ISSN 0316-6295

# An Algorithmic Approach to Normalization of Relational Database Schemas+

by

Philip A. Bernstein++
and
Catriel Beeri+++

Computer Systems Research Group
University of Toronto

Technical Report CSRG-73
September 1976

COMPUTER SYSTEMS RESEARCH GROUP
UNIVERSITY OF TORONTO

# An Algorithmic Approach to Normalization of Relational Database Schemas+

by

Philip A. Bernstein++
and
Catriel Beeri+++

Computer Systems Research Group
University of Toronto

Technical Report CSRG-73
September 1976

+ This work has been supported in part by the National Research Council of Canada and by a fellowship from the Lady Davis Fellowship Fund of Jerusalem.

++ Present address: Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, Cambridge, Mass. 02138.

+++Present address: Department of Electrical Engineering and Computer Science, Princeton University, Princeton, N.J. 08540.

The Computer Systems Research Group (CSRG) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their applications. It is jointly administered by the Department of Electrical Engineering and the Department of Computer Science of the University of Toronto, and is supported in part by the National Research Council of Canada.

## ABSTRACT

It has been proposed that the description of a relational data base can be formulated as a set of functional relationships among data base attributes. These functional relationships can then be used to synthesize algorithmically a relational schema. In parts I and II we present an effective and efficient procedure to perform such a synthesis. An abstract view of the procedure is presented in Part I. The schema that results from this procedure is proved to be in Codd's third normal form and to contain the fewest possible number of relations. Problems with earlier attempts at constructing such a procedure are also discussed.

A basic step in the synthesis algorithm is to check for membership of a functional dependency in the closure of a given set of functional dependencies. In part II we present a linear time membership algorithm and we show how it can be used for an efficient implementation of the synthesis algorithm. In part III we treat problems related to Boyce-Codd normal form and to key finding . In particular, the problems of whether a schema is in Boyce-Codd normal form and of whether additional keys exist in a relation are shown to be NP-complete.



Keywords and Phrases: database schema, functional dependency, relational model, semantics of data, third normal form

CR Categories: 3.50, 4.33

## TABLE OF CONTENTS

# PREFACE

In this report we present the results of our efforts towards an algorithmic treatment of several problems related to normalization in relational data base schemas. Our goal was to find efficient algorithms for the solution of these problems. The problems we treat are: Synthesizing a third normal form schema from a set of functional dependencies, deciding the membership problem for functional dependencies, synthesizing a Boyce-Codd normal form schema from a set of functional dependencies and finding all keys of a given relation. The first problem is treated in part I, the second problem is treated in part II, and the last two problems are treated in part III.

The notion of normalization in relational data bases was presented by Codd shortly after he introduced the relational model. Attributes in a relational schema may be related by various relationships. One type of relationship is the functional dependency. It was found that if attributes in a relation are related by functional dependencies in certain ways, then various problems such as update anomalies may arise. Normalization was offered as a solution to this problem. As presented originally it was an ad hoc procedure. The relational schema was assumed to be given. If undesirable functional dependencies existed in a relation of the schema then the relation was decomposed into two or more relations in which the problem did not exist. The process was repeated until no problem functional dependencies existed in any relation; then the schema was said to be in third normal form.

The functional dependency has an important role in the definition and construction of normalized relational schemas. It is also an important concept in the modeling of the semantics of relational data bases. It has, therefore, been suggested that it might be more appropriate to start the construction of a schema with a set of functional dependencies. The relations of the schema would then be synthesized from the given functional dependencies. The success of this approach lies in the existence of a suitable synthesis algorithm. Such an algorithm should produce a schema that represents the information embodied in the given set of functional dependencies and is in third normal form. It should also be efficient enough to be used in real life situations.

There have been several attempts in the literature to describe synthesis algorithms. However, as shown in [2], none of these algorithms was correct. It was observed in [2] that when looking for a synthesis algorithm one must take into account the algebraic properties of functional dependencies. Also, synthesis algorithm based on these properties was present there. However, examples were given of cases where th algorithm produced non-normalized schemas. These examples apply to all the previously published algorithms as well.

In the first part of this report we present a synthesis algorithm which, to our knowledge, is the first to satisfy the goals described above. The synthesized schema represents (in a precisely defined way) the information contents of the original set of functional dependencies, it is provably in third normal form and , furthermore, it contains the minimal number of relations among all possible schemas. At the end of part II we present an efficient implementation of this algorithm. Therefore, we believe that this algorithm is a satisfying solution to the synthesis problem.

A possible reason why all the previous algorithms have failed is that the problem was not specified precisely. In particular, one need to know what is the exact relationship between the original set of functional dependencies and the resulting schema. We define here what it means for a schema to represent the information content of a set of functional dependencies. In Codd's papers on normalization functional dependencies were considered as information external to the schema. For us this posed a problem since we do not know of any relational system in which functional dependencies can be defined. The concept of representation offers a solution to this problem; functional dependencies are now part of the schema and need not be defined separately. We believe that this notion of representation will prove to be useful in the theory of relational schemas.

A basic operation which is used many times in the synthesis algorithm is to check if a given functional dependency can be derived from other functional dependencies. We call this operation the membership test. In the second part of the paper we present an algorithm for the membership test that works in linear time. Using this algorithm, we present a quadratic time implementation of the synthesis algorithm. We doubt if a significant improvement on this time bound is possible. Also, using a known relationship between functional dependencies and the propositional calculus we show that the membership algorithm can be used to decide tautologihood for a restricted class of propositional formulas in linear time. (The last result is not connected to the rest of the report.)

The results in the first two parts indicate that the algorithmic approach to the construction of relational schemas is quite successful. This success seems to be based on the use of the algebra of functional dependencies. Motivated by this success, we tried to analyze two other problems of relational schemas which can be formulated in terms of functional dependencies. These problems are the construction of Boyce-Codd normal form schemas and the computation of all keys of a relation.

The normal form we have been using is the one called in the literature third normal form. A stronger normal form, called Boyce-Codd normal form, was introduced in [8]. One might want to know if our synthesis algorithm would produce schemas in Boyce-Codd normal form. We show that it is not so. There are sets of functional dependencies that cannot be represented by Boyce-Codd normal form schemas. Also, even when a Boyce-Codd normal form schema exists for a set of functional dependencies, the algorithm may produce a non-Boyce-Codd normal form schema. Clearly, the problem of deciding if a given schema is in Boyce-Codd normal form is algorithmically solvable. However, we prove that the problem is NP-complete. These results suggest that a feasible algorithm (that is, an algorithm that runs in polynomial time) for synthesizing Boyce-Codd normal form schemas does not exist.

A key of a relation is a minimal subset of its attributes such that all other attributes of the relation are dependent on it. The synthesis algorithm produces for each relation one or more keys. We present examples where additional (non-synthesized) keys exist in relations. Several algorithms have been described in the literature for the computation of all keys of a relation. However, they are all slow -- they may take exponential time in the worst case. We prove that the problem of whether additional keys exist in a given relation is NP-complete and explain why this result implies that an efficient key finding algorithm probably does not exist.

The report is organized in three parts which are, to some extent, independet. It is possible to understand the results of one part without having to read first the other parts. However, sections I.2 and I.4 contain basic material which is essential to the understanding of all three parts and should, therefore, be read before part II and Part III.

The work reported here was done during the first half of 1976. The first part is an extension and an improvement of the results in chapters 2,5 of [2]. It will be published shortly in the ACM Transactions On Database Systems.

# PART I THE SYNTHESIS PROBLEM

## I.1 INTRODUCTION

Research on the relational data base model has shown that the functional relationship is an important concept when considering how to group attributes into relations [3,7,9,10,15,17]. It has been proposed by some that the basic description of a data base can be formulated purely as a set of such functional relationships from which the relational schema can be synthesized algorithmically [3,17]. It is our purpose in this part of the paper to develop a provably sound and effective procedure for synthesizing relations satisfying Codd's third normal form from a given set of functional relationships. Also, the schema synthesized by our procedure will be shown to contain a minimal number of relations.

This method assumes the existence of at most one functional relationship connecting any one set of attributes to another. This uniqueness assumption, which is required by all earlier methods as well, raises difficult semantic questions that will be discussed in detail.

The first three sections of this part are an introduction to the problem of synthesizing relations from functional dependencies. Section 2 reviews the relational model, the concept of functional dependency, and Codd's normal forms. Two new concepts are introduced: "superkey" and "embodiment". Section 3 outlines the general synthesis problem and presents a simple algorithm for synthesizing relations from functional dependencies. Section 4 introduces Armstrong's axiomatization of functional dependencies and comments on the uniqueness assumption and other semantic considerations of this theory. The main synthesis algorithm is described in section 5. Section 6 examines Codd's third normal form property applied to relations that are synthesized by two versions of the synthesis algorithm. The section concludes with the presentation of a new algorithm for synthesizing provably third normal form relations. In section 7, the synthesized schema is shown to be minimal in size.

## I.2 THE RELATIONAL MODEL

### I.2.1 Relations

In Codd's relational data base model, mathematical relations over a set of domains are used to describe connections among data items [6]. However, not all relations serve equally well in describing these connections [7]. To judge the efficacy of various classes of relations, we begin by reviewing the terminology associated with the relational model.

Conceptually, a <u>relation</u> is a table in which each column corresponds to a distinct <u>attribute</u> and each row to a distinct <u>entity</u> (or <u>tuple</u>). For each attribute there is a set of possible associated values, called the <u>domain</u> of that attribute. It is common for different attributes to share a single domain. For example, the attributes QUANTITY_IN_STOCK and SIZE_OF_CLASS both assume values from the domain called NON-NEGATIVE INTEGERS.

An <entity, attribute> entry in a relation is a value associated with the entity chosen from the domain of the attribute. Formally, a relation is a (finite) subset of the cartesian product of the domains associated with the relation's attributes.

The notation for describing the structure of a data base relation includes a relation name (say R) and a set of attributes in R (say {A1, A2, ..., An}), and is written: R(A1, A2, ..., An), e.g., see fig. 1a. The ordering of attributes is immaterial, since attribute names are distinct within a relation. (This is one reason for distinguishing between attributes and domains.) Notationally, we will use upper case letters near the beginning of the alphabet for simple (i.e., singleton) attributes (e.g., A, B, C) and ones near the end of the alphabet for composite (i.e., groups of) attributes (e.g., X, Y, Z).

The set of entities that comprise a relation normally changes over time, as entities are inserted, deleted, and modified. This is one important way that data base relations differ from mathematical relations.

The word "relation" is often used in the literature to describe both the structure of the relation (e.g., R(A1,...,An)), called its <u>intention</u>, which is static, and the set of tuples in the relation, called its <u>extension</u>. <u>In the sequel, the word "relation" will refer to an intention unless explicitly stated otherwise</u>. That is, we will usually be referring to the structure of a relation, rather than the set of tuples themselves.

### I.2.2 <u>Functional Dependencies</u>

As we will see in later sections, it is important to consider functional relationships when choosing how to group attributes into relations. Functional relationships among data base attributes are formalized in the concept of functional dependency.

Let A and B be attributes, let DOM(A) be the domain of A and DOM(B) be the domain of B, and let f be a time-varying function such that f:DOM(A)->DOM(B). f is not a function in the precise mathematical sense, because we allow the extension of f to vary over time in the same sense that we allow extensions of data base relations to change over time. That is, if f is thought of as a set of ordered pairs {(a,b) | a ∈ DOM(A) and b ∈ DOM(B)}, then at every point in time for a given value of a ∈ DOM(A) there will be at most one value of b ∈ DOM(B). To distinguish f from a mathematical function, we call f a <u>functional dependency</u> (abbr. FD). For notational convenience, we generally leave out the "DOM"s and write f:A->B. If there is an FD f:A->B, then B is said to be <u>functionally dependent</u> on A.

Figure 1

## Relations and Functional Dependencies

(a) An example of a relational schema

EMPLOYEE (EMP#, NAME, DEPT#)

DEPARTMENT (DEPT#, MGR#)

INVENTORY (STOCK#, DEPT#, QTY)

(b) Functional dependencies for the above schema

EMP# -> NAME

EMP# -> DEPT#

DEPT# -> MGR#

STOCK#, DEPT# -> QTY

The above definitions are generalized in the obvious way for functional dependencies over compound attributes. If X = {A1,...,An} and Y = {B1,...,Bm} are sets of attributes, then f:X -> Y means f:DOM(A1)x...xDOM(An) -> DOM(B1)x...xDOM(Bm). We will normally leave off the set notation in FDs and write f:{A1,...,An} -> {B1,...,Bm} simply as f:A1,...,An -> B1,...,Bm. As an example, the functional dependencies for attributes in figure 1a are given in figure 1b.

In this paper we will assume that for any two sets of attributes X and Y, there is at most one FD X -> Y. Attributes may need to be renamed to guarantee this assumption. This restriction is an important one, and will be discussed in detail in section 4.2. We will also show later that nonfunctional relationships need not satisfy this uniqueness assumption.

