DB(m) = DB(j)
                Reduced Keys of DB(m) = Reduced Keys of DB(j)
                F(m) =  F(m) U F(j)
                found:= true
            else    m := m+1
       if  (not found) then
          j := j + 1
       i := i +1
   endfor
   for i = 1 to no_schemes DO
       print(DB(i))
       print(Reduced_keys(i))
       print( F(i))
   endfor
   end
```

The algorithm is clearly correct and guaranteed to halt as G is finite. The schemes as well as some of the keys for each of the relation are determined in polynomial time. The complexity of this algorithm is clearly the number of

compound functional dependencies which in the worst case is $O(p^2)$.

Note not all reduced keys of schemes are determined. For example :

Example 3.9: Let R(ABCD) be a scheme and let a set of functional dependencies, F, be { AB --->CD, C ---> AE, D ---> BF}.

Resulting decomposition will be

R1(ABCD) K_{11}={ AB }

R2(ACE) K_{21} ={ C }

R3(BDF) K_{31} ={D}.

But R1 has two reduced keys {AB} and {CD}.

The decomposition resulting from the presented approach is not always in optimal DPNF. This is due to the fact that attributes are checked for extraneousness in random order. Consider the following example.

Example 3.10: Let R(ABCD) be relation scheme and a set of functional dependecies, F, be { AB ---> C, B ---> D, D---> B, C ---> BD }.

Two different reduced annular covers can result from F: AN1={(AB) --->C, (B,D) --->{}, (C) --->B} and AN2 = {(AB) ---> C, (B,D) ---> {}, (C) ---> D}.

Decomposition due to AN1 before step 8 will be R1(ABC), R2(BD) and R3(BC). After step 8 it will be:

R1= (ABC) K_{11}={ AB }

R2 = (BD) K_{21} ={B}, K_{22} ={D}

Which is in optimal DPNF. But the decomposition due to AN2 will be R1(ABC) R2(BD) and R3(CD). After step 8 the decomposition will be:

R1(ABC) K_{11}={ AB }

R2(BD) K_{21} ={B}, K_{22} ={D}

R3(CD) K_{31} ={C}.

This decomposition is not in optimal DPNF.

A remark regarding subsets with respect to the superfluous attributes described in (Ling, et al. 1981) is desired. Consider the following example.

Example 3.11: Let R(A, B, C, D, E, F) be a relation scheme and a set of functional dependencies, F, be { AD ---> B, B ---> C, C ---> D, AB ---> E, AC

---> F, E ---> A }.

The above set of functional dependencies is reduced. The traditional approach may generate the following :

R1(A, B, C, D, E, F) K_{11} = {A, B}, K_{12} = {A, C}, K_{13} = {A, D}.

R2(B, C) K_{21} = {B}.

R3(C, D) K_{31} = {C}.

R4 (A, E) K_{41} = {E}.

In (Ling, et al. 1981) it has been shown that C is superfluous attribute in R1 and therefore even if C is removed from R1 all the functional dependecies are preserved. Therefore (Ling, et al. 1981) will obtain the following schemes:

$R1^-$(A, B, D, E, F) K_{11} = {A, B}, K_{12} = {A, D}.

R2(B, C) K_{21} = {B}.

R3(C, D) K_{31} = {C}.

R4(A, E) K_{41} = {E}.

As one can see, removal of R4 will not result in any lost FD because E ---> A is still contained in $R1^-$.

Therefore the method described here will result in the following decomposition.

$R1^-$(A, B, D, E, F) K_{11} = {A, B}, K_{12} = {A, D}.

R2(B, C) K_{21} = {B}.

R3(C, D) K_{31} = {C}.

The method described is also preferred because while it generates lossless decompositon along with no lost FDs, it provides a more optimal database scheme with a smaller number of relation schemes. One does not have to worry about the null values C may take as C is a prime attribute in R1.

The following text will show that the schemes constructed by algorithm MakeDPNF satisfy properties D1, D2, D4.

Theorem 3. 1: Let X ---> A be in F^+ then there exists a set of attributes Y such that X ---> Y and Y ===> A.

Proof: If X does not directly imply A then there exists X_1 such that X ---> X_1 and X_1 ---> A. Similarly if A is transitively depending on X_1 then there exists a X_2 such that X_1 ---> X_2 and X_2 ---> A. If any Xi <---> X then each X_{i-1} <---> X_i which would mean that x ===> A to start with. This process must finish since F^+ is finite, then there is some X_i such that X_i ===> A and X_i -/-> X.

Theorem 3.2: Let Y ===> A then A is one of the attributes in CFD_Y of a reduced annular cover.

Proof: Assume A is not in CFD_Y then A must be in some other CFD say CFD_{Y1} such that Y_1 is in Y^+. If Y_1 Ú Y^+ and Y_1 <-/-> Y then A is transitively dependent on Y which contradicts the assumption that Y ===> A. If Y1 <---> Y then it is a contradiction to the construction of annular cover.

Theorem 3.3: If A is a right attribute of CFD_X then X ===> A.

Proof: Suppose there exists w in F s.t. X ---> W, W -/-> X and W ---> A. Let W^- be s.t. W ---> W^- and W^- ===> A. Thus by theorem 3.2 A is in CFD_{W^-} then it is extraneous in CFD_X which contradicts our construction.

Corollary 3.1: If A is on the right side of some CFD_X then there is no decomposition of CFD_X into CFD_X-A and W ---> A unless W contains a left set of CFD_X.

Proof: It follows from theorem 3.3.

Theorem 3.4: A decomposition of a CFD_X into CFD_X-A and W ---> A will result in a lost FD unless W ---> X.

Proof: For attributes on the right side of CFD_X the assertion follows from corollary 3.1.

If A is member of some left set of CFD_X , say Y then we lose FDs Y ---> X, since Y has no extraneous attributes.

If Y ---> X is not lost then W ---> A is redundant because A would be in some CFD_{W^-} s.t. W^- ===> A. But this means that A could have been removed from CFD_X in our construction.

Theorem 3.5: The schemes so constructed satisfy properties D1, D2, D4.

Proof: Since schemes are constructed by transforming CFDs into schemes all left sets are keys of the scheme and theorem 3.3 shows that no member of the right set is transitively dependent on the left sets.

Due to the fact that reduced annular cover is used to construct schemes it is obvious that all FDs are preserved.

Theorem 3.4 shows that the scheme conctructed out of reduced annular cover cannot be decomposed further without losing FDs except for keybased decomposition which is not desired.

Checking Lossless Join Property

In order to check for lossless join decomposition, one only needs to check if any key of R is embeded in some scheme R_i. It has been proved in (Biskup, et al. 1979) if all functional dependencies are preserved then presence of a key of R is sufficient for lossless join decomposition. The algorithm to test for lossless join decomposition in presence of functional dependencies is as follows:

```
Algorithm  Lossless ( F, DB, A )

 Input  : F  - Reduced cover of functional dependencies.
          DB - Set of relation schemes in the current database.
          A  - Attribute set.
 Output : If lossless then true else false.
 Body   :
    For i = 1 to no_schemes do
        If LinDerives(F,DB[i],R) then
```