Given this assumption, if f:A->B, then we will frequently write A->B as an abbreviation. The notation A-/>B means that there is no FD A -> B that is of interest (although at a given point in time in some relation, it may be true that no value of A has more than one corresponding value of B).

Let R(A1,...,An) be a relation and let X be a subset of {A1,...,An}. X is called a <u>key</u> of R if every attribute in {A1,...,An} that is not in X is functionally dependent upon X and if no subset of X has this property. Clearly, a relation can have many keys. A <u>superkey</u> of R is any set of attributes in R that contains a key of R. (Every key is also a superkey.) The concept of superkey is introduced mainly to simplify our proofs in later sections.

### I.2.3 <u>Operations on Relations</u>

In his original description of the relational model, Codd introduced the <u>relational algebra</u> as a data manipulation language for the relational data base model [6]. There are two basic relational algebraic operations that will be of some interest to us: projection and join.

The <u>projection</u> of the extension of a relation, R, on a subset of its attributes, X, is the set of tuples obtained by excising those attributes not in X. If two tuples are now indistinguishable because they only differed in the attributes that were eliminated, then they are "merged" into a single tuple. That is, the result of the projection must be a subset of the cartesian product of the domains associated with the attributes of X.

The join operation is used to make a connection between attributes that appear in different relations. The only join operation we will consider here is the <u>natural join</u> (i.e., equality join). The natural join of the extension of a relation R(A,B) with the extension of relation S(B,C) on domain B, denoted R*S, is defined to be {(a,b,c) | (a,b) ∈ R and (b,c) ∈ S}. That is, it links together all values of A and C that are related to common B values.

### I.2.4 Schemas

The purpose of any data model, relational or otherwise, is to allow the user of the model to describe and manipulate those relationships among objects in the real world that he intends to be stored in the data base. In the relational model, such a collection of relationships is represented in a relational schema. A relational schema consists of a set of data base relations and for each relation the specification of one or more keys (e.g., see fig. 1a).

We will say that a functional dependency X -> A is embodied in a relation R if X is a key of R and A is any other attribute of R. The set of FDs embodied in a schema is the union of the sets of FDs embodied in all of the relations of the schema.

Notice that this formulation of schema is a modification of Codd's [9], where FDs are given as information additional to the relations and their keys. We have adopted this modified notion of schema for several reasons. First, all data definition languages that we know of only allow the specification of relations and keys. Second, our definition of schema eliminates the need to talk explicitly of FDs. The FDs exist implicitly by virtue of our definition of keys and embodiment. Third, we will see that third normal form organizes a relational schema so that every FD that is given as external information is either embodied in some relation or can be recovered from embodied FDs by the join and projection operations. Taking this notion of embodiment as a primitive concept simplifies the ensuing theory considerably.

### I.2.5 Normalization

Codd observed that certain relations have structural properties that are undesirable for describing data bases. This led him to define a series of three normal forms for relations.

First, relation-valued domains are excluded from relations. A relation is in first normal form (abbr. 1NF) if each domain contains simple values. There are two main advantages to 1NF [6]. First, it allows the data base to be viewed as a collection of tables -- a very simple and understandable structure. Second, it permits the definition of a small class of primitive operators that are capable of manipulating relations to obtain all necessary logical connections among attributes.

The second and third normal forms are introduced to correct problems caused by certain functional dependencies. To examine these problems, consider the relation DEPT_INV(STOCK#,DEPT#, QTY, MGR#) obtained by joining the DEPARTMENT and INVENTORY relations of fig. 1a on the attribute DEPT#. The insertion of the first inventory item for a particular DEPT# into the extension of DEPT_INV creates a new connection betweeen that DEPT# and its MGR#. The deletion of the last inventory item for a particular DEPT# loses the connection between that DEPT# and its MGR#. These side effects, called insertion-deletion anomalies, only occur when the first or last tuple of a DEPT# is inserted or deleted. Also, the repetition of the connection between a DEPT# and its MGR# for each STOCK# in the DEPT# can lead to an inconsistent relation if arbitrary updates on individual tuples are permitted. These problems arise because MGR# is functionally dependenct on only part of the key STOCK#, DEPT#. To eliminate these problems from DEPT_INV, DEPT_INV must be put into second normal form.

A partial dependency occurs when an attribute is functionally dependent upon a subset of a set of attributes. Let f:A1,...,An->B and g:A1,...,Am->B be functional dependencies where m<n. The attributes Am+1, Am+2, ..., An are extraneous in f, since A1, ..., Am are sufficient to functionally determine B. In this case, B is said to be partially dependent on A1,...,An. If for a given f there is no g with the above property, then B is fully dependent on A1,...,An. That is, there are no extraneous attributes in the domain of f.

If an attribute Ai appears in any key of R then it is said to be prime in R. Otherwise, it is nonprime in R. A relation is in second normal form (abbr. 2NF) if it is in 1NF and each of its nonprime attributes is fully dependent upon every key [7]. The relation DEPT_INV(STOCK#,DEPT#, QTY, MGR#) is not in 2NF, because MGR# is a nonprime attribute and is partially dependent on the key STOCK#, DEPT#. The relations DEPT and INVENTORY in fig. 1a are in 2NF.

Consider now the relation EMP_DEPT(EMP#, NAME, DEPT#, MGR#) obtained by joining the EMPLOYEE and DEPARTMENT relations of fig. 1a on DEPT#. Although EMP_DEPT is in 2NF, it displays the same problems as DEPT_INV. Inserting or deleting the first EMP# in a particular DEPT# creates an anomaly, for a DEPT#-MGR# connection is created or destroyed in the process. The repetition of the DEPT#-MGR# connection for each EMP# in the DEPT# creates the same consistency problem as in DEPT_INV. In this case, the problems arise because MGR# is functionally dependent on the key EMP# via the attribute DEPT#. To eliminate the problems, the relations EMP_DEPT must be put into third normal form.

Let R(A1,...,An) be a relation. An attribute, Ai, is transitively dependent upon a set of attributes, X, if there exists a set of attributes, Y $\subseteq$ {A1,...,An}, such that X -> Y, Y -/> X, and Y -> Ai with Ai not an element of X or Y.

A relation is in third normal form (abbr. 3NF) if none of its nonprime attributes are transitively dependent upon any key [7]. A 3NF relation is also in 2NF; for if an attribute Ai is partially dependent on a key X, then Ai is transitively dependent on X, since X -> X', X' -/> X, and X' -> Ai for some X' c X. The relation EMP_DEPT is not in 3NF, because MGR# is nonprime and is transitively dependent upon the key EMP#. All of the relations in figure 1a are in 3NF (and hence 2NF), given the

FDs of figure 1b. Further examples of normal form relations and surrounding problems can be found in [6,7,9].

## I.3 SYNTHESIZING A RELATIONAL SCHEMA

### I.3.1 The Synthesis Problem and Nonfunctional Relationships

Codd showed that by applying simple decomposition steps to a 1NF relation in which the FDs were known, the relation could be split up into a set of relations in 3NF that embodies all of the FDs [7]. In [3], it was proposed that since the FDs completely determine whether or not a relation is in 3NF, one could choose the FDs as the basic concept and build 3NF relations from them. In advancing this proposal, an efficient algorithmic technique was presented to actually construct relations from FDs. In this part of the paper we present an improved algorithm and then discuss properties of schemas synthesized by this algorithm.

The approach of building a relational schema from FDs rests entirely on the ability to represent all data relationships as FDs. Clearly, though, not every logical connection in the world is functional. Nevertheless, we claim that all connections among attributes in a data base description can be represented by FDs. As long as connections are functional there is of course no problem. Nonfunctional connections require special treatment.

A nonfunctional connection, f, among a group of attributes A1,A2,...,An will be represented as the following FD: f:A1,A2,...,An -> $\theta$. $\theta$ is an attribute that is unique to f; it does not appear in any other FD. Each FD representing a nonfunctional relationship has its own private $\theta$ attribute. The underlying domain for all of these $\theta$ attributes is the set {0,1}. For each element (a1,a2,...,an) $\in$ DOM(A1)xDOM(A2)x...xDOM(An), f(a1,a2,...,an) = 1 if and only if (a1,a2,...,an) is related under f. Thus, the extension of f completely defines a nonfunctional relationship among A1,...,An. For example, a nonfunctional relationship between a DRIVER and AUTOMOBILE, where each AUTOMOBILE can be driven by more than one DRIVER and each DRIVER can drive more than one AUTOMOBILE, is represented by the FD: DRIVER,AUTOMOBILE -> $\theta$1.

Notice that more than one nonfunctional relationship can exist among a set of attributes without violating the uniqueness assumption of FDs. For example, we can have a second relationship between DRIVER and AUTOMOBILE that indicates ownership: DRIVER,AUTOMOBILE -> $\theta$2. By assigning a unique $\theta$ to each nonfunctional relationship, the uniqueness assumption for FDs is retained.

This $\theta$ notation allows us to represent all nonfunctional relationships as FDs. The synthesis algorithm will produce approximately one relation for each of these nonfunctional relationships. In section 5, we will show precisely how each of these "nonfunctional FDs" becomes embodied in the synthesized relational schema.

### I.3.2 Formalizing the Synthesis Problem

Although the motivation for the synthesis problem is from data base management, one can formalize the problem in purely symbolic terms as follows. We are given a set, S, of symbols (i.e., attributes) and a set, F, of mappings of sets of symbols into symbols (i.e., FDs). The problem is to find a collection C = {C1,...,Cm} of subsets of S (i.e., a collection of relations) and for each Ci a collection of subsets of Ci (i.e., a collection of keys for each relation) satisfying three properties: First, F is "embodied" in C (i.e., the relations embody the given FDs). Second, each Ci can have no transitive dependencies (i.e., it is in 3NF). Third, the cardinality of C is minimal.

This treatment of the problem is still somewhat fuzzy, since we have not yet discussed the algebraic rules for composing FDs. To motivate the need for these rules, we present a simple synthesis algorithm. This algorithm ignores algebraic considerations and will be shown to be inadequate.

### I.3.3 A Simple Synthesis Procedure

One (overly) simple way to obtain relations from a given set of FDs is to group together all attributes that are functionally dependent upon the same set of attributes. This suggests the following procedure. First, partition the given set of FDs into groups such that all of the FDs in each group have identical left sides. Then, for each group construct a relation consisting of all the attributes appearing in that group. The left side of the FD in each group is a key of the corresponding relation. For example, see figure 2.

Several undesirable properties of this method can be seen in the example. First, the synthesized relations are not in 3NF. For example, in relation R1 of fig. 2, C is transitively dependent on the key A. In R4, B is partially dependent on the key AE. The unnormalized relations are due to redundancies in the given set of FDs. We will see later that f2 is redundant and that F is an extraneous attribute in f6.

Second, the left sides of FDs are not necessarily keys of the relations, although they are always superkeys. In R4, ABF is a superkey but not a key, since B is extraneous.

Third, this procedure synthesizes too many relations. Since f4 and f5 are inverses of each other, the relation R3 is extraneous. This results from a failure of the procedure when constructing R2 to recognize D as a second key by virtue of f5, rather than to put f5 into a separate relation.

To solve these problems, we must first formalize the concept of a redundant FD. We will then return to a presentation of a synthesis algorithm that overcomes the above difficulties.

Figure 2

## Deriving a Schema from FDs

We are given the following set of FDs:

f1: A -> B

f2: A -> C

f3: B -> C

f4: B -> D

f5: D -> B

f6: ABE -> F

We group the FDs according to common left hand sides, obtaining three groups:

g1 = {f1,f2}

g2 = {f3,f4}

g3 = {f5}

g4 = {f6}

For each group we construct a relation consisting of all of the attributes in the group:

R1($\underline{A}$,B,C)

R2($\underline{B}$,C,D)

R3($\underline{D}$,B)

R4($\underline{A}$,$\underline{E}$,B,F)

where the underscored attributes are keys.

## I.4 THE ALGEBRA AND SEMANTICS OF FUNCTIONAL DEPENDENCIES

### I.4.1 Armstrong's Axiomatization of Functional Dependencies

The complete axiomatization of FDs given by Armstrong [1] provides a theoretical background to the study of the algebra of FDs that is treated in later sections. Armstrong shows that if a given set of FDs exist in (the extension of) a relation, then any FDs that can be derived from the given set using the axioms must also exist. Armstrong presents several equivalent axiomatizations of FDs. The one we will use is based on properties of FDs proved by Delobel and Casey [10]. They are:

A1. (reflexivity) X -> X

A2. (augmentation) if X -> Z then X+Y -> Z

A3. (pseudotransitivity) if X -> Y and Y+Z -> W then X+Z -> W

where the symbol "+" means "set union" (of not necessarily disjoint sets).

If R(A,B) is a relation, then axiom A1 can be applied with X={A,B} to show that A,B -> A,B or with X={A} to show that A -> A.