```
        return(TRUE)
    return(false)
    end
```

Thus the schemes so constructed are first checked for lossless decompostion, if the decomposition is lossless then decomposition is done, otherwise add a new scheme to make it lossless. Add a key of the original relation scheme as a new scheme, as it makes it lossless decomposition.

Finding a Reduced Key of the Original Relation Scheme

A simple algorithm to compute a reduced key of a relation is as follows:

```
Algorithm   GetKey( R, AnnCov, key)

 Input  :  R          -  Set of all  attributes of the relation scheme.
           AnnCov     -  Reduced annular cover of functional dependencies.
 Process: S           -  A key of the original relation scheme.
           Subkey     -  Union of one member of left sets of each CFD
            URightsets- Union of Right sets of all FDs.
           ULeftsets  - Union of left sets of all FDs.
           Non_Key    - The set of all attributes that appear only on the right.
           Kernel     -  Set of attributes which are essential for any key.
 Output :  Key        - One of the reduced keys of R.
 Body   :
    Non_key := Urightsets - Uleftsets
    Kernel := R - Urightsets
    S := R - Nonkey
    S- := S - Kernel
    for each element A in S- do
        If AnnDerives( S- {A}, R, AnnCov) then
            S := S - A
    Key := S
    end
```

The above algorihm obtains a reduced Key by removing extraneous attributes from key S. Instead of checking for every attribute in S, check only those attributes which are not part of the kernel and also appear in both Uleftsets and Urightset. The complexity of this algorithm is based on the complexity of AnnDerives and is

$O(a^2p)$. This approach of obtaining a reduced key of the original relation is faster than the one proposed in (Osborn, et al. 1978). This is because of the fact that here the reduced cover rather than the original set of functional dependencies are considered. To obtain a reduced key, pass to the above algorithm GetKey(R, AnnCov, key). This is faster because Uleftsets and Urightsets don't have to be computed.

The computations, described above, for Kernel and key are faster than (Osborn, et al. 1978) for two reasons: Getkey() works with a reduced cover, secondly, it tests a smaller number of attributes rather than the whole relation.

Adding New Relation Scheme for Lossless Decomposition

When the database is not lossless, add the reduced key obtained in step 5 as a subscheme to yield a lossless decomposition. The algorithm is as follows:

```
Algorithm  AddScheme (DB, Key)

 Input   :  DB   - Set of essential relation schemes in the database, obtained by
                   algorithm Database.
            Key  -  Reduced Key of the original relation scheme.
 Output :   DB    - modified by adding a new scheme to it.
 Body   :
    no_schemes := no_schemes + 1
    DB(no_schemes) := Key
    K no_schemes,1 := Key
    end
```

This algorithm is clearly correct as it is known that adding a reduced key gives lossless decomposition ((Ullman 1980), pp. 242). However the new scheme added may need some further decomposition.

Lemma 3.3: There are no FDs contained in the scheme added to obtain losslessness of decomposition.

Proof: If scheme consisted of XYZ and Y ---> Z then XY is a reduced key which contradicts our construction of a reduced key.

Corollary 3.2: Since the new scheme does not have any FD contained in it, one cannot decompose it further for lossless decomposition on the basis of FDs. The only decomposition possible for the newly added scheme is on the basis of SMVDs (and of course JVDs which is not a concern here).

Decompositon of New Relation Scheme

The technique used in ((Fagin 1977), (Lien 1981), (Yuan, et al. 1987)) are integrated and modified in order to obtain a DPNF decomposition. (Fagin 1977) defines all the properties of the multivalued dependencies and also 4NF decomposition. (Lien 1981) gives the concept of minimal 4NF covering and also produces schemes which are not redundant. (Yuan, et al. 1987) gives an efficient method to obtain reduced multivalued dependencies and also 4NF covering. However obtaining a minimal 4NF covering is a complex process.

From Corollary 3.2, it follows that the only scheme to be considered for decomposition is the new scheme which was obtained in step 6. Let Q denote the newly added scheme. We consider the set of multivalued dependencies, provided by database designer, for this new scheme, Q. Let M be a set of multivalued dependencies for the scheme Q. The algorithm below is based on (Yuan, et al. 1987). This algorithm completes the construction of DPNF database.

```
Algorithm DPNF( Q,  M,  no, S )

 Input :  Q  -  Scheme under consideration.
          M  - A cover of multivalued dependencies.
 Output : no  - Number of schemes added.
          S   - Schemes modified i.e. S[1], S[2], .. S[no].
 Body:
    j := 1
    S[j] := Q
    for each MVD X --->---> W in M do
        set := X U W
        for i = 1 to j do
            if ( set ⊂ S[i] ) then
              temp := S[i]
              S[i] := set
```

```
            j := j + 1
            S[j] := temp - W
          endif
        endfor
    endfor
    no := j
    end
```

It has been shown in (Yuan, et al. 1987) that the MVD covering produces distinct and reduced database schemes. We now present an algorithm to organize the schemes as follows:

```
Algorithm Organize(DB, M, no)

 Input : DB   :  Set of schemes i.e. DB(1), ... DB(no).
         M    :  A cover of multivalued dependencies.
         no   :  Number of schemes.
 Output: DB   :  Schemes modified.
         no   :  Number of schemes modified.
 Body:
    RemoveSmvds(DB(no), M, count, S)
    DB(no) := S[1]
    for j = 2 to count do
        DB(no + j - 1) := S[j]
    endfor
    no := no + count - 1
    end
```

The above algorithm is guaranteed to halt and is clearly correct. However we find that after decomposition we may get schemes which are subsets of other schemes. This is shown in the following example.

Example 3.12: Let R = (A, B, C, D) and F = { AB ---> C, D ---> C }.

After transforming F to a reduced annular cover, we obtain the following schemes:

R1 (A, B, C) with K_{11} = {A, B}

R2 (D, C) with K_{21} = {D}

The above schemes satisfy properties D1, D2 and D4. However the schemes above do not have lossless decomposition. This is shown below.

r	r1	r2
a1 b1 c1 d1	a1 b1 c1	c1 d1
a2 b1 c1 d2	a2 b1 c1	c1 d2

Clearly r1 |X| r2 ≠ r. So add a key ABD as a new scheme and which results in the following schemes:

R1 (A, B, C) with K_{11} = {A, B}

R2 (D, C) with K_{21} = {D}

R3 (A, B, D) with K31 = {A, B, D}

let A --->---> B hold in R3. This will result in the following schemes:

R1 (A, B, C) with K_{11} = {A, B}

R2 (D, C) with K_{21} = {D}

R3 (A, B) with K_{31} = {AB}

R4 (A, D) with K_{41} = {AD}

R3 is a subset of R1 and hence there is no point in considering R3. Discarding R3 and reorganizing will result in the following schemes which are in DPNF and also satisfy the properties D1 ... D4.

R1 (A, B, C) with K_{11} = {A, B}

R2 (D, C) with K_{21} = {D}

R3 (A, D) with K_{31} = {AD}.

Hence, finally remove the schemes which are subsets and obtain the schemes which are in DPNF. DPNF database schemes are obtained in polynomial time.

Corollary 3.3: The schemes obtained by algorithm MakeDPNF are in DPNF.

Proof: Theorem 3.5 shows that decomposition resulting by algorithm MakeDPNF satisfy properties D1, D2 and D4.

It has been shown in (Biskup, et al. 1979) that if all FDs are preserved then the presence of a Key of R is sufficient for lossless join decomposition.

Since MakeDPNF preserves all FDs and the presence of at least one reduced key of R is guaranteed, therefore MakeDPNF also satisfies property D3.

Hence the set of schemes obtained by MakeDPNF is always in DPNF.

CHAPTER IV

DESIGN ENHANCEMENTS

The database constructed above is neither minimal nor optimal. Presented below is an algorithm to obtain a more optimal design which is slower than the above technique, but still good in terms of complexity. The near optimal design comes from the fact that annular cover is not unique (refer example 3.10). Also, for computing the key the order of discarding the attributes makes it near optimal, rather than optimal. In addition, as was mentioned before, the algorithm MakeDPNF does not find all of the keys of each scheme added.

The top down organization to obtain such a design is as follows:

FOR i = 1 to number of input schemes DO

STEP 0 : Read in a given relation scheme U_i and its functional dependencies, F.

STEP 1: Compute the reduced annular cover of the set of functional dependencies.

STEP 2 : Determine a set of relation schemes of the database satisfying properties D1, D2, D4.

STEP 3 : Check whether any of the left sets is a key of U_i.

STEP 4 : If yes, then do step 8 or else do step 5.

STEP 5 : Determine a minimal key, K, of the original scheme U_i.

STEP 6 : Add a new scheme composed of attributes of the minimal key.

STEP 7: Remove strict multivalued dependencies from K to obtain DPNF decomposition.

STEP 8: Remove redundant schemes.

STEP 9: Find all reduced keys of U_i and of all the schemes.

Some steps from above are the same as the previous design.

Computing All the Keys for each Scheme

Sometimes, not all of the reduced keys for the subschemes are found. This can be illustrated by the example 3.8 considered before.

The set F is left reduced. Transforming F into annular cover we obtain:

(Z) ---> C & (CS) ---> Z

Thus the database consists of the following schemes:

R1(C, S, Z) with K_{11} = { CS }

However, ZS is also a reduced key for subscheme R2 which is not obtained.

In order to obtain all of the reduced keys for each subscheme, similar concepts from the algorithm of (Osborn, et al. 1978) are applied. The algorithm is as follows:

```
Algorithm AllKeys(F, DB, K)

 Input :  F  - Set of functional dependencies which are reduced.
          DB - Set of essential relation schemes for the data base.
          K  - Two dimensional array and each array is a set of attributes where
               k_ij means j^th reduced key for i^th scheme. 1 ≤ i ≤ no. of schemes
               and 1 ≤ j ≤ num_keys(i).
 Output : K modified for each scheme.
 Body:
    for i = 1 to no_schemes do
        n := num_keys(i)
        K_S := { K_i1, .... K_in }
        for each element say Key in Ks do
            for each FD X ---> Y in F do
              if ((Key - Y) ≠ Key) and (X ⊆ DB(i))) then
                  temp := X U ( Key - Y )
                  dup := false
                  j := 1
                  While ((not dup) and (j <= n )) do
                     if (K_ij ⊆ temp) then dup := true
                     j := j + 1
                  end
                  if (not dup) then
                     call GetKey( DB(i), F, temp)
                     n := n + 1
```

```
                K_in := temp
                K_S := K_S U {K_in}
            end
        end
    end
    num_keys(i) := n
end
```

This algorithm is correct and its complexity is determined to be bounded by a polynomial in the number of keys, functional dependencies and number of attributes of the relation scheme (Osborn, et al. 1978), for each of the subschemes under consideration. So all of the candidate keys for each subscheme are discovered.

CHAPTER V

COMPLEXITY AND EXAMPLE

The complexity of obtaining the DPNF database is $O(a^2p^4)$ where a is the number of attributes of original relation scheme and p is the number of functional dependencies. The algorithm presented here is compact and considers all of the properties for the construction of database.

Trace the whole algorithm with an example.

Example 5.1: Let R(A, B, C, D, E, H, I, J, K, L, M, N, P) and set of functional dependencies F={ ABM ---> CDEJKL, B ---> M, D ---> BJ, N ---> PH, NH ---> I, P ---> H, P ---> I, JKL ---> ABM }.

First step in finding annular cover is to compute equivalence classes, e_F and E_F. The e_F and E_F classes for F are:

e_F= { e_F(ABM)= e_F(JKL)={ABM, JKL},

e_F(B)={B},

e_F(D)={D},

e_F(N)= e_F(NH)={N, NH},

e_F(P)={P} }

E_F={ E_F(ABM)= E_F(JKL)={ ABM ---> CDEJKL, JKL --->ABM},

E_F(B)={ B ---> M},

E_F(D)={ D ---> BJ},

E_F(N)= E_F(NH)={ N ---> PH, NH ---> I },

E_F(P)={ P ---> I, P ---> H } }

Once equivalence classes are computed, then an annular cover can be formed for F. Let G be annular cover of F.

G={ (ABM, JKL) ---> CDE, (B) ---> M, (D) ---> BJ, (N, NH) ---> PI, (P) ---> HI }

After shifting shiftable left sets to the right, we obtain the following annular cover:

G={ (ABM, JKL) ---> CDE, (B) ---> M, (D) ---> BJ, (N) ---> HIP, (P) ---> HI }

We find that M is extraneous in left set ABM of CFD (ABM, JKL) ---> CDE. After discarding all extraneous attributes in members of left sets of all CFDs we get the following:

G={ (AB, JKL) ---> CDE, (B) ---> M, (D) ---> BJ, (N) ---> HIP, (P) ---> HI }

The final step in obtaining reduced annular cover is to remove all extraneous attributes from right sets and discard CFDs of the form (x) ---> {}. It is obvious that HI can be discarded from CFD (N) ---> HIP without altering the closure of F or G. Therefore, the reduced annular obtained is :

G={ (AB, JKL) ---> CDE, (B) ---> M, (D) ---> BJ, (N) ---> P, (P) ---> HI }

Once a reduced annular cover is computed, then it can be used to generate decompositions by translating all CFDs into relation schemes. After transforming CFDs into relation schemes we obtain the following decomposition:

R1(ABCDEJKL) K_{11}={AB}, K_{12}={JKL}

R2(BM) K_{21}={B}

R3(BDJ) K_{31}={D}

R4(NP) K_{41}={N}

R5(PHI) K_{51}={P}

But R3 is a subset of R1 i.e R3 is redundant, therefore, discarding it will not result in any lost FD. Hence, the new decomposition is:

R1(ABCDEJKL) K_{11}={AB}, K_{12}={JKL}

R2(BM) K_{21}={B}

R3(NP) K_{31}={N}

R4(PHI) K_{41}={P}

The decomposition above is in 3NF and also satisfies propeties D1, D2, and D4. Since neither of the decomposed schemes above contain a key of R, it is not a lossless decomposition. We need to add a key of R to the decomposition as a scheme. The reduced key of R found by Getkey algorithm is {JKLN}. The resulting decomposition is given below:

R1(ABCDEJKL) K_{11}={AB}, K_{12}={JKL}

R2(BM) K_{21}={B}

R3(NP) K_{31}={N}

R4(PHI) K_{41}={P}

R5(JKLN) K_{51}={JKLN}

Suppose J --->---> L holds in R5 then R5 needs to be further decomposed and the result will be:

R1(ABCDEJKL) K_{11}={AB}, K_{12}={JKL}

R2(BM) K_{21}={B}

R3(NP) K_{31}={N}

R4(PHI) K_{41}={P}

R5(JKN) K_{51}={JKN}

R6(JL) K_{61}={JL}

But R6 is redundant; therefore, the new and final decomposition will be the following:

R1(ABCDEJKL) K_{11}={AB}, K_{12}={JKL}

R2(BM) K_{21}={B}

R3(NP) K_{31}={N}

R4(PHI) K_{41}={P}

R5(JKN) K_{51}={JKN}

The above decomposition is in DPNF and satisfies properties D1, D2, D3 and D4.

CHAPTER VI

CONCLUSION

The goal was to efficiently construct a database satisfying all the properties D1, D2, D3 and D4 in chapter II. A new normal form, Dependency Preserving Normal Form, which is superior to 3NF is introduced. Only functional and multivalued dependencies are considered for such a construction of the database. The algorithm described produces DPNF and is faster and simpler than the earlier approaches to find 3NF, BCNF or 4NF. It also produces a more optimal design.

The new normal form DPNF, which is presented here, is stronger than 3NF but weaker than 4NF. DPNF guarantees a lossless decomposition. All decompositions of DPNF are in 3NF. DPNF preserves all functional dependencies. It yields a better decomposition by discarding some of the redundant schemes and also by using CFDs as a way of combining two schemes whose keys are equivalent. DPNF decomposition is obtained in polynomial time.

The algorithm to produce DPNF decomposition uses a cover of FDs to produce a reduced annular cover consisting of compound functional dependencies. The approach to find the reduced annular cover is faster than the traditional methods. It uses Linderives and Annderives which are faster than Iinclosure. Instead of computing left reduced cover, right reduced cover, reduced cover, and then forming the annular cover, the presented approach uses the original cover of FDs to form annular cover. The database is constructed by transforming each compound functional dependency of the reduced annular cover into a relation scheme, with each left set of that compound functional dependency being a key for that scheme. It is shown with examples that sometimes all of the keys for some subschemes are not discovered. Some improvements are made to the algorithm given in (Osborn, et al. 1978) to find all of the keys for each subscheme of the database. However it has been proved (Osborn, et al. 1978) that to determine all keys of cardinality m of a

relation scheme is NP-Complete.

It also removes schemes that happen to be subsets of other schemes. The redundancy happens because of the fact that some functional dependencies take part in more than one relation. The schemes whose superset exists are redundant and therefore are not needed in the decomposition.

The schemes produced by this approach may be in Boyce Codd Normal form. However it has been shown in (Beeri, et al. 1979) that it is NP-Complete just to determine whether a relation scheme is in Boyce Codd Normal form. It is well known that Boyce Codd Normal form does not always preserve functional dependencies. For this reason DPNF is preferred over BCNF.

If the subschemes do not have the lossless join property then it has been shown in (Biskup, et al. 1979) that a key is needed. Two approaches are used to find a key of the original relation scheme. One method is faster but it does not guarantee optimal DPNF. The second approach guarantees the minimal key but it is slower. The fact is, if we have a minimal key then we can have a relation scheme with fewer attributes.

The algorithm outputs every scheme with FDs that are contained in it along with some of the corresponding keys.

The document notes earlier drawbacks and compares the presented algorithm with other approaches. The complexities at each stage are also determined. Also, for simplicity and clarity, an example has been traced through the whole algorithm.

APPENDIX

IMPLEMENTATION OF THE ALGORITHM

The program to obtain a DPNF decomposition is given below. If a new scheme is added to the decomposition to obtain a lossless join property, then this program will not decompose on the basis of MVDs.