The meaning of A2 is simply that if f:X -> Z, then one can create another FD, g, where the domain of g includes X as well as some other extraneous attributes, Y, whose values have no effect on the value of Z selected by g. So, knowing that A -> A, we can obtain A,B -> A (i.e., X={A}, Z={A}, and Y={B}).

Axiom A3 is a substitution rule for composing FDs. Let f:X -> Y and g:Y+Z -> W. The axiom claims that there is an h:X+Z -> W. To see where h comes from, consider the application of h to a given x ∈ DOM(X) and z ∈ DOM(Z) in two steps. First, f is applied to x, yielding a unique y ∈ DOM(Y). Second, g is applied to y and z, yielding a unique w ∈ DOM(W), and thereby completing the application of h. Symbolically, we can say h(x,z) is defined to be g(f(x),z). Also, note that in the statement of axiom A3 if Z is the null set, then pseudotransitivity becomes simple transitivity.

Let G be a set of FDs. The <u>closure</u> of G, denoted $G^+$, is defined to be the smallest superset of G that is closed under A1, A2, and A3. For a given G, $G^+$ can be shown to be unique. By Armstrong's theory we know that if G is a given set of FDs for a relation R, then each FD in $G^+$ also exists in R.

An FD g ∈ G is <u>redundant in G</u> if $G^+ = (G-\{g\})^+$. H is a <u>nonredundant covering</u> of a given set of FDs, G, if $G^+=H^+$ and H contains no redundant FDs.

An important property of FDs that will be used later to prove a number of theorems is stated in lemma 1. It is based on the concept of a "derivation", which we will informally consider to be a series of applications of Armstrong's axioms on a given set of FDs. A formal development appears in part II.

Lemma 1: Let G be a set of FDs, and let g:X -> Y be an FD in G. If h:V -> W is in G+ and g is used for some derivation of h from G, then V -> X is in G+.

Proof We give here an intuitive argument using an informal notion of a derivation. A formal proof using the "derivation tree" model of derivations is given in section II.1.2

We introduce the notation U ==> Z to mean that the FD U -> Z can be derived by an application of one of Armstrong's axioms on a given set of FDs. The notation U =*=> Z means that U -> Z is derivable using several applications of the axioms. Now, the lemma states there is a derivation V =*=> W using g. That is, there is a derivation V =*=> ZX ==> ZY =*=> W for some (possible empty) set of attributes Z (the step ZX ==> ZY is the step that uses g). But V =*=> ZX implies V -> ZX, which implies V -> X, thereby proving the lemma. □

### I.4.2 Uniqueness and the Semantics of FDs

The treatment of FDs in this paper is a strictly syntactic one based on Armstrong's axioms. To use this approach, we must make the following assumption of uniqueness: for a given set of FDs G and an FD X -> Y, either X -> Y is not in G+ or there exists a unique FD X -> Y in G+. That is, if there are two FDs on the same set of attributes, then they are the same FD; if f:X -> Y and g:X -> Y then f is identical to g. Thus, the set of FDs that are accepted as input to the synthesis algorithm is assumed to satisfy not only Armstrong's axioms, but also the uniqueness assumption. (Both of these assumptions are also required for all previous syntactic approaches to 3NF (e.g., [10,15,17]).) That this uniqueness assumption is quite strong can be seen from several examples.

Let f1:DEPT# -> MGR# and f2:MGR#,FLOOR -> NUMBER_OF_EMPLOYEES. One interpretation of f1 and f2 is that f1 determines the manager of each department and f2 determines the number of employees working for a particular manager on a particular floor. By applying pseudotransitivity to f1 and f2 we obtain f3:DEPT#,FLOOR -> NUMBER_OF_EMPLOYEES, which determines the number of employees of the manager of a particular department on a particular floor. If a manager can manage more than one department, then f3 is not the same as the syntactically identical FD g1:DEPT#,FLOOR -> NUMBER_OF_EMPLOYEES, which determines the number of employees of a particular department on a particular floor. To make g1 distinct from f3, one has to change an attribute name to make the FDs syntactically distinct. For example, one could change f2 and g such that f2:MGR#,FLOOR -> NUMBER_OF_EMPLOYEES_OF_MANAGER and g1:DEPT#,FLOOR ->

NUMBER_OF_EMPLOYEES_OF_DEPT. Now, g1 is distinct from the composition of f1 and f2.

As a second example, let f4:EMP# -> MGR# and f5:MGR# -> EMP#. It must be the case, here, that f4 is the inverse of f5. For if we compose f4 and f5, we obtain g2:EMP# -> EMP#. Since there is only one FD connecting EMP# to EMP# (by our assumption), and since by Armstrong's axioms the identity function must exist, then g2 must be the identity map. This implies $f4 = f5^{-1}$. If we take the interpretation that f4 maps an employee into his manager and f5 maps a manager's MGR# into his corresponding EMP#, then of course $f4 \neq f5^{-1}$. So to take this interpretation, one must make f4 and f5 syntactically distinct (e.g., f5:MGR# -> EMP#_OF_MGR).

As a third example, let f6:STOCK# -> STORE# and f7:STOCK#,STORE# -> QTY. Since the composition of f6 and f7 is g3:STOCK# -> QTY, it must be (by our assumption) that the attribute STORE# in f7 is not needed. But suppose f6 maps a STOCK# into the STORE# of the store that is in charge of ordering that item and f7 maps the STOCK# of an item and the STORE# of the store in which it is being sold into the quantity on hand. In this case, g3 does not imply that STORE# is extraneous in f7. To prevent this syntactic inference from taking place, we must change an attribute name (e.g., f6:STOCK# -> ORDERING_STORE#).

In each of these examples, a syntactic inference was either erroneous or misleading. In each case, we solved the problem by renaming an attribute to distinguish it from another attribute. This renaming essentially moves some semantic knowledge that we have about an FD onto the syntactic level, where it can be used by the algebra of FDs.

Specifying a set of FDs that can lead to no invalid syntactic inferences is clearly a difficult problem. For no syntactic check based only on the algebra of FDs can determine whether a given set of FDs satisfies the uniqueness assumption. Yet, if we are to make use of a formal algebra of FDs, we must make the assumption that all syntactic inferences are valid. If we had an automated semantic analyzer that could judge the validity of each syntactic inference, then we could use it as a sieve to toss out invalid inferences. Unfortunately, such a semantic analyzer is well beyond the state of the art. So, we will add to our assumption of the validity of syntactic inferences the proviso that all syntactic inferences are (or at least can be) checked for semantic validity. If an inference is invalid, it can either result in renaming of some attributes or be simply rejected.

Third normal form is a strictly syntactic property that is governed by the algebra of FDs. In this work we give a complete account of mapping from FDs into a 3NF schema, given that Armstrong's axioms and the uniqueness assumption are accepted. Given Armstrong's completeness proof, we believe these assumptions to be quite reasonable in modelling relational data bases. We are not attacking the problem of how to judge the semantic validity of syntactic inferences. Semantic problems of this type are not well understood and seem to be more difficult than the syntactic problem of determining 3NF. Their solution remains a matter for further research.

## I.5 A MORE SOPHISTICATED SYNTHESIS PROCEDURE

### I.5.1 A Description of the Algorithm

The simple synthesis procedure of section 3.3 led to problems because the rules for composing FDs were ignored. The main difficulty is that redundant FDs that filter into the synthesized schema create extra attributes and contribute to unnormalized connections among attributes. By first taking a nonredundant covering of the given set of FDs, the normalization problems can be alleviated. In fig. 2, for example, f2 is redundant and therefore will not appear in a nonredundant covering of the given FDs, thereby avoiding the 3NF violation of R1.

Finding a nonredundant covering is not sufficient to avoid problem FDs such as f6 in fig. 2. This further problem can be eliminated by excising extraneous attributes from the left sides of FDs. An attribute Ai is _extraneous in an FD_ g ∈ G, g:A1,...,Ap -> B, if A1,...,Ai-1,Ai+1,...,Ap -> B is in $G^+$. Eliminating extraneous attributes helps to avoid partial dependencies and superkeys that are not keys, such as in R4 of fig. 2.

If two relations have keys that are functionally dependent upon each other (i.e., are equivalent), then the two relations can be merged together. This can be accomplished in the synthesis procedure by merging together two groups of FDs if their left sides are functionally equivalent. For example, g2 and g3 in fig. 2 can be merged into a single group.

Algorithm 1 (see figure 3) includes the above improvements. In the sequel, we will refer to Algorithm 1 with step 4 excised as Algorithm 1a.

A linear time algorithm for testing membership in the closure of a set of FDs is presented in part II. It is also shown there that using this procedure, one can implement Algorithm 1 with a time bound of $O(L^2)$, where L is the length of the string encoding the given set of FDs.

### I.5.2 Completeness of the Synthesized Schema

A schema S _represents_ a set of FDs F if the closure of the FDs embodied in the relations of S equals $F^+$. To show that Algorithm 1 synthesizes a schema that represents the given FDs, consider a set of FDs F that is given as input to Algorithm 1. Let H be the set of FDs that result from eliminating extraneous attributes and redundant FDs. Clearly, $H^+$ still equals $F^+$. Let S be a schema synthesized from F. Since H is exactly the set of FDs embodied in S and $H^+ = F^+$, S represents F.

Figure 3

## Algorithm 1

## Synthesizing a Relational Schema from a Set of FDs

1. (Eliminate extraneous attributes) Let F be the given set of FDs. Eliminate extraneous attributes from the left side of each FD in F, producing the set G. An attribute is extraneous if its elimination does not alter the closure of the set of FDs.

2. (Find covering) Find a nonredundant covering, H, of G.

3. (Partition) Partition H into groups such that all of the FDs in each group have identical left sides.

4. (Merge equivalent keys) For each pair of groups, say H1 and H2, with left sides X and Y respectively, merge H1 and H2 together if there is a bijection X <-> Y in $H^+$.

5. (Construct relations) For each group, construct a relation consisting of all the attributes appearing in that group. Each set of attributes that appears on the left side of any FD in the group is a key of the relation. (Step 1 guarantees that no such set contains any extra attributes.) All keys found by this algorithm will be called <u>synthesized</u>. The set of constructed relations constitutes a schema for the given set of FDs.

We would like to be certain that the extension of every FD in the given set F can be retrieved from the extension of the synthesized schema S using relational algebra. We will argue that this follows from the fact that S represents F.

Consider some $f:X \rightarrow A \in F$. We begin by noting that extraneous attributes in X can be ignored. That is, if the extension of an FD $f':X' \rightarrow A$ where $X' \subseteq X$ can be retrieved from the extension of S, then since f can be obtained from f' simply by augmentation, we can treat f' to be the same FD as f. Now, since S represents F, there is a derivation for f' based on the set of FDs, H, that are embodied in S. In part II, we show that if f' has no extraneous attributes, then it can be derived from H using only the pseudotransivity axiom. Since an application of psuedotransitivity corresponds exactly to a join in relational algebra, the derivation for f' from H can be simulated by a sequence of joins on the extension of the relations of S. In this way, the extension of every $f \in F$ can be retrieved from the extension of S using relational algebra. That is, our notion of 'representation' satisfies the intuition that all relationships specified in the given set of FDs are actually retrievable from the extension of the synthesized schema.

I.5.3 Nonfunctional Relationships

We introduced a special notation for representing nonfunctional relationships in our input FDs. We must now make sure that these FDs behave in the expected way.

If $X \rightarrow \theta$ is in the set of FDs given to Algorithm 1, then either $X \rightarrow \theta$ or $Y \rightarrow \theta$, where $Y \leftrightarrow X$, appears in the schema synthesized by the algorithm. This is a consequence of the following lemma, which is proved in part II. Thus, the nonfunctional relationships appear in the schema in nearly the same form that they are specified in the given set of FDs.

Lemma 2: If $X \rightarrow \theta$ is in a set of FDs G, then for any nonredundant covering H of G, either $X \rightarrow \theta$ is in H or $Y \rightarrow \theta$ is in H, where $X \rightarrow Y$ and $Y \rightarrow X$. □

By the above lemma, the $\theta$ attributes, which were invented to permit the representation of nonfunctional relationships, always appear in the synthesized schema. How are they interpreted? To see this, consider the following example. Suppose two nonfunctional relationships were specified in the given set of FDs: $f1:AB \rightarrow \theta1$ and $f2:AB \rightarrow \theta2$. (Notice again that the uniqueness assumption of FDs does not force uniqueness of nonfunctional relationships between A and B.) Step 4 of Algorithm 1 merges these two FDs into a single group, yielding a relation $R(\underline{A},\underline{B},\theta1,\theta2)$. In order to distinguish whether a given pair of values for A and B satisfy f1, f2, or both f1 and f2, the $\theta1$ and $\theta2$ attributes must be retained. For example, $\langle a, b, 0, 1\rangle \in R$ means a,b satisfies f2 but not f1. Notice that if there is only one nonfunctional relationship among a set of attributes, then the $\theta$ attribute can generally be dropped, since this problem of distinguishing among relationships disappears. For example, if only f1 were present, then it is customary only to include <a,b> pairs that are related under f1; a tuple <a,b,0> would normally not be included in the extension. Therefore, in this case, the $\theta$ attribute would be dropped altogether.