```
(*************************************************************************
 *  This program takes relation scheme as input which may or may not be     *
 *  Universal.  It is also supplied with functional dependencies which      *
 *  specify the relation among the attributes. The program first determines *
 *  equivalence classes i.e. all the f.ds whose left sides are equivalent.  *
 *  Now we convert each set of f.d with equivalent left sets into compound  *
 *  functional dependencies. This collection of CFDs forms an annular cover.*
 *  The annular cover is then converted into reduced annular cover. Each    *
 *  cfd is then converted into a relation scheme with all of cfd attributes *
 *  being attributes of the relation scheme and left sets form reduced keys.*
 *  This approach is due to synthesis. So we obtain a set of relation scheme*
 *  from original scheme to start with. Some of the schemes may be subset of*
 *  the other schemes therefore those schemes which have superset are not   *
 *  needed. These schemes are discarded.  The schemes obtained may not have *
 *  lossless join property, therefore a check is made to see if schemes have*
 *  this property. If it is then we are done otherwise we determine one of  *
 *  reduced keys of the original scheme. The reduced key is added to the set*
 *  of decomposed schemes as a new scheme.                                  *
 *  Finally the decomposed schemes are printed along with some of their     *
 *  reduced keys and all of the FDs contained in them.                      *
 *************************************************************************)

PROGRAM cf(input, inp, output, out);

(*************************************************************************
 *                          Constant  declarations                       *
 *************************************************************************)
CONST

    mx_attr   =   20;   { Maximum number of attributes  }
    mx_length =   20;   { Maximum length of each attribute }
    mx_fds    =   25;   { Maximum number of fds   }

(*************************************************************************
 *                          Type  declarations                           *
 *************************************************************************)
```

```
TYPE
   marker_set =   set of ' ' .. '~';   { Keeping track of endmarker }
   attr_set   =   set of 0 .. mx_attr; { Set of attributes }
   fd_set     =   set of 1 .. mx_fds;          { Set of f.ds }
   nametp     =   string[mx_length];           { attribute name }
   attr_ar    =   array[1..mx_attr] of nametp; { array of attribute names }
   fd_rec     =   RECORD                { an f.d with left containing all the }
                      left,             { left attributes of it and right the }
                      right : attr_set; { right attributes of it. }
                  END;
   functional =   array[1..mx_fds] of fd_rec;  { array of f.ds }
   right_ar   =   array[1..mx_fds]  of attr_set;
   equi_set   =   array[1..mx_fds] of fd_set;  { contains f.ds whose left    }
                                               { sides are equivalent }
   cfd_d      =   RECORD
                      card : integer;
                      Lset : right_ar;
                      Rset : attr_set;
                  END;
   annular    =   array[1..mx_fds] of cfd_d;
   count_d    =   array[1..mx_fds, 1..mx_fds] of integer;
   list_d     =   array[1..mx_attr, 1..mx_fds] of integer;
   int_ar     =   array[1..mx_attr] of integer;
   key_ar     =   array[1..mx_fds] of integer;

(*****************************************************************************
 *                          Variable  declarations                           *
 *****************************************************************************)
VAR

  inp, out     :   Text;       {  input file and output file }
  inpf_name,                   { Holds name of input file.   }
  outf_name    :   Nametp;     { Holds name of output file.  }
  no_fd        :   Integer;    {  number of f.ds read        }
  attrb        :   attr_ar;
  fd           :   functional;
  pf,                          { array for holding pointer to contained cfds. }
  equi         :   equi_set;
  e_left,
  e_right,
  clos,
  rel          :   right_ar;   { an array of relation schemes }
  R,
  rkey         :   attr_set;
  no_schemes,
  ecounter,
  probl_no     :   integer; { an indicator keeping track of different problem }
  keys         :   key_ar;     { keeps a pointer to cfd that holds reduced keys.}
  count        :   count_d;    { count of attributes not counted for in cfd }
  list1,                       { list used in annderive for pointer to cfds }
  list2        :   list_d;
```