## I.6 THIRD NORMAL FORM SCHEMAS

### I.6.1 Introduction

In this section we show under what conditions various synthesized relations are in 3NF. We begin by showing that Algorithm 1a (i.e., Algorithm 1 without step 4) always produces a 3NF schema. We then examine Algorithm 1. A property of derivations of nonprime attributes is introduced and shown to be a sufficient condition for Algorithm 1 to produce a schema in 3NF. Unfortunately, there are cases of FDs that do not satisfy this property and therefore can lead to relations with transitive dependencies. One such example is presented and is shown to be a counterexample to a theorem given by Delobel and Casey.

### I.6.2 Algorithm 1a Schemas

To prove that every relation synthesized by Algorithm 1a is in 3NF, we show that a transitive dependency implies the existence of a redundant FD in the nonredundant covering. We will use lemma 1 (cf. section 4.1) to show the existence of an FD that creates the contradiction. Lemma 1 will be used in this way in all succeeding 3NF proofs.

Theorem 1: Let R(A1,...,An) be a relation synthesized from the set of FDs F using Algorithm 1a. Then no nonprime attribute of F is transitively dependent upon any key of R. That is, R is in 3NF.

Proof Suppose Ai is nonprime and is transitively dependent upon a key, K, of R. (K need not be synthesized.) That is, there is an X $\subseteq$ {A1,...,An} such that K -> X, X $\not\rightarrow$ K, and X -> Ai are in $F^+$, and Ai is not in X.

We first observe that Ai is transitively dependent upon the synthesized key of R. Let Z be the key of R that appears on the left side of the FDs that were used in synthesizing R. Clearly, Z -> X is in $F^+$. Furthermore, X $\not\rightarrow$ Z. For if X -> Z, then X -> Z and Z -> K would imply X -> K, contradicting X $\not\rightarrow$ K in the original transitive dependency. Hence, Z -> X, X $\not\rightarrow$ Z, and X -> Ai is also a transitive dependency.

Let H be the nonredundant covering of G computed in Algorithm 1a. We will now show that Z -> Ai, which appears in H, is redundant. To do this, it is sufficient to show that Z -> X and X -> Ai can both be derived from H - {Z->Ai}.

Since the only FDs used in synthesizing R are of the form Z -> Aj, it must be that Z -> A is in H for all A $\in$ X. Since Ai is not in X, Z -> A is in H - {Z -> Ai} for all A $\in$ X.

Suppose there is a derivation for X -> Ai in H that uses Z -> Ai. Then, by lemma 1 we have X -> Z. But this violates X -/> Z in the transitive dependency. So X -> Ai must be derivable without using Z -> Ai.

Since Z -> X and X -> Ai can both be derived from H - {Z -> Ai}, it must be that Z -> Ai is redundant in H, contradicting the fact that H is nonredundant. But this, in turn, must mean that the transitive dependency did not exist. □

The above theorem was first presented by Wang and Wedekind [17], however their proof was incorrect [4]. In the proof, they only argued that the transitive dependency was derivable in H, not H-{Z->Ai}. In terms of the above proof, they claimed that if K -> X and X -> Ai is a transitive dependency, then K -> Ai is derivable by pseudotransitivity. This, they asserted, violates the fact that K -> Ai is in a nonredundant covering. However, the latter is only true if one can show that both K -> X and X -> Ai are derivable from the closure without using K -> Ai. For example, G = {A -> B, B -> A, A -> C} is a set of FDs where A -> B and B -> C are in the closure but A -> C is not redundant, because B -> C cannot be derived from G without A -> C. In any case, their theorem was correct as stated, and the above argument fixes their proof, using lemma 1 and the important fact that X -/> K in the transitive dependency.

It is interesting to note that they did not eliminate superkeys in their version of Algorithm 1a. This was not an error, since they explicitly <u>assumed</u> that extraneous attributes did not exist on the left sides of FDs. However, one need not make this general assumption, since some extraneous attributes can be eliminated algorithmically. In fact, to be entirely consistent with the algebra of FDs, one <u>must</u> eliminate such extraneous attributes. Of course, not all such extraneous attributes can be eliminated in this way; many semantic errors must remain the user's responsibility for reasons discussed in section 4.2.

One might expect the proof of Theorem 1 to generalize to schemas synthesized by Algorithm 1. Unfortunately, this is not the case. A schema that is not in 3NF can be synthesized by Algorithm 1, as shown in figure 4(ii). In the next section we will add a further precondition that is sufficient to guarantee 3NF for schemas produced by Algorithm 1.

### I.6.3 <u>A Sufficient Condition for 3NF</u>

To gaurantee that no nonprime attribute is transitively dependent upon any key of R, we will use the following property:

> An attribute A is said to satisfy <u>property P</u> in relation R if the following proposition holds: Let H be the nonredundant covering produced from step 2 of Algorithm 1. If K -> A is in H and K -> A is used in synthesizing R, then for any prime

attribute B of R, the FD K -> B can be derived without using K -> A (i.e., can be derived in H - {K -> A}).

Property P is strictly a syntactic property of derivations of FDs, and to our knowledge has no semantic interpretation in terms of real world relationships. It is the weakest property we know of that is sufficient to guarantee that Algorithm 1 produces 3NF relations. The proof that property P is sufficient to guarantee 3NF follows the same lines as the proof of theorem 1.

<u>Theorem</u> <u>2</u>: Let R(A1,...,An) be one of the relations synthesized using Algorithm 1 from a set of FDs, F. If all nonprime attributes of R satisfy property P, then R is in 3NF.

<u>Proof</u> Let Ai be a nonprime attribute of R that is transitively dependent upon some key Y of R. That is, there is a Z c {A1,...,An} such that Y -> Z, Z -/> Y, and Z -> Ai with Ai not in Z. Let H be the nonredundant covering computed in Alg. 1 and let K be a key such that h:K -> Ai is in H. That is, h is an FD that brought Ai into R by Alg. 1.

Since K is a key, K -> Z. Furthermore, Z -/> K. For if Z -> K, then Z -> K and K -> Y implies Z -> Y, a contradiction. So, we have a new transitive dependency: K -> Z, Z -/> K, and Z -> Ai. We want to show that K -> Z and Z -> Ai are in (H - {K -> Ai})+ to establish a contradiction that H is redundant.

Let Z = {B1,...,Bm}. We distinguish two cases. If Bj is prime, then property P guarantees that K -> Bj is derivable without using K -> Ai. If Bj is not prime, then there is an FD K' -> Bj that brought Bj into R. Since K -> K' is derivable (by property P) from H - {K -> Ai} and K' -> Bj is in H -{K -> Ai}, we obtain that K -> Bj is derivable without using K -> Ai. Hence, K -> Z is in (H-{K->Ai})+.

Now, assume Z -> Ai uses K -> Ai in its derivation. Then by lemma 1, Z -> K, contradicting the transitive dependency. Hence, Z -> Ai is in (H - {K -> Ai})+.

The FDs K -> Z and Z -> Ai are in (H - {K -> Ai})+, establishing that H is redundant, a contradiction. Hence the transitive dependency could not have existed. □

The need for property P arises from the merging of equivalent keys in step 4 of Algorithm 1. Suppose K1 and K2 are merged in step 4 because K1 <-> K2, and suppose K1 -> Z, Z -/> K1, Z -> A is a transitive dependency in the synthesized relation. This transitive dependency would not exist if K1 and K2 were the keys of two separate relations, as would be the case using Algorithm 1a. One way the transitive dependency can arise is that there is a Zi ∈ Z, with K2 -> Zi (and K1 -> A) in the nonredundant covering, but K1 -> Zi (and K2 -> A) not in the

Figure 4

**Examples of Violations of Property P**

| FDs | Schema Synthesized by Algorithm 1 |
|---|---|
| f1: X1,X2 -> A | R1(X1,X2,C,A) (from f1 & f2) |
| f2: C -> X1,X2 | |
| f3: A,X1 -> B | R2(A,X1,B) |
| f4: B,X2 -> C | R3(B,X2,C) |

A does not satisfy property P, yet R1 is in 3NF.

(i)

| | |
|---|---|
| g1: X1,X2 -> A,D | S1(X1,X2,C,D,A) (from g1 & g2) |
| g2: C,D -> X1 X2 | |
| g3: A,X1 -> B | S2(A,X1,B) |
| g4: B,X2 -> C | S3(B,X2,C) |
| g5: C -> A | S4(C,A) |

S1 is not in 2NF, since A is partially dependent upon the key CD.

(ii)

covering. Thus, a relation must contain both K1 and K2 to manifest the transitive dependency. The FD K1 -> Z in the transitive dependency is a compostion of K1 -> K2 -> Z. If A does not have property P, then K1 -> A may be necessary to obtain K1 -> K2, in which case K1 -> A need not be redundant (as it would be in Algorithm 1a). However, if A does have property P, then K1 -> K2 does not need K1 -> A, so K1 -> A _is_ redundant, and we have the theorem.

Consider the set of FDs in figure 4(i) which produces the relation R1(_X1_,_X2_, _C_, A) via Algorithm 1. (The reader can check that X1,X2 -> C is in the closure of the given FDs.) The attribute A is nonprime in R1 and does not satisfy property P, since the only way to derive X1,X2 -> C is using X1,X2 -> A. However, despite the violation of property P, relation R1 is in 3NF. Hence, property P is not a necessary condition for 3NF.

Figure 4(ii) presents an example of FDs that exhibit the same violation of property P as figure 4(i) but induce a partial (and, hence, a transitive) dependency. In terms of the above discussion regarding transitive dependencies, we have X1,X2 -> C, C -/> X1,X2, and C -> A; but X1,X2 -> A is not redundant, since X1,X2 -> C needs X1,X2 -> A in its derivation.

Property P affects one other published procedure that synthesizes relations from FDs. Delobel and Casey [10] claim that their decomposition procedure, which is in some sense comparable to our Algorithm 1, produces 3NF relations. Their claim, however, is incorrect in that the example in figure 4(ii) falsifies their theorem.

### I.6.4 _Putting Relations into 3NF_

A violation of property P may induce a 3NF violation. Once a particular violation of 3NF is found, then to put the relation into 3NF the offending dependency must be removed. Conveniently enough, if a nonprime attribute is transitively dependent upon a key of a relation, then the attribute can simply be removed from the relation, and the resulting schema will still embody the same FDs.

_Theorem 3_: Let Rk(A1,...,An) be a relation in a schema S={R1,...,Rm} that was synthesized using Algorithm 1. Let H be the set of FDs embodied in S. Let Ai be an attribute of Rk that appears in none of Rk's _synthesized_ keys, and let Ai be transitively dependent upon a key of Rk. Suppose Ai is removed from Rk, resulting in a new relation Rk' and, hence, a new schema S' = {R1,...,Rk',...,Rm}. Then the closure of the set of FDs embodied in S' equals H+.

_Proof_ Suppose Ai is removed from Rk. Since Ai does not appear in any of the keys synthesized by Alg. 1, its removal can only affect embodied FDs of the form f:X->Ai, where X is a synthesized key of Rk. Let H' be the set of FDs embodied in S'. If we show that all such f are in (H')+, then H+ = (H')+.

By the same argument used in the proofs of theorems 1 and 2, if Ai is transitively dependent upon any key of Rk, then it is transitively dependent upon all of them. For each f of the above form, X is a key. Therefore, for each such X, X -> V, V -/> X and V -> Ai, Ai not in V, is a transitive dependency.

Since Ai is not in V, for each X the FD X -> V is still embodied in Rk even after Ai is removed. Hence, each X -> V is in $H^+$. Now, to show V -> Ai $\in$ $(H')^+$, we must show that V -> Ai cannot use any of the FDs X -> Ai in its derivation. But this follows directly, since if X -> Ai is used to derive V -> Ai, then by lemma 1 V -> X, contradicting V -/> X in one of the transitive dependencies.

Since X -> V and V -> Ai are in $(H')^+$, thus f:X -> Ai is in $(H')^+$, for all such f. Hence, $(H')^+ = H^+$. □

Theorem 3 provides us with a simple means of removing an unwanted transitive dependency. Namely, excise the offending attribute from the relation. The theorem guarantees that the resulting schema still represents the given set of FDs.

That a transitive dependency can be removed so easily is rather surprising, since the FDs that form the schema are nonredundant. It would seem that excising an attribute should result in the loss of an FD. However, in synthesizing the schema from a nonredundant covering of FDs, we have implicitly added new FDs to the covering in step 4 of Alg. 1, and these FDs are explicitly embodied in the schema. If X and Y are the left sides of two distinct FDs in the nonredundant covering, H, with X -> Y and Y -> X in $H^+$, then X and Y are put into a single relation. This adds X->Y and Y->X as two new FDs that are explicitly embodied in the schema, even though they may not have appeared in the covering. For example, in fig. 4(ii) the FDs X1,X2 -> C,D and C,D -> X1,X2 are explicilty embodied in S1, even though the former FD is not in the covering. It is the addition of the extra FDs that allows us to excise a transitive dependency without affecting the closure of the embodied FDs.

Looking at theorem 3 in a different light, we can now modify Alg. 1 to synthesize schemas that are guaranteed to be in 3NF. Let H be the nonredundant covering resulting from step 2 of Alg. 1. Let J be the set of all FDs X -> Y such that X and Y are equivalent keys discovered in step 4 of Alg. 1. Let h:Z -> Ai, h $\in$ H, be embodied in Rk such that Ai appears in no synthesized key of Rk and Ai is transitively dependent upon a key of Rk. Then theorem 4 says that h is redundant; that is, h $\in$ $(H+J-\{h\})^+$. So, if we eliminate every FD h $\in$ H whose right side is not in any synthesized key and h $\in$ $(H+J-\{h\})^+$, then we will have eliminated all transitive dependencies. For if there were a nonprime Ai that is transitively dependent on a key of Rk, then theorem 4 guarantees that our extra redundancy check would have eliminated it. This leads us to our main result, Algorithm 2 (see figure 5), which synthesizes a provably 3NF schema.

Figure 5

## Algorithm 2

## Synthesizing a Relational Schema from a Set of FDs

1. (Eliminate extraneous attributes) Let F be the given set of FDs. Eliminate extraneous attributes from the left side of each FD in F, producing the set G. An attribute is extraneous if its elimination does not alter the closure of the set of FDs.

2. (Find covering) Find a nonredundant covering, H, of G.

3. (Partition) Partition H into groups such that all of the FDs in each group have identical left sides.

4. (Merge equivalent keys) Let J = $\phi$. For each pair of groups, say Hi and Hj, with left sides X and Y respectively, merge H1 and H2 together if there is a bijection X <-> Y in $H^+$. For each such bijection, add X -> Y and Y -> X to J. For each A $\in$ Y if X -> A is in H, then delete it from H. Do the same for each Y -> B in H with B $\in$ X.

5. (Eliminate transitive dependencies) Find an H' $\subseteq$ H such that $(H'+J)^+$ = $(H+J)^+$ and no proper subset of H' has this property. Add each FD of J into its corresponding group of H'.

6. (Construct relations) For each group, construct a relation consisting of all the attributes appearing in that group. Each set of attributes that appears on the left side of any FD in the group is a key of the relation. (Step 1 guarantees that no such set contains any extra attributes.) All keys found by this algorithm will be called <u>synthesized</u>. The set of constructed relations constitutes a schema for the given set of FDs.

Steps 1-4 of Algorithm 2 are effectively implemented as in Algorithm 1. Step 5 can be effectively implemented using the membership algorithm presented in part II. Algorithm 2 can then be implemented in the same $O(L^2)$ time bound as Algorithm 1. For details see section II.4.

## I.7 PROOF OF MINIMALITY

The purpose of this section is to examine the number of relations synthesized by Algorithm 2 (or 1) for a given set of FDs, compared with any other relational schema that represents those FDs. y We will show that all nonredundant coverings generate the same number of relations, by showing the number of equivalence classes of synthesized keys to be the same across all nonredundant coverings of a given set of FDs. This will then imply that the schemas synthesized by Algorithm 2 are minimal in the number of relations synthesized.

Lemma 3: Let G1 and G2 be two nonredundant sets of FDs with $G1^+=G2^+$. If g:X -> A is in G1, then there exists an h:Y -> B with h $\in$ G2 and with Y -> X and X -> Y in $G1^+$.

Proof If g $\in$ G1 and $G1^+=G2^+$, then g $\in$ $G2^+$. Hence, there is a derivation for g in G2. Each FD, h, used to derive g is in G2, so each such h is in $G1^+$.

If no h requires g for its derivation from G1, then we can construct a derivation for g in G1. The derivation is constructed by mimicking the derivation of g in G2, replacing each h in this derivation by h's derivation in G1. Since this derivation does not use g, g must be redundant in G1, a contradiction. So, at least one h must require g for its derivation from G1.

Say that h:Y -> B requires g:X-> A for its derivation from G1. Then, by lemma 1 X -> Y. Also, since h appeared in a derivation for g from G2, by lemma 1 Y -> X. Hence, h:Y -> B is in G2 with X -> Y and Y -> X, completing the proof. □

Lemma 2 cannot be strengthened so that X=Y. That is, one can have two nonredundant coverings with equivalent closures, such that the two coverings have different left sides representing a key equivalence class. For example, in figure 6 g3 and h3 have functionally equivalent left sides, since CF <-> DE; yet CF≠DE.

Using lemma 3 and recognizing that Algorithm 2 synthesizes a relation from each maximal group of FDs that have functionally equivalent left sides, we can now see that all nonredundant coverings of a given set of FDs produce the same number of relations by Algorithm 2.

Figure 6

Two Equivalent Coverings with Different Keys

```
G = {g1:C -> D,
     g2:D -> C,
     g3:CE -> F}

H = {h1:C -> D,
     h2:D -> C,
     h3:DE -> F}
```

G and H are nonredundant and $G^+=H^+$. Yet g3 and h3 generate different relations. This is an example of lemma 3 where $X \neq Y$.

Theorem 4: Let F be a set of FDs. Any two nonredundant coverings of F will produce the same number of relations via Algorithm 2.

Proof Let G1 and G2 be two nonredundant coverings of F. By lemma 3, if an FD g:X -> A is in G1, then there is an h:Y -> B in G2 with X -> Y and Y -> X. Thus, for any group of FDs in G1 with functionally equivalent left sides, there must be exactly one such group in G2, namely, the one that has the same functionally equivalent left sides. Since each such group generates one relation, G1 and G2 must produce the same number of relations.□

Theorem 4 states that all choices of nonredundant coverings are equally good in terms of number of relations synthesized. This is somewhat surprising in that it contradicts the intuition that perhaps a minimal-sized nonredundant covering would produce fewer relations than other larger nonredundant coverings.

The theorem also shows that on the logical level there is not very much choice as to how to pick relations that cover the given set of FDs. Some of the decomposition approaches (e.g., [10,15,17]) claim to allow the system to choose among a class of possible schemas, directing the choice by efficiency considerations. Since all coverings have the same set of equivalence classes of keys, the class of possible schemas is really quite small. Hence, if one is guided on the logical level by normalization considerations rather than by efficiency considerations, one arrives at a set of nearly identical possible schemas.

From theorem 4, we can see that the number of relations generated by Algorithm 2 is minimal among all those that embody the same given set of FDs. This gives a complete characterization of the optimal 3NF schemas discussed in [7].

Corollary: Let S be a schema synthesized from a set of FDs F using Algorithm 2. Let S' be any schema representing a set of FDs G that covers F. Then $|S'| \geq |S|$.

Proof Let H c G be a nonredundant covering of F. Certainly H will generate, via Algorithm 2, no more relations than are in S'. Furthermore, by theorem 4, Algorithm 2 will generate the same number of relations from G as from F. Hence, $|S'| \geq |S|$. □

## I.8 CONCLUSION

The purpose of this part of the paper was to develop an algorithm for synthesizing a 3NF schema from a given set of FDs and to examine some properties of such schemas. The main results were:

1. Certain simple algorithms for synthesizing schemas either produce too many relations or violate 3NF.

2. An algorithm that synthesizes provably 3NF schemas was presented. The essential aspect of this algorithm is that it eliminates as much redundancy as possible from the given set of FDs.

3. All nonredundant coverings produce the same number of relations using this latter method. Hence, synthesized schemas contain a minimal number of relations.

This is the first successful attempt, to our knowledge, of implementing Codd's normalization procedure [7] both provably and effectively. (Errors in two earlier similar attempts were isolated.) Furthermore, by the corollary to theorem 4, the sythesized relations satisfy Codd's optimality criterion -- no other schema covering the same FDs has fewer relations.

# PART II THE MEMBERSHIP PROBLEM

## II.1 INTRODUCTION

We have presented in part I an algorithm for synthesizing 3NF relational schemas from FDs. The usefulness of the algorithm depends on whether it can be implemented efficiently. It turns out that the problems of deciding if an attribute in a left side of an FD is extraneous or if an FD in a given set of FDs is redundant are instances of the general problem -- to decide if a given FD is in the closure of a given set of FDs. We call this latter problem the membership problem. The membership problem arises whenever the algebra of FDs is used, e.g., in the synthesis algorithm, in the context of key finding etc. In this part of the paper we present an efficient algorithm for the solution of the membership problem and show how it can be used in an implementation of the synthesis algorithm.

In section 2 we introduce the notion of a derivation tree as a model for derivations in the algebra of FDs. We prove some properties of derivation trees and, in particular, we prove the lemmas which were used in part I without proof. A derivation tree is essentially a graph representation of a derivation. It is true that one can discuss derivations without using derivation trees. However, derivation trees are coceptually simple, are easy to manipulate and their use simplifies the proofs considerably.

In section 3 we present the algorithm for the solution of the membership problem and prove that it works in linear time. It is known that the algebra of FDs is closely related to the propositional calculus. We explore this relationship to show that the algorithm can also be used to find tautologies in a restricted class of propositional calculus formulas, with the same time bound. In section 4 we present an implementation of the synthesis algorithm using the membership algorithm as a basic step. Under this implementation the synthesis algorithm works in quadratic time.

## II.2 DERIVATION TREES

### II.2.1 Derivation Trees as a Model for Derivations

Armstrong [1] proved that, given the FDs X -> B1,...,X -> Bk, the FD X -> B1,...,Bk can be derived using the axioms A1, A2, A3. It is trivial that each of the FDs X -> B1,...,X -> Bk is derivable from X ->B1,...,Bk. Thus, the FD X -> B1,...,Bk is equivalent to the set {X ->B1,...,X -> Bk}. In this section we will regard an FD of the form X -> B1,...,Bk merely as a representation of the FDs X -> B1,...,X -> Bk.

Let X be a set of attributes, let G be a set of FDs over X, and let g:B1,...,Bk -> C be an FD over X. If $g \in G^+$, then there is a sequence of applications of axioms A1, A2, and A3 on G that yields g. In this section we will develop a graph model, called a derivation tree, for such a sequence of applications of the axioms.

Let G be a set of FDs. <u>G-based derivation trees</u> (abbr. G-based DT) are formally defined as follows:

1. If C is an attribute, then the labelled node C is a G-based DT.

2. If T is a G-based DT with C as a leaf node, and f:B1,...,Bm -> C is an FD in G, then the tree constructed from T by adding B1,...,Bm as children of the leaf node C is also a G-based DT.

The derivation tree is a simple model for the successive composition of FDs by pseudotransitivity (this is formalized below). A sample derivation tree construction is given in figure 7.

A DT is characterized by its root, by its leaf set and by the FDs that appear in it. We will abbreviate the expression "a DT whose leaf set is contained by {A1,...,An}" by "an {A1,...,An}-DT". If T is an X-DT rooted at A then we call it a "derivation tree for the FD X -> A". (This terminology will be justified by theorem 5 below.)

The following lemma is an important step towards a formal characterization of the connection between derivation trees and derivations of FDs. It is a formal restatement, using DTs, of lemma 1 which was used extensively in proving the main theorems of part I.

<u>Lemma 4</u>: Let T be a G-based derivation tree. Let X be a nonempty subset of the nodes of T and let Y be the set of all attributes that appear as leaves of T. Then Y -> X is in $G^+$.

<u>Proof</u> Consider first the case that X is simply the root node. This sublemma can be proved by induction on the number of FDs that are added to the DT (i.e., applications of (2) above). This follows directly since each such addition preserves the desired property that the root is functionally dependent upon the set of leaves by virtue of the psuedotransitivity rule.

Now suppose Xi ∈ X is any internal node of T. Since Xi roots a Y-DT, by the above sublemma we have Y -> Xi. By the observation at the begining of the section, if Y -> Xi for all Xi ∈ X, then Y -> X, completing the proof. □

To make the DT model complete with respect to Armstrong's axioms we have to consider axioms A1 (reflexivity) and A2 (augmentation) as well. Except for FDs of the form X->X,

Figure 7

A Sample Derivation

Given: G = {g1:AB->C; g2:C->D; g2:DE->F; f4:A->E}

Show: f:AB -> F $\in G^+$

```
   FD Used            Derivation            Current FD
in this Step      Tree Construction      Represented by
                                            the Tree


     g3                   F                  DE -> F
                          |
                      ---------
                      |       |
                      D       E

     g4                   F                  DA -> F
                          |
                      ---------
                      |       |
                      D       E
                              |
                              A

     g2                   F                  CA -> F
                          |
                      ---------
                      |       |
                      D       E
                      |       |
                      C       A

     g1                   F                  AB -> F
                          |
                      ---------
                      |       |
                      D       E
                      |       |
                      C       A
                      |
                  --------
                  |      |
                  A      B
```

any FD that can be derived with reflexivity and psuedotransitivity can also be derived without reflexivity simply by eliminating all the applications of axiom A1. Therefore, we may assume without loss of generality that reflexivity is not used in derivations except for deriving FDs of the form f:X->X. Similiarly, in a DT, reflexivity corresponds to taking a leaf node, making a copy of it, and connecting the copy as a child of the original leaf. Clearly, this rule can add no new nodes to the leaf set of a DT, and hence is basically a null operation and need not be included in the definition of a DT. The FDs of the form X -> X are handled by part (1) of the DT definition.