```
  cfd          :   annular;   { array for annular cover}

(*************************************************************************
 *                          Initialize All the variables                 *
 *************************************************************************)
procedure initialize_all( var F:functional; var attr:attr_ar; var equi:
                          equi_set; var e_left, e_right, db:right_ar);

{   This procedure initializes all the variables used, to empty   }

VAR
    x  :  integer;

BEGIN
    for x := 1 to mx_fds do
    BEGIN
        F[x].left  := [];
        F[x].right := [];
        equi[x]    := [];
        e_left[x]  := [];
        e_right[x] := [];
        rel[x]      := [];
    END;
    for x := 1 to mx_attr do
       attr[x] := '';
END;

(*************************************************************************
 *                      Print Attribute Set                              *
 *************************************************************************)
procedure print_at_set( var out : text; a_set : attr_set; attr : attr_ar );

{  This procedure converts each integer into appropriate attribute name from
   the set which is passed. It then prints the attribute.                 }

VAR
    x  :  integer;

BEGIN
    if (a_set <> []) then
        for x := 1 to mx_attr do
           if ( x in a_set ) then
           BEGIN
               write(out, attr[x]);
               write(out, '  ');
           END;
END;

(*************************************************************************
 *                      Print Functional Dependency                      *
 *************************************************************************)
```

```
procedure print_fd_set( var out : text; f : functional; no_fd : integer;
                        attr : attr_ar);

{ This procedure prints each f.d. from the set of f.ds in the proper order   }

VAR
   x, cou  : integer;

BEGIN
    writeln(out);
    writeln(out, ' ':10, ' Functional  Dependencies ( F.D. ) are : ');
    writeln(out, ' ':10, ' ----------  ------------ -------- ---');
    cou := 1;
    for x := 1 to no_fd do
       if ( f[x].left <> [] ) then
       BEGIN
           write(out, ' ':10);
           if ( cou <= 9 ) then write(out, ' ');
           write(out, cou, ') ');
           print_at_set(out, f[x].left, attr);
           write(out, ' ---> ');
           print_at_set(out, f[x].right, attr);
           writeln(out); writeln(out);
           cou := cou + 1;
       END;
END;

(*****************************************************************************
 *                              Print  C.f.d.                                *
 *****************************************************************************)
procedure pr_cfds(var out:text; cfd:annular; ecounter:integer; attrb:attr_ar);

{  This procedure prints compound functional dependency by checking the      }
{  equivalence classes.                                                      }

VAR
    i, x : integer;

BEGIN
    i := 1;
    writeln(out);
    writeln(out,' ':10, ' Compound Functional Dependencies are :');
    writeln(out,' ':10, ' -------- ---------- ------------ ---');
    for i := 1 to ecounter do
    BEGIN
        writeln(out); write(out, ' ':10);
        if ( i <= 9 ) then write(out, ' ');
        write(out, i, ')  ( ');
        for x := 1 to cfd[i].card do
        BEGIN
            print_at_set(out, cfd[i].lset[x], attrb);
```

```
            if ( x < cfd[i].card ) then write(out,', ')
            else write(out, ')');
        END;
        write(out, ' ---> ');
        if (cfd[i].rset <> []) then
             print_at_set(out, cfd[i].rset, attrb)
        else
             write(out,'{ }');
        writeln(out);
    END;
END;

(*************************************************************************
 *                          Heading for the title                        *
 *************************************************************************)
procedure heading(var out:text; s:nametp; var probl_no: integer);

{    This procedure prints the title with name passed as a parameter.      }

BEGIN
    writeln(out, chr(27), chr(12));
    writeln(out); writeln(out);
    writeln(out,' ':20, 'The Input File is : ', s);
    writeln(out,' ':20, '------------------ ');
    writeln(out); writeln(out);
    writeln(out, ' ':20, 'Data Set No. : ',probl_no);
    writeln(out, ' ':20, '-----------');
    probl_no := probl_no + 1;
END;

(*************************************************************************
 *                    Read the file names for  input and output          *
 *************************************************************************)
procedure read_file_name(var infil, outfil: text; var i_name, o_name: nametp);

{ This procedure returns the file variable used for both reading and writing }
{ It also checks the error condition like whether file exists or not.       }

VAR
   ch        : char;
   correct   : boolean;

BEGIN
    clrscr;
    writeln; writeln; correct := false;
    while (not correct)
    BEGIN
        writeln;
        write(' Enter the name of file containing fds : ');
        read(i_name); writeln;
        assign(infil, i_name);
```

```
        {$I-} reset(infil) {$I+};
        if ( IORESULT  = 0 ) then correct := true
        else  writeln('File cannot be opened ... Try again ...');
    END;
    writeln;
    write(' Enter the name of output file : ');
    read(o_name);
    assign(outfil, o_name);
    rewrite(outfil);
END;

(*************************************************************************
 *                            X Closure                                  *
 *************************************************************************)
procedure xclosure( F:functional; no_fds:integer; left:attr_set;
                    var lclosure:attr_set);

{       This procedure computes the closure of set of attributes.        }

VAR
    oldc, newc   :  attr_set;
    x            :  integer;

BEGIN
    oldc := [];
    newc := left;
    while (newc <> oldc) do
    BEGIN
        oldc := newc;
        for x := 1 to no_fds do
            if ( F[x].left <> [] ) then
                if ( newc >= F[x].left ) then newc := newc + F[x].right;
    END;
    lclosure := newc;
END;

(*************************************************************************
 *                          Derives                                      *
 *************************************************************************)
function derives(F:functional; no_fds:integer; left, right:attr_set):boolean;

{    This function returns true if left derives right otherwise false    }

VAR
   lclosure : attr_set;

BEGIN
    xclosure( F, no_fds, left, lclosure);
    if ( right <= lclosure ) then derives := true
    else derives := false;
END;
```

```
(*************************************************************************
 *                              Equivalent                               *
 *************************************************************************)
function equivalent( F:functional; no_fds:integer; x, y:attr_set ) : boolean;

{  A function which returns true if two sets are equivalent otherwise false. }

BEGIN
    equivalent := false;
    if (derives(F, no_fds, x, y) and derives(F, no_fds, y, x)) then
       equivalent := true;
END;

(*************************************************************************
 *                            Form Annular                               *
 *************************************************************************)
procedure form_annular( F:functional; no_fds:integer; var equi:equi_set;
                        var e_left, e_right:right_ar; var ecounter :integer);

{  This procedure determines all the equivalence classes of the left sides   }
{ and using these classes it forms annular cover.                            }

VAR
    cnt, x, y  : integer;
    ebuf           : fd_set;