Augmentation corresponds to the addition of extra leaf nodes connected to an internal node of the DT. All of the children of any node that was added by augmentation could themselves have been added by augmentation. Consider a DT in which augmentation was used to produce what is now a non-leaf node, E, of the tree. One can eliminate E from the tree by replacing it by all of its descendants that are leaves. Doing this to all internal nodes that were produced by augmentation yields a DT in which all applications of augmentation produce leaves. Similiarly, one application of augmentation at the very last step of a derivation is all that is needed to derive any derivable FD. Therefore, we do not need to use augmentation in DTs; after a DT is constructed we can simply add any attribute to the left side of the FD it represents. This leads us to the following theorem for the completeness of DTs.

<u>Theorem</u> <u>5</u>: For a given FD g:X -> C and a set of FDs G, g $\in$ G+ if and only if there is a G-based X-DT, T, rooted at C.

<u>Proof</u> Let T be a G-based X-DT rooted at C. T represents an FD X' -> C in G+ where X' $\subseteq$ X. Hence, by lemma 4 and augmentation, g $\in$ G+. To prove the converse, we know that if g $\in$ G+, then there is a sequence of (say) N applications of Armstrong's axioms yielding g from G. From the above discussion, we can assume there are no applications of reflexivity in the sequence, and that applications of augmentation are all postponed to the last step. Thus, the first N-1 steps are all applications of pseudotransitivity and can be simulated by a G-based X-DT rooted at C.□

By the theorem, DTs serve as a model for derivations of FDs in which the right side is a single attribute. However, to show that X->Y1...Yk is derivable, it is sufficient to construct derivation trees for X->Y1,...,X->Yk. Thus, the concept of a DT is general enough for our needs.

### II.2.2 <u>Additional Properties of Derivation Trees</u>

Using theorem 5 and lemma 4, we can now prove lemmas 1 and 2 which were stated in Part I without formal proofs.

Lemma 1 (cf. section I.4.1): Let G be a set of FDs, and let g:X -> Y be an FD in G. If h:V -> W is in $G^+$ and g is used for some derivation of h from $G^+$, then V -> X is in $G^+$.

Proof Without loss of generality we may assume that W is a single attribute. If h $\in$ $G^+$ and g is used in some derivation of h, then there is a G-based V-DT, T, rooted at W in which g appears. Every attribute of X is a node of T. Hence, we can apply lemma 4 to obtain V -> X. □

The proof of lemma 2 uses lemma 1 and the fact that each $\theta$ attribute appears in only one FD.

Lemma 2 (cf. section I.5.3): If X -> $\theta$ is in a set of FDs G, then for any nonredundant covering H of G, either X -> $\theta$ is in H or Y -> $\theta$ is in H, where Y -> X and X -> Y.

Proof If X -> $\theta$ is in H, then we are done, so assume not. Since H covers G, there must be a derivation for X -> $\theta$ from H. Let Y -> $\theta$ be the root FD of an H-based derivation tree for X -> $\theta$. By lemma 1, X -> Y. To show Y -> X, we examine $G^+$. In G, X -> $\theta$ is the only FD containing $\theta$. Thus, any derivation for any FD in $G^+$ with $\theta$ on the right side must use X -> $\theta$ as the root FD. In particular, X -> $\theta$ is the root FD of any derivation for Y -> $\theta$ from G. Hence, by lemma 1, Y -> X. □

A priori, derivation trees can be arbitrarily large. The follwing lemma states that, for all practical purposes, we can restrict our attention to "small" trees. The rationale behind the lemma underlies our work in the next section. The lemma is essentially the same as a well known result about derivation trees in the theory of context free languages.

Lemma 5: If g $\in$ $G^+$ then there exists a G-based derivation tree for g, T, such that in T no path from the root to a leaf contains more than one occurrence of any attribute.

Proof Suppose g $\in$ $G^+$ and let T1 be a G-based derivation tree for g. If T1 satisfies the condition of the lemma, then T = T1 and we are done. Otherwise, let us look at some path (A1,...,Ak), where A1 is the root and Ak is a leaf, such that the path contains at least two occurrences of some attribute A.

Let Ai and Aj be two occurrences of A where i < j. If we excise from T1 the subtree rooted at Ai (=A) and replace it by the subtree rooted at Aj (=A), then we obtain a tree T2 such that:

1. All FDs used in T2 belong to G.

2. The set of leaves of T2 is a subset of the set of leaves of T1.

3. T2 and T1 have the same root.

4. T2 has fewer nodes than T1.

Therefore, T2 is a G-based derivation tree for g and is smaller than T1.

This process of replacing subtrees by smaller subtrees can be continued as long as the trees produced do not satisfy the condition of the lemma. Since these trees contain fewer and fewer nodes, the process must terminate. The last tree produced by the process satisfies the condition and is the required tree T. □

## II.3 A LINEAR TIME MEMBERSHIP ALGORITHM

### II.3.1 Overview of the Membership Problem

The membership problem for a set G of FDs is: Given an FD g, decide if $g \in G^+$. In this section we present an algorithm that solves the problem in time proportional to the size of G. In the following we assume, without loss of generality, that the right side of g is a single attribute.

Since $g \in G^+$ if and only if there exists a G-based derivation tree for g, one obvious way to solve the problem is to try to find such a derivation tree or, at least, prove that one exists. From lemma 5 it follows that a search for such a derivation tree must terminate. For any set G of FDs, the number of attributes in G is finite and, therefore, the number of G-based derivation trees satisfying the condition of lemma 5 is finite. Given g, one can decide if $g \in G^+$ by enumerating these trees and checking each one to see if it is a derivation tree for g. However, the number of these trees may be quite large and this enumeration algorithm may be too time consuming.

A more feasible approach to the problem is to try to construct a derivation tree for g step by step. While we may occasionally make a mistake by trying an FD that is not used in a DT for g, if the number of such mistakes is not too large then the resulting algorithm can be quite efficient. In [2] a bottom-up algorithm for constructing a DT for g is presented that works in time roughly proportional to the square of the size of G. In the next subsection, we present an improved version of this algorithm that works in time linearly proportional to the size of G.

### II.3.2 The Algorithm

Let G = {g1,...,gn} be a set of FDs involving attributes from the set {A1,...,Am}. We assume that G is given as a string of pairs where each pair represents an FD and consists of a left side and a right side. Each side is a sequence of attributes. Attributes are represented as numbers in the set {1,...,m}. The length of the string representing G will be denoted by |G|.

Let $g: B1, \ldots, Bk \rightarrow C$ be given where $\{B1, \ldots, Bk, C\} \subseteq \{A1, \ldots, Am\}$. To check if $g \in G^+$ we try to compute the set of attributes that are functionally dependent upon B1,...,Bk in G. Then, $g \in G^+$ if and only if C is in this set. This can be done as follows.

We use a set variable, DEPEND, to hold attributes that are functionally dependent upon B1,...,Bk. Initially, we set DEPEND to {B1,...,Bk} as, clearly, each Bi is functionally dependent on this set (by reflexivity and augmentation). Given the set DEPEND, we look for an FD in G such that its left side is contained in DEPEND but its right side is not. Since every attribute on its left side is functionally dependent upon B1,...,Bk, so is also, by pseudotransitivity, every attribute on its right side. Therefore, the attributes on its right side are added to DEPEND. (Conceptually, we start with DEPEND containing the leaf set of a DT for g. Each time we find a node whose children are all in DEPEND we add the node to DEPEND.) This operation is iterated until no new attributes to be added are found. Then DEPEND contains all attributes that label nodes of {B1,...,Bk}-DTs and $g \in G^+$ if and only if $C \in$ DEPEND. The method is formally implemented as Algorithm 3, see figure 8. (For brevity, we use in the algorithm the abbreviations LS, RS for left side and right side, respectively.)

To analyze the time complexity of the algorithm we note that in each iteration of the OUTER loop (except when the final iteration results in FOUND = FALSE) at least one attribute is added to DEPEND. In the worst case, the number of iterations of OUTER may be close to m. In any such iteration, the INNER loop scans the input string G. Therefore, in the worst case, the total time spent by the algorithm may be proportional to $m|G|$. (Actually, it seems that even more time is required. However, in a clever implementation this time bound can be achieved. We will not go into the details now, as we will present a more efficient algorithm.) We will now try to improve the algorithm so as to reduce this time bound.

The algorithm is obviously inefficient. First, we note that, when an FD gi satisfies the condition in INNER, its right side is added to DEPEND. The values assumed by DEPEND form a monotonically increasing sequence of sets; so gi will not satisfy the condition a second time and it need not be checked in future iterations. Another problem is the fact that each attribute on the left side of each FD is checked in each iteration of OUTER. Now, once an attribute on the left side of an FD is known to be in DEPEND, it is redundant to check repeatedly this attribute's membership in DEPEND.

These two problems can be solved by appropriate changes to the algorithm. The first problem was actually solved in the algorithm presented in [2]; the second can be solved similiarly.

Figure 8

ALGORITHM 3

Deciding the Membership Problem

INPUT: A set G of FDs and an FD g:B1,...,Bk -> C.

OUTPUT: 'yes' if g ∈ G+, 'no' if G ¬∈ G+.

```
/*  Data structures  */

DEPEND:        The set of all attributes found to be dependent
               on {B1,...,Bk} so far.

(FLAG, FOUND): Boolean variables;


/* Initialize */

DEPEND = {B1,...,Bk};
FLAG = TRUE;


/* Build up the set DEPEND */

OUTER: do while (FLAG);

          FOUND = FALSE;

          INNER: do for each gi ∈ G;

                    if (LS(gi) ⊆ DEPEND &
                        RS(gi) ¬⊆ DEPEND)
                    then do;
                           add RS(gi) to DEPEND;
                           FOUND = TRUE;
                          end;

                 end INNER;

        FLAG = FOUND;

        end OUTER;


/* Print results */

PRINT: if (C ∈ DEPEND)
          then print 'yes';
          else print 'no';
```

However, even with these two changes, the worst case time bound would still be O(m|G|). The reason is that these problems are special cases of a more general problem, namely, that in each iteration of OUTER all of G is scanned although only a small part of G is actually involved in any operation. Thus, if G contains FDs whose left side contain attributes which are not derivable from {B1,...,Bk}, then these FDs will be scanned in each iteration. Also, even in the algorithm presented in [2], the left side of an FD that eventually does appear in a DT for q is unnecessarily scanned many times before its right side is added to DEPEND.

Our problem then is to find a way to change the algorithm such that the following holds: In each iteration of OUTER an FD is visited only if there is some operation to be performed on it in that iteration, and only the attributes that are actually involved in the operation are visited.

The basic operation on an attribute on a left side of an FD is to "mark" it as belonging to DEPEND. The basic operation on an FD is to add its right side to DEPEND if its left side is contained in DEPEND. As new attributes are added to DEPEND in each iteration, it seems reasonable to visit an FD only if its left side contains an occurrence of an attribute that was added to DEPEND in the previous iteration. Each such occurrence is "marked" and never visited again (and, if all attributes on the corresponding left side are marked, then the right side is added to DEPEND and is also never visited again). The question is how can we arrange to visit only these occurrences of attributes on the left sides of FDs; that is, how can we locate them without scanning all of G?

To solve this problem, we propose to use linked lists that are threaded through the input string. For each attribute in {A1,...,Am}, we have a linked list of all occurrences of that attribute on left sides of FDs. Thus, each attribute on the left side of an FD appears in excatly one such list. After an attribute is added to DEPEND we can follow the links on its corresponding list, "mark" each occurrence of the attribute on the linked list as being derivable from B1,...,Bk, and check if the FD in which it appears now satisfies the condition in INNER.

The left side of an FD is in DEPEND if and only if all of the attributes on that left side are marked. To determine this latter condition, we maintain a counter for each FD. The counter holds the number of attributes that are on the left side of the FD and do not yet belong to DEPEND. "Marking" an occurrence of an attribute in an FD then reduces to decrementing the counter of that FD; checking whether the left side of the FD is contained in DEPEND reduces to comparing the value in the counter to zero. If the counter can be accessed directly from the occurrence of the attriubte, then these two operations can be done in a constant number of steps.

Notice that each occurrence of an attribute in the input string is visited at most once, since an attribute is added to

Figure 9

## ALGORITHM 4

### Deciding the Membership Problem

INPUT: A set G of FDs and an FD g:B1,...,Bk -> C.

OUTPUT: 'yes' if g ∈ G+, 'no' if g ¬∈ G+.

```
/*  Data Structures  */

FD(1:n):        FD(i) is a structure describing the i-th FD,
                consisting of an integer COUNTER and a RIGHT_SIDE
                containing an attribute.

ATTRLIST(1:m): a singly-linked list of those FDs with Aj on their
                left hand sides.

DEPEND:         the set of all attributes found to be dependent
                upon {B1,...,Bk} so far.

NEWDEPEND:   the subset of DEPEND that has not yet been examined.


/* Preprocess G to build ATTRLIST and FD */

do for each gi ∈ G;

   do for each Aj ∈ LS(gi);

      add gi to ATTRLIST(j);
      increment COUNTER of FD(i) by 1;

   end;

   set RIGHT_SIDE of FD(i) to be RS(gi);

end;
```

Figure 9 - page 2

```
/*          Main body of the algorithm          */

/* Initialize */

DEPEND = {B1,...,Bk};
NEWDEPEND = DEPEND;


/* Build up the set DEPEND. */

OUTER: do while(NEWDEPEND is not empty);

           remove an attribute from NEWDEPEND
             and assign it to the variable NEXT_TO_CHECK;

   INNER: do for each gi on ATTRLIST(NEXT_TO_CHECK);

             decrement COUNTER of FD(i) by 1;
             if ((COUNTER of FD(i) = 0) then
                do for each attribute A in RIGHT_SIDE of FD(i);

                       if (A ¬∈ DEPEND) then
                          add A to DEPEND and NEWDEPEND;

                     end;

           end INNER;

       end OUTER;


/*  Print results  */

PRINT: if (C ∈ DEPEND)
          then print 'yes'
          else print 'no'
```

DEPEND at most once. Attributes that are not derivable from {B1,...,Bk} will not be visited at all. This solves all the problems mentioned above. Most important, though, is the fact that since each visit to an occurrence of an attribute takes a bounded number of steps and each occurrence is visited at most once, the algorithm takes time linearly proportional to |G|. The algorithm is formally implemented as Algorithm 4, see figure 9.

REMARK We have assumed that the right side of g is a single attribute. This assumption was for simplicity of presentation only and is not essential to the algorithm. If the right side of g is C1,...,Cp then the only change needed is to check in the printing stage if C1,...,Cp are elements of DEPEND.

### II.3.3 Analysis of the Algorithm

In this section we prove that Algorithm 4 is correct and analyze its time complexity.

The preprocessing step of the algorithm consists of a single scan of G. For each occurrence of an attribute, a constant number of steps is performed. Therefore, this part terminates and takes time O(|G|). After termination of this part the following hold:

1. For each gi ∈ G

   a. The value of the COUNTER of FD(i) is equal to the number of attributes on the left side of gi.

   b. The set RIGHT_SIDE of FD(i) contains the attributes on the right side of gi.

2. For each attribute Aj, each FD that contains Aj on its left side is on ATTRLIST(j).

We now turn our attention to the second part -- the main body of the algorithm. We present an informal proof that this part terminates and produces correct results.

The sets DEPEND and NEWDEPEND are initialized to {B1,...,Bk}. The only other place that an attribute can be added to these sets is in the if-statement in the INNER loop. Since attributes are never deleted from DEPEND, it follows from the condition in the if-statement that an attribute can be added to these sets at most once. In each iteration of OUTER one attribute is removed from NEWDEPEND so the number of iterations is exactly the number of attributes added to the sets and is at most m. In any iteration of OUTER, the INNER loop has the form "do for each element of a finite set", and must therefore terminate. From this it follows that the algorithm terminates

To prove correctness, we start with the following observation. Before each test of the condition of OUTER, the COUNTER of FD(i), for each i, is equal to the number of attributes on the left side of gi that either are not in DEPEND or are in the intersection of DEPEND and NEWDEPEND. This is obviously true at the time of the first test, since then DEPEND = NEWDEPEND and each COUNTER is equal to the number of attributes on the left side of the corresponding FD (see 1a. above). In an iteration of OUTER one attribute is removed from NEWDEPEND and the COUNTER of any FD that contains it on its left side is decremented; so the claim is true after the iteration.

It is easy to show that each attribute added to DEPEND is dependent by G on B1,...,Bk. This is trivially true for B1,...,Bk. Any other attribute is added only if it appears on the right side of some gi and the COUNTER of FD(i) is 0. By the above observation, this means that all attributes on the left side of gi have already been added to DEPEND and are therefore (by induction hypothesis) derivable from B1,...,Bk. Thus the new attribute is also derivable from them.

Finally, we show that all attributes that depend by G on B1,...,Bk will be in DEPEND when the algorithm terminates. We will use induction on the depth of derivation trees. For derivation trees of depth zero, we have to consider only B1,...,Bk and they are all in DEPEND. Given an attribute that has a derivation tree of depth i+1, we look at the root FD of the tree, gj:D1...Dp->E. The attributes D1,...,Dp all have derivation trees of depth ≤ i. Therefore, each of these attributes is added to DEPEND. Now, when the last of these D's is removed from NEWDEPEND, the COUNTER of FD(j) will be set to zero so E will be added to DEPEND unless it is already there. This concludes the correctness proof.

Having proved the correctness of the algorithm, we can now add a shortcut. When the attribute C is added to DEPEND, we exit from OUTER and proceed directly to the output step. Obviously, this can only lead to a faster algorithm.

We have already seen that the preprocessing stage takes time proportional to |G|. In the main body of the algorithm each attribute in NEWDEPEND is removed exactly once. The processing in the corresponding iteration of OUTER consists of a constant number of steps performed for each occurrence of the attribute on a left side of an FD. Similiarly, the RIGHT_SIDE of FD(i) is visited at most once and then a constant number of steps is performed. Therefore, the algorithm works in time O(|G|). (We have assumed that |g| ≤ |G|.)

While the worst case time of the main body of the algorithm is O(|G|), this is not always the best estimate. If G contains many FDs that do not contain attributes derivable from B1,...,Bk, then these FDs will not be visited at all. Also, the running time of the algorithm depends on the depth of derivation trees of g. If g has a shallow derivation tree, its right side will be added to DEPEND at an early stage and the algorithm will be faster. (It can be shown that all attributes that have G-based {B1,...,Bk}-derivation trees of depth ≤ i are added to DEPEND before any attribute that has only derivation trees of depth >i is added.) These considerations are of importance in cases where many membership tests based on one group of FDs are performed. Preprocessing can be done once and then only the body need be applied for each FD being tested.

Theorem 6: Membership in the closure of a set of FDs can be tested in linear time. □

REMARK We have presented here a bottom-up algorithm. We have also developed a linear time top-down algorithm. In principle, it works by a left-to-right depth-first expansion of the required DT, starting with the root. However, it is quite complicated and a description of it will not be given.

### II.3.4 An Algorithm for a Restricted Class of Tautologies

It is known that there is a close relationship between the theory of FDs and the propositional calculus. This fact was originally observed by Delobel and Casey [10]. Fagin, in a recent paper [11], has given a clear statement of this relationship and presented two alternative proofs. We will now show that it follows from this relationship that tautologihood can be decided for a restricted class of propositional formulas (to be defined later) in linear time.

In the following we use => as the implication symbol of the propositional calculus. Let N be the following mapping from FDs and sets of FDs to propositional formulas:

For g:B1,...,Bk -> C1,...,Cp    N(g) = B1&...&Bk => C1&...&Cp

For G = {g1,...,gn}    N(G) = N(g1)&...&N(gn)

The relationship as stated by Fagin is:

Theorem (Fagin [11]): For any FD g and for any set G of FDs, g ∈ G+ if and only if N(g) is derivable in the propositional calculus from N(G).

Proof Proofs can be found in [11]. We will sketch briefly the idea behind the first proof. The transformation N is essentially a translation of FDs into the language of the propositional calculus. Fagin showed that if Armstrong's axioms are similiarly translated the result is a complete set of axioms for the propositional calculus. Therefore, a derivation of g from G, when translated, is a proof of N(g) from N(G) that uses these translated axioms. Similiarly, a proof of N(g) from N(G) which uses the translated axioms can be translated back to a derivation of g from G. □

In the propositional calculus a formula A is derivable from a formula B if and only if the formula A => B is a tautology. Thus the theorem can be stated as follows: "g ∈ G+ if and only if N(G) => N(g) is a tautology". It now follows that to check if a formula of the form N(G) => N(g) (for some G, g) is a tautology one can use Algorithm 4. One way of doing it is to translate the problem back into a membership problem. It is also easy to see that the algorithm can be applied directly to propositional formulas of that form.

Let us denote the class of propositinal formulas in disjunctive normal form in which each disjunct includes at most one negated literal by DNF(1-neg). We now show that tautolcgihood of formulas in this class can be checked in linear time.

<u>Theorem</u> <u>7</u>: The tautology problem for DNF(1-neg) can be decided in time linearly proportional to the size of the input.

<u>Proof</u> Let f be a given formula in DNF(1-neg). We first observe that if f contains a non-negated literal in each disjunct then f is not a tautology; it can be falsified by assigning the value FALSE to all the literals. Therfore , we assume that some disjuncts of f do not contain a non-negated literal. These disjuncts must then cosist of a single negated literal. We also assume that f contains at least one disjunct which consists of a single non-negated literal. (If there is no such disjunct , we can add one. For if C is a new literal that does not appear in f then f is a tautology if and only if C | f is a tautology.)

Our strategy will be to transform f into an equivalent formula of the form N(G) => N(g) for suitable G and g. There are several cases to consider:

1. Disjuncts that include both a negated literal and a non-negated literal. Let E1&...&Ek&¬D be such a disjunct. We replace it by the equivalent disjunct ¬(E1&...&Ek => D)

2. Disjuncts that include at least two literals but no non-negated literal. These disjuncts cannot be transformed directly. Let E1&...&Ek be such a disjunct and let D be a new literal which does not appear in f. We replace the disjunct by E1&...&Ek&¬D and add to f the disjunct consisting of the single literal D. Clearly, the new formula thus obtained is a tautology if and only if f is a tautology. Now E1&...&Ek&¬D can be replaced as in 1.

3. Disjuncts consisting of a single negated or non-negated literal. Let these disjuncts be C1 | ... | Cp | ¬B1 | ... | ¬Bk. We replace them by the single equivalent disjunct B1&...&Bk => C1&...&Cp.

When the above transformations are completed we have a formula of the form

N(g) | ¬N(g1) | ... | ¬N(gn)

for some g,g1,...,gn. This formula is equivalent to the formula N({g1,...,gn}) => N(g). Now, the algorithm can be applied to this formula. Since all the transformations described above can be performed in linear time, the theorem follows. □

We have chosen a proof by way of reduction for ease of presentation only. The algorithm can be applied directly to formulas in DNF(1-neg). Some changes in the terminology used in the algorithm will be necessary; also, a consideration of the possible types of disjuncts will have to be incorporated in the algorithm. We leave it to the reader to rewrite the algorithm for DNF(1-neg).

We note that the theorem is true also, by symmetry, for the class of formulas in which each disjunct contains at most one non-negated literal. However, the theorem can not be generalized to obtain, say, a polynomial time bound for the tautology problem for the class of formulas with at most two negated literal per disjunct. The problem for this class is as general as the tautolgy problem for formulas in DNF which is known to be NP-complete.

## II.4 IMPLEMENTATION OF THE SYNTHESIS ALGORITHM

### II.4.1 Preprocessing

In this section we present an implementation of the synthesis algorithm, Algorithm 2 (cf. section I.6.4), using the membership test as a basic operation. While this implementation seems to be natural and quite efficient, others are conceivable. In any case, the proofs of the properties of schemas synthesized by Algorithm 2 do not depend on how it is implmented.

In the first two steps of the algorithm, extraneous attributes and redundant FDs are eliminated. Each time an attribute (or an FD) is eliminated, we obtain a new set of FDs. When the membership test is applied to this new set, the preprocessing has to be redone. However, the difference between the old set and the new set is quite small. It would be much more efficient if preprocessing is done only once and each time the set of FDs is changed, only the appropriate changes in its preprocessed form are performed.

If, instead of singly-linked lists, we use doubly-linked lists to connect all occurrences of each attribute, then eliminating an attribute can be done in a constant number of steps. Eliminating an FD is done by eliminating all attributes on its left side. Thus, the total time spent in the first two steps of the algorithm in eliminating attributes and FDs will be proportional to the number of attributes eliminated and is proportional to the size of the input in the worst case.

Using the same (or almost the same) preprocessed input for several membership tests means that the counters will have to be reset for each test. This also can be done efficiently, e.g., by having two counters for each FD, one of which always contains the original value. In what follows we assume, therefore, that the input has been preprocessed once and for all. Each time an attribute (or an FD) is eliminated, the necessary local changes are performed. To apply the membership test will mean to apply the main body of the membership algorithm.

II.4.2 Implementation

Let f:A1,...,Ak -> B be an FD in F. The attribute Ai is extraneous in f (cf. section I.5.1) if A1,...,Ai-1, Ai+1,...,Ak -> B is in F+. The procedure for eliminating extraneous attributes is shown in figure 10.