BEGIN
    ebuf := [];  ecounter := 0;
    for x := 1 to mx_fds do
    BEGIN
        equi[x]    := [];
        e_left[x]  := [];
        e_right[x] := [];
    END;
    for x := 1 to no_fds do
        if (F[x].left <> []) then
            if not ( x in ebuf ) then
            BEGIN
                cnt := 1;
                ecounter := ecounter + 1;
                cfd[ecounter].lset[cnt] := F[x].left;
                equi[ecounter] := [x];
                ebuf := ebuf + [x];
                e_left[ecounter]  := F[x].left;
                e_right[ecounter] := F[x].right;
                for y := x + 1 to no_fds do
                    if (F[y].left <> []) and (not (y in ebuf)) then
                       if (equivalent(F,no_fds,F[x].left,F[y].left)) then
                       BEGIN
                           cnt := cnt+ 1;
```

```
                        cfd[ecounter].lset[cnt] := F[y].left;
                        equi[ecounter] := equi[ecounter] + [y];
                        ebuf := ebuf + [y];
                        e_left[ecounter] := e_left[ecounter] + F[y].left;
                        e_right[ecounter] := e_right[ecounter]+ F[y].right;
                    END;
                e_right[ecounter] := e_right[ecounter] - e_left[ecounter];
                cfd[ecounter].rset := e_right[ecounter];
                cfd[ecounter].lset[cnt+1] := [];
                cfd[ecounter].card := cnt;
            END;
END;

(*************************************************************************
 *                          Put In List                                  *
 *************************************************************************)
procedure putinlist(var l1, l2 :list_d; A, x, y :integer);

VAR
     i : integer;

BEGIN
     i := 1;
     while (l1[A,i] <> 0) do
         i := i+1;
     l1[A,i] := x; l1[A, i+1] := 0;
     l2[A,i] := y; l2[A, i+1] := 0;
END;

(*************************************************************************
 *                          Ann Derives                                  *
 *************************************************************************)
function Annderives( cfd: annular; left, right : attr_set) : boolean;

{       This returns true if left derives right using annular cover of cfds. }

VAR
    der                 : boolean;
    A, c, i, j, r, x, y :  integer;
    newc, oldc, ext     : attr_set;

BEGIN
    der := false;
    for x := 1 to mx_attr do
       list1[X,1] := 0;
    if ( right <= left) then
       der := true
    else
       for x := 1 to ecounter do
          With cfd[x] do
          BEGIN
```

```
                clos[x] := rset;  i:= 1;    j:= 1;
                while(i <= card) do
                BEGIN
                    while (lset[j] =[]) do j:= j+1;
                    count[x,j] := 0; clos[x] := clos[x] + lset[j];
                    for A := 1 to mx_attr do
                        if (A in lset[j]) then
                        BEGIN
                            count[x,j] := count[x,j] + 1;
                            putinlist(list1, list2, A, x, j);
                        END;
                    i := i +1;
                    j := j +1;
                END;
            END;
        newc := left;
        if der then
           oldc := []
        else
           oldc := left;
        while (oldc <> []) do
        BEGIN
            for A := 1 to mx_attr do
               if ( A in oldc)   then
               BEGIN
                   i := 1;
                   while (list1[A,i] <> 0) do
                   BEGIN
                       c := list1[A,i];  r := list2[A,i];
                       count[c,r]:= count[c,r] -1;
                       if (count[c,r] = 0) then
                       BEGIN
                           ext := clos[c] - newc;
                           newc := newc + ext;
                           oldc := oldc + ext;
                           if ( right <= newc) then
                           BEGIN
                              der := true;
                              oldc := [];
                           END;
                       END;
                       i := i +1;
                   END;
                   oldc := oldc -[A];
               END;
        END;
        annderives := der;
END;

(*****************************************************************************
 *                          Empty Cfd                                        *
```

```
 ***************************************************************************)
procedure Empty_Cfd(var cf: cfd_d);

{       This procedure empties all entries of the cfd.                      }

VAR
    i, j        :  integer;

BEGIN
    i := 1; j := 1;
    while (i <= cf.card) do
    BEGIN
        if (cf.lset[j] <> []) then
        BEGIN
            cf.lset[j] := [];
            i := i +1;
        END;
        j := j +1;
    END;
    cf.rset :=[];
    cf.card := 0;
END;

(***************************************************************************
 *                          Move Cfd                                       *
 ***************************************************************************)
procedure Move_Cfd( var cf1, cf2: cfd_d);

{       This procedure moves a cfd from one location to other. }

VAR
    i, j        :  integer;

BEGIN
    i := 1; j := 1;
    while (i <= cf1.card) do
    BEGIN
        if (cf1.lset[j] <> []) then
        BEGIN
            cf2.lset[j] := cf1.lset[j];
            cf1.lset[j] := [];
            i := i +1;
        END;
        j := j +1;
    END;
    cf2.rset := cf1.rset;  cf1.rset :=[];
    cf2.card := cf1.card;  cf1.card := 0;
END;

(***************************************************************************
 *                          Get Right Set                                  *
```

```
 ***************************************************************************)
procedure getrightset( x : integer; temp: attr_set; Var final: attr_set);

{   This function finds the right set of a given cfd after one of the left   }
{   sets is removed or changed.                                              }
VAR
    i, j  : integer;
    left  : attr_set;

BEGIN
    with cfd[x] do
    BEGIN
        i := 1; j := 1;
        while (i <= card) do
        BEGIN
            if (lset[j] <> []) then
            BEGIN
                 i:= i +1;
                 left := left + lset[j];
            END;
            j:= j +1;
        END;
        temp := rset + temp;
        final := temp - left;
    END;
END;

(***************************************************************************
 *                          ArrangeLSets                                   *
 ***************************************************************************)
procedure arrangelsets( var cfd: annular; num:integer; cnt:integer);

{ This procedure puts the left sets of cfds in sequence.          }

VAR
    i, j    : integer;

BEGIN
    i := 1; j :=1;
    while (i <= cnt) do
    BEGIN
        while(cfd[num].lset[j] = []) do   j:= j+1;
        if( j > i) then
        BEGIN
            cfd[num].lset[i] := cfd[num].lset[j];
            cfd[num].lset[j] := [];
        END;
        i := i +1;
        j := j +1;
    END;
    cfd[num].card := cnt;
```

```
END;

(*************************************************************************
 *                          Shift Left Sets                              *
 *************************************************************************)
procedure shift_left_sets(Var cfd: annular; Var ecounter: integer);

{      This procedure shifts extraneous left sets to the rigt set.        }

VAR
    x, cnt, i, j    : integer;
    temp            : attr_set;

BEGIN
    for x := 1 to ecounter do
       if cfd[x].card > 1 then
         With cfd[x] do
         BEGIN
             cnt := card; i :=2;
             while (i <= cnt) and (card > 1) do
             BEGIN
                 temp := lset[i];  lset[i] := []; card := card -1;
                 if annderives(cfd,temp,lset[1]) then
                       getrightset(X,temp, rset)
                 else
                   BEGIN
                       card := card +1;
                       lset[i] := temp;
                   END;
                 i := i +1;
             END;
             if (card > 1) then
             BEGIN
                 temp := lset[1];  i :=2;
                 lset[1] := []; card := card -1;
                 while ( lset[i] = []) do i:= i +1;
                 if annderives(cfd,temp,lset[i]) then
                       getrightset(X, temp, rset)
                 else
                   BEGIN
                       card := card +1;
                       lset[1] := temp;
                   END;
             END;
             if ( card < cnt) then arrangelsets(cfd,X,card);
         END;
END;


(*************************************************************************
 *                          Remove Left Attrs                            *
```

```
 ***************************************************************************)
procedure Remove_left_attrs(Var cfd: annular; Var ecounter: integer);

{     This procedure discards extraneous attributes in left sets. }

VAR
    x, y, z      : integer;

BEGIN
    for x := 1 to ecounter do
       with cfd[x] do
          for y := 1 to card do
             for Z:= 1 to mx_attr do
                if (z in lset[y]) then
                   if annderives(cfd,lset[y]-[z], [z]) then
                       lset[y] := lset[y] - [z];
END;

(***************************************************************************
 *                              Right Reduce                               *
 ***************************************************************************)
procedure Right_reduce(Var cfd: annular; Var ecounter: integer);

{    This procedure discards the extraneous attributes from the right sets.  }

VAR
    i, x, z      : integer;

BEGIN
    for x := 1 to ecounter do
       for z := 1 to mx_attr do
           if ( z in cfd[x].rset) then
           BEGIN
               cfd[x].rset := cfd[x].rset - [z];
               if (not annderives(cfd,cfd[x].lset[1], [z])) then
                    cfd[x].rset := cfd[x].rset + [z];
           END;
END;

(***************************************************************************
 *                              Reduce Annular                             *
 ***************************************************************************)
procedure Reduce_Annular(var out: text; var cfd: annular; var ecounter:integer;
                         F: functional; no_fds: integer; var equi: equi_set;
                         var e_left, e_right: right_ar);

{   This procedure computes the reduced annular cover from fds.              }

VAR
    x, cnt, i, j      : integer;
```

```
BEGIN
    form_annular( F, no_fds, equi, e_left, e_right, ecounter);
    shift_left_sets(cfd, ecounter);
    remove_left_attrs(cfd, ecounter);
    right_reduce(cfd, ecounter);
    cnt := 0;
    for x:= 1 to ecounter do
       if (cfd[x].card <= 1) and (cfd[x].rset = []) then
          empty_cfd(cfd[x])
       else
          cnt := cnt +1;
    ecounter := cnt;   i:= 1;   j:= 1;
    while (i <= cnt) do
    BEGIN
        while(cfd[j].card = 0) do   j:= j+1;
        if( j > i) then
           move_cfd(cfd[j], cfd[i]);
        i := i +1;
        j := j +1;
    END;
    pr_cfds(out,cfd,ecounter, attrb);
END;

(*************************************************************************
 *                              Reduced  Key                             *
 *************************************************************************)
procedure reduced_key(var key: attr_set; cfd: annular;
                          R: attr_set; ecounter:integer);

{ This procedure determines the primary key by removing an attribute at a    }
{ time (if it can) from a key.                                               }

VAR
    A,  x,  i          : integer;
    Uright, Uleft, p,
    ronly, kernel, S   : attr_set;

BEGIN
    for x := 1 to ecounter do
       with cfd[x] do
       BEGIN
           Uright := Uright + rset;
           if (card = 1) then Uleft := Uleft + lset[1]
           else
               for i := 1 to card do
               BEGIN
                   Uleft := Uleft + lset[i];
                   Uright := Uright + lset[i];
               END;
       END;
    ronly := Uright - Uleft;
```

```
    kernel := R - Uright;
    S := R - ronly;
    P := s - kernel;
    for A := 1 to mx_attr do
       if ( A in P) then
          if annderives( cfd, S - [A], R) then
             S := S - [A];
    Key := S;
END;

(*************************************************************************
 *                              Lossless                                 *
 *************************************************************************)
function lossless(rel : right_ar; R: attr_set; no_schemes:integer):boolean;

{ The function returns true value if decomposition has lossless join property.}
VAR
      x  : integer;
BEGIN
   lossless := false;
   for x := 1 to no_schemes do
       if annderives(cfd, rel[x], R) then lossless := true;
 END;

(*************************************************************************
 *                            Print Contained Fds                        *
 *************************************************************************)
procedure pr_cont_fds(var out: text; cfd: annular; cvr: fd_set; atr: attr_ar);

{ This procedure prints fds that are contained in a scheme. }

VAR
       x, y   : integer;
BEGIN
    for x:= 1 to ecounter do
       if  (x in cvr) then
          with cfd[x] do
          BEGIN
              if (card > 1) then
                 for y := 2 to card do
                 BEGIN
                     write(out,' ':15);
                     print_at_set(out,lset[y], atr);
                     write(out,' ---> ');
                     print_at_set(out,lset[y-1], atr);
                     writeln(out);
                 END;
              write(out,' ':15);
              print_at_set(out,lset[1], atr);
              write(out,' ---> ');
              if (card=1) then
```

```
                print_at_set(out, rset, atr)
             else
                print_at_set(out, lset[card] + rset, atr);
             writeln(out);
          END;
END;

(*************************************************************************
 *                          Print  Database                              *
 *************************************************************************)
procedure print_db(var out:text; db:right_ar; atr:attr_ar; key:attr_set;
                   R:attr_set; keys: key_ar);

{ This procedure  prints the database with keys and also the primary key     }
{ for the original relation.                                                 }

VAR
    i, j : integer;

BEGIN
    i := 1;
    writeln(out); writeln(out);
    writeln(out, ' ':10, ' Relational  Database');
    writeln(out, ' ':10, ' ----------  --------');
    writeln(out); write(out,' ':10, 'R',' ( ');
    print_at_set(out, R, atr);
    writeln(out,')');
    write(out,' ':10, 'Candidate Key for original scheme : ');
    print_at_set(out, key, atr);
    writeln(out);
    for i := 1 to no_schemes do
    BEGIN
        writeln(out); write(out,' ':10, 'R',i,' ( ');
        print_at_set(out, db[i], atr);
        writeln(out,')');
        writeln(out,' ':10, ' Keys are : ');
        write(out, ' ':15);
        if (keys[i] = 0 ) then
           print_at_set(out,db[i],atr)
        else
           for j := 1 to cfd[keys[i]].card do
           BEGIN
               print_at_set(out,cfd[keys[i]].lset[j], atr);
               write(out,' ':3);
           END;
        writeln(out);
        writeln(out,' ':10,'Contained fds :');
        pr_cont_fds(out,cfd,pf[i],atr);
    END;
    writeln(out);
END;
```

```
(*************************************************************************
 *                            Make  Relation                             *
 *************************************************************************)
procedure mkrel(cf: cfd_d; var relation: attr_set);

{ This procedure converts a cfd into a relation. }

VAR
      x   : integer;
BEGIN
    relation := cf.rset;
    for x := 1 to cf.card do
        relation := relation + cf.lset[x];
END;

(*************************************************************************
 *                            Make  Database                             *
 *************************************************************************)
procedure make_db(var rel:right_ar; var pf: equi_set; var keys: key_ar;
                  var no_schemes:integer);

{ This procedure determines the database, also, the keys for subschemes are }
{ determined along with contained fds. }

VAR
    new, x, i   : integer;
    found       : boolean;

BEGIN
    new :=1;
    for x := 1 to ecounter do
    BEGIN
        mkrel(cfd[x], rel[new]);
        pf[new] := [x];    keys[new] := x;
        found := false;    i :=1;
        while ( i < new) and (not found) do
        BEGIN
            if (rel[new] <= rel[i]) then
            BEGIN
                pf[i] := pf[i] + pf[new];
                found := true;
            END
            else
               if (rel[i] <= rel[new]) then
               BEGIN
                   rel[i] := rel[new];
                   pf[i] := pf[i] + pf[new];
                   keys[i] := keys[new];
                   found := true;
               END;
```

```
            i := i+1;
        END;
        if (not found)  then new:= new +1;
    END;
    no_schemes := new -1;
END;

(*************************************************************************
 *                          Put  Attribute                               *
 *************************************************************************)
procedure put_at( var at : attr_ar; new : nametp; var at_num : integer;
                  var R : attr_set );

{  This procedure puts the attribute in the array of attributes if not there }
{  already, while reading.                                                    }

BEGIN
    at_num := 1;
    while ((at[at_num] <> '') and (at[at_num] <> new)) do
        at_num := at_num + 1;
    at[at_num] := new;
    R := R + [at_num];
END;

(*************************************************************************
 *                          Get  F.d.s                                   *
 *************************************************************************)
procedure get_fds(var Infile : Text; var no_fds : Integer; var attr : attr_ar;
                  var fd : functional; var R: attr_set);

{  This procedure reads all the f.ds. from the file for each problem         }
{  separately and stores it in the array of records.                         }

LABEL
     e1;

VAR
   i, attr_num    : Integer;
   temp           : nametp;
   ch, lmarker,
   rmarker,
   relmarker,
   pmarker        : char;
   curr_set       : attr_set;
   marker         : marker_set;

BEGIN
    for i := 1 to mx_attr do
      attr[i] := '';
    R := [];
    Marker  := [',', '$', '-', ' ', '&', '%'];
```

```
    lmarker := '-';
    rmarker := '$';
    relmarker := '&';
    pmarker   := '%';
    curr_set := []; no_fds := 1; temp := '';
    while not eof (infile) do
    BEGIN
        while not eoln (infile) do
        BEGIN
            read(infile, ch);
            if not (ch in Marker) then temp := temp + ch
            else
            BEGIN
                if (temp <> '') then
                BEGIN
                    put_at(attr, temp, attr_num, R);
                    curr_set := curr_set + [attr_num];
                    temp := '';
                END;
                if (ch = lmarker) then
                BEGIN
                    fd[no_fds].left := curr_set;
                    curr_set := [];
                END;
                if (ch = rmarker) then
                BEGIN
                    fd[no_fds].right := curr_set;
                    curr_set := [];
                    no_fds := no_fds + 1;
                END;
                if (ch = relmarker) then curr_set := [];
                if (ch = pmarker)   then goto e1; (* end problem *)
            END;
        END;
        readln(infile);
    END;
e1:
    no_fds := no_fds - 1;    (* Number of fds read *)
END;

(*************************************************************************
 *                        Main  Program                                  *
 *************************************************************************)
BEGIN
    read_file_name(inp, out, inpf_name, outf_name);
    probl_no := 1;
    while (not eof (inp)) do
    BEGIN
        heading(out,inpf_name,probl_no);
        initialize_all(fd, attrb,  equi, e_left, e_right, rel);
        get_fds(inp, no_fd, attrb, fd, R);
```

```
        print_fd_set( out, fd, no_fd, attrb);
        reduce_annular(out,cfd,ecounter, fd, no_fd, equi, e_left, e_right);
        make_db(rel, pf, keys, no_schemes);
        reduced_key( rkey, cfd, R, ecounter);
        if not (lossless(rel, R, no_schemes)) then
        BEGIN
            no_schemes := no_schemes +1;
            rel[no_schemes] := rkey;
            pf[no_schemes] := [];
            keys[no_schemes] := 0;
        END;
        print_db(out, rel, attrb, rkey, R, keys);
        readln(inp);
    END;
    close(out);
    close(inp);
END.
(*****************************************************************************
 *                          End of the Program                               *
 *****************************************************************************)
```

BIBLIOGRAPHY

Aho, A.H., Beeri, C., and Ullman, J.D., "The Theory of Joins in Relational Databases." ACM TODS 4:3, (Sept. 1979) : 297-314.

Armstrong, W.W., "Dependency Structures of Database Relationships", 1974 IFIP Cong., Geneva, Switzerland, (1974) :580-583.

Beeri, C., and Bernstein, P.A., "Computational Problems Related to Design of Normal Form Relational Schemas", ACM TODS 4:1, (March 1979) :30-59.

Beeri, C., "On The Membership Problem for Functional and Multivalued Dependencies in Relational Databases", ACM TODS 5:3, (Sept. 1980) :241-259.

Beeri, C., and Kifer, M., "An Integrated Approach to Logical Design of Relational Database Schemes", ACM TODS 11:2, (June 1986) :135 - 157.

Bernstein, P.A., "Synthesizing Third Normal Form Relations From Functional Dependencies", ACM TODS 1:4, (December 1976) :277-298.

Biskup, J., Dayal, U., and Bernstein, P.A., "Synthesizing Independent Database Schemas", ACM SIGMOD International Symposium on Management of Data, :143-152.

Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", Commun. ACM 13:6, (June 1970) :377-387.

Codd, E.F., "Further Normalization of The Database Relational Model In Database Systems", Courant Computer Science Symposia 6, R. Rustin, Ed., Englewood Cliffs, N.J. : Prentice-Hall, (1971) :65-98.

Delobel, C. and Casey, R.G., "Decompostion of a Database and The Theory of Boolean Switching Functions", IBM J. Res. Dev. 17:5, (Sept. 1972) :374-386.

Fagin, R., "The Decomposition Versus Synthetic Approach to Relational Database Design", Proc. third int. conf. Very large databases, Tokyo, Japan, (Oct. 1977) :441-446.

Fagin, R., "Multivalued Dependencies and a New Normal Form for Relational Databases", ACM TODS 2:3, (Sept. 1977) :262-278.

Fagin, R., Mendelzon, A.O., and Ullman, J.D., "A Simplified Universal Relation Assumption and Its Properties", ACM TODS 7:3, (Sept. 1982) :343-360.

Kent, W., "Consequences of Assuming a Universal Relation", ACM TODS 6:4, (December 1981) :539-556.

Kent, W., "The Universal Relation Revisited", ACM TODS 8:4, (December 1983).

Lien, Y.E., "Hierarchical Schemata for Relational Databases", ACM TODS 6:1, (March 1981) :48-69.

Ling, T.W., Tompa, F.W., and Kameda, T., "An Improved Third Normal Form for Relational Databases", ACM TODS 6:2, (June 1981) :329-346.

Maier, D., The Theory of Relational Databases, Rockville, MD: Computer Science Press, (1983).

Osborn, S.L., and Lucchesi, C.L., "Candidate Keys for Relations", J. of Computer and System Sciences 17, (1978) :270-279.

Ullman, J.D., Principles of Database Systems, Rockville, MD: Computer Science Press, (1980).

Ullman, J.D., "On Kents "Consequences Of A ssuming A Universal Relation"", ACM TODS 8:4, (December 1983).

Yuan, L., and Özsoyoglu, Z.M., "Reduced MVD's And Minimal Covers", ACM TODS 12:3, (Sept. 1987) :377-394.