Note that during the execution of the IN group the left side of g changes dynamically. The loop is performed exactly once for each attriubte in the original left side of g. It is quite clear that each time an attriubte is eliminated from the left side of g, the closure of the resulting set of FDs is the same as F+. (The FD g is replaced by g' such that g' is in G and g is derivable from g' by augmentation, therefore G+ = (G - {g} U {g'})+.) To prove the correctness of the algorithm it suffices to show that after it terminates, the left side of any g in G does not contain extraneous attributes.

Suppose the left side of g after termination is A1,...,Aj and that Ai is extraneous. This means that A1,...,Ai-1,Ai+1,...,Aj -> RS(g) is in G+. But then, when IN was executed for Ai, the left side contained A1,...,Ai-1,Ai+1,...,Aj so Ai should have been eliminated -- a contradition.

We use a similar procedure to implement the second step of Algorithm 2 -- elimination of redundant FDs (see figure 11). Here again the membership test is performed once for each FD in the original set G and H changes dynamically. It is obvious that the closure remains the same throughout the execution of the procedure. Also, after termination of the procedure no h ∈ H is redundant. If h ∈ (H - {h})+ then for any set H' containing H, h ∈ (H' - {h})+, so h should have been eliminated when it was tested for redundancy.

The implementation of step 3 (partition) is straightforward. All left sides are arranged in a sequence and each one is compared to the left sides preceding it in the sequence. However, a more efficient implementation exists. It will be described in the next section. For step 4 (merge equivalent keys), we again use the membership test to check if two given left sides are equivalent. For step 5 we can use the same procedure that was used for step 2. We first order the set H+J such that all elements of H come first and we perform the redundancy check only for elements of H. Implementation of step 6 is also straightforward.

Figure 10

<u>Implementation of Step 1 of Algorithm 4</u>

<u>Eliminating Extraneous Attributes</u>

```
input = F;

G = F;

OUT: do for each g ∈ G;

     IN: do for each attribute A in LS(g);

            if (LS(g) - {A}) -> RS(g) is in G+
               then eliminate A from LS(g);

        end IN;

     end OUT;

output G;
```

Figure 11

<u>Implementation of step 2 of algorithm 4</u>

<u>Finding a Nonredundant Covering</u>

```
input = G

H = G;

do for each h∈H;

    if h∈(H - {h})+
      then H = H - {h};

end;

output H;
```

Assuming that the "do for each" groups are executed, say, from left to right, then the output H of the second step depends not only on the input F to the first step, but also on the order of FDs in F and the order of attributes on the left sides of the FDs in F. Different representations of the same set F may lead to different nonredundant coverings. However, as we have mentioned before, the properties of the synthesized schemas do not depend on any particular implementation or order of operations. Furthermore, the results of section I.7 imply that we obtain (up to equivalent keys) essentially the same schemas independent of which covering is chosen.

II.4.3 <u>Analysis of the Implementation</u>

We will now analyze the time complexity of the synthesis algorithm under the above implementation. Preprocessing the input takes time $O(|F|)$. Step 1 consists of a membership test for each attribute on a left side of an FD, so it takes time $O(|F|^2)$. Step 2 consists of (at most) a membership test for each FD, so it takes time $O(n|F|)$, where n is the number of FDs in F. For step 3 we note that there are $O(n^2)$ comparisons to be made. Comparing any two left sides takes time $O(m)$ so step 3 should take time $O(mn^2)$. However, we shall sketch an implementation under which step 3 takes only time $O(mn)$.

The problem can be formulated as follows: We have n sets S1,...,Sn which are subsets of {A1,...,Am} and are possibly not all distinct. We want to partition these sets into classes such that all sets in any class are equal, sets in different classes are not equal and every set belongs to one class.

We first construct an n by m matrix where the (i,j) element of the matrix is 1 if Aj $\in$ Si and is 0 if Aj $\neg\in$ Sj. This construction can be done in $O(mn)$ steps. Then we put all the sets S1,...,Sn into one class which is represented as a list. Now, we enter the partitioning process.

Suppose we already have a partition according to A1,...,Aj. That is, we have several lists such that for any pair of sets from two different lists one of A1,...,Aj belongs to one set of the pair but not to the other set; also, for any pair of sets from the same list they contain the same elements of A1,...,Aj. We now perform a partition according to Aj+1. We split each list into two lists. In the first list we put all sets of the original list which contain Aj+1; in the second list we put the sets which do not contain Aj+1. Empty lists are discarded, so the total number of lists is at most n. After m iterations of this process we have the required partition.

Now, the partition process for one attribute which we have just described consists essentially of scanning one column of the matrix so it takes $O(n)$ steps. Thus, the time spent in the whole partition process, including the construction of the matrix, is $O(mn)$.

Having at most n groups generated in step 3, step 4 consists of at most $O(n)$ membership tests, so it takes time $O(n|F|)$. In step 5 we perform at most n membership tests, since only elements of H are tested. However, the test is based on the set H+J. Thus, to estimate the time spent in a membership test, we need an estimate of the size of J.

The size of J depends on the way J is constructed in step 4. Let {X1,...Xk} be an equivalence class, where the elements of the class are listed in the order in which they have been added to the class. Let us assume that each time we want to check if a given left side belongs to the class, we compare it to the last element added to the class. Then the FDs in J that correspond to this class are X1<->X2, X2<->X3,.... Any left side appears in at most two equivalences in J (e.g., X2 appears in X1<->X2 and in X2<->X3). An equivalence X<->Y can be represented by the FDs X->Y and Y->X. Each left side of an FD in H appears in at most four such FDs and it follows that $|J| = O(|H|) = O(|F|)$.

Step 5 takes, then, $O(n|F|)$. Step 6 is free, if appropriate operations are performed in the previous steps, i.e., equivalence classes can be represented as relations when step 4 is performed and redundant attributes in these relations are excised in step 5. Thus, the total time spent in the algorithm is

$$O(|F|) + O(|F|^2) + O(n|F|) + O(nm) + O(n|F|) + O(n|F|)$$

$$= O(\max(|F|^2, n|F|, nm))$$

$$= O(|F|^2).$$

# PART III THE BCNF AND THE KEY FINDING PROBLEMS

## III.1 INTRODUCTION

### III.1.1 A Survey of the Results

In the first two parts of this work we have exhibited the feasibility of the algorithmic approach to the problem of synthesizing 3NF schemas. We turn now to two related problems - Boyce Codd normal form and the existence of additional keys. In both cases , our results strongly suggest that efficient algorithms for the treatment of these problems do not exist.

A new normal form, called Boyce-Codd normal form (abbr. BCNF), which is strictly stronger then 3NF was presented in [8]. We have seen that any set of FDs can be represented by a 3NF schema and there exists an algorithm that produces such a schema for any given set. We would like to know if similiar results hold for BCNF. That is, can any set of FDs be represented by a BCNF schema and can the synthesis algorithm or an extension thereof be used to produce BCNF schemas? These problems are treated in section 2. We show that BCNF violations are inherent in some sets of FDs; for these sets no BCNF schema exists. We also show that even when a BCNF schema exists for a given set of FDs, the algorithm may produce a schema which is not in BCNF. Finally, we prove that the problem whether there is a BCNF violation in a given relation (where a set of FDs is also given) is NP-complete. This is true even when it is known that the relation was produced by the synthesis algorithm from the given set of FDs. These results imply that extending the synthesis algorithm to produce a BCNF schema, even when such a schema does exist, is probably not computationally feasible.

A relation in a schema contains one or more designated keys. These keys may be specified by the user or produced by the synthesis algorithm. In addition to these keys, other keys may exist in a relation by virtue of the given set of FDs. In section 3 we treat problems relating to such additional keys. We show that, given a set of FDs and a relation with some designated keys, there may exist in the relation additional keys. Again, this is true even when the relation and its keys are produced by the synthesis algorithm from the given FDs. We also prove that the problem of deciding whether an additional key exists in a given relation is NP-complete. As in the case of the BCNF violation, these results seem to imply that key finding is an inherently difficult problem. Finally, some results by Lucchesi and Osborne [14], based on a restricted form of our model, are compared to the other results in the section.

### III.1.2 A Review of Definitions

In order to give an adequate and precise treatment of the problems mentioned above we will review here some of the definitions from part I (see section I.2).

A relational schema S consists of a finite set of relation names; for each relation is given the set of attributes that appear in it and one or more subsets of this set called 'keys'. (We will explain later why the word key appears in qoutes.) Given a relation R, we say that an FD: $X \rightarrow A$ is <u>embodied</u> in R if X is a given 'key' of R and A is any attribute of R. The set of FDs that are embodied in the schema S is the collection of all FDS which are embodied in relations of S. A schema S <u>represents</u> a set F of FDs if the closure of the set of FDs embodied by S is equal to $F^+$ (cf. section I.5.2).

Given a relation R and a set of FDs F, a subset of the attributes of R is a <u>superkey</u> of R if for any attribute A in R, the FD $K \rightarrow A$ is in $F^+$. A superkey K is a <u>key</u> if it does not contain any proper subset that is also a superkey.

If the relation R is part of a schema that represents the set of FDs F, then all given 'key's of R are superkeys by definition. But they are not necessarily keys; they may contain extraneous attributes. By the results of part I, in any relation generated by the synthesis algorithm all 'key's are actually keys. Furthemore, we note that the first step of the synthesis algorithm can be used as a procedure for eliminating extraneous attributes from superkeys and it can be applied to any relation. Therefore, we will assume that any 'key's given in a relation are actually keys, that is, that extraneous attributes have been eliminated.

For our treatment of the problems in this part we do not require that a schema be given. If a set of FDs is given then it is possible to ask about a single relation if it is in BCNF. Similiarly, it is possible to ask if it contains additional keys. To summarize, we assume that the following is given:

1. A set F of FDs.

2. A relation R with some 'key's such that

    a. the set of FDs embodied in R is a subset of $F^+$, and

    b. all 'key's given in R are keys.

(Because of 2b. we will, from now on, omit the quotes from the word key.) In some cases we will construct a schema from the given set of FDs. This is done only to show that the corresponding result holds even if it is known that the relation and its keys have been produced by the synthesis algorithm. No use is made of the other relations in the schema.

## III.2 <u>BOYCE-CODD NORMAL FORM</u>

### III.2.1 <u>Updates in BCNF Relations</u>

Third normal form was introduced to solve certain kinds of update anomalies and consistency difficulties among nonprime attributes in a relation (cf. section I.2.5). However, 3NF does not eliminate such problems among prime attributes. Boyce-Codd normal form was introduced as a strictly stronger formulation of 3NF that extends 3NF to cover prime attributes as well. Conceputally, every FD among attributes in a Boyce-Codd normal form relation is of the form "key functionally determines attribute." Formally, a relation R is in Boyce-Codd normal form if the following condition holds: If there is an attribute A in R and a set of attributes X in R with A not in X and X -> A, then every attribute in R is functionally dependent upon X.

It is easily seen that every BCNF relation is also in 3NF. For if X -> Y, Y -/> X, Y -> A were a transitive dependency in a BCNF relation, then Y -> A and Y -/> X would be a violation of BCNF. On the other hand, not every 3NF relation is in BCNF.

An example of a 3NF relation that is not in BCNF is POSTAL_DISTRICT(CITY, ADDRESS, POSTALCODE) where CITY,ADDRESS -> POSTALCODE and POSTALCODE -> CITY. The only two keys of POSTAL_DISTRICT are CITY,ADDRESS and POSTALCODE,ADDRESS. The relation POSTAL_DISTRICT is trivially in 3NF, since it has no nonprime attributes. Yet POSTAL_DISTRICT is not in BCNF, since POSTAL_CODE -> CITY but POSTAL_CODE -/> ADDRESS. Notice also that ADDRESS -/> CITY, since an address (e.g., 10 Elm Street) could appear in more than one city.

In non-BCNF relations, problems arise that are basically the same as those caused by transitive dependencies of non-prime attributes on keys. The reason is that the extensions of FDs in a non-BCNF relation cannot be independently updated. For example, consider the FDs CITY,ADDRESS -> POSTALCODE and POSTALCODE -> CITY in the POSTALDISTRICT relation. One cannot arbitrarily change the POSTALCODE for a particular CITY and ADDRESS, because by doing so one can violate the FD POSTALCODE -> CITY. This is essentially the consistency problem found in 3NF violations. Also, the insertion of the first CITY,ADDRESS combination for a particular POSTALCODE creates a new POSTALCODE -> CITY connection. Thus, insertion/deletion anomalies appear here as well.

One can look at BCNF as an attempt at making tuple updates completely independent. That is, since each tuple in a relation normally represents an object or relationship in the world (e.g., see [16]), one would expect to be able to update any one tuple in a relation without regard to any other in that relation. The above example shows that this is not always possible in a 3NF relation. However, as we will now explain, it is always possible in a BCNF schema.

Suppose we want to change the values of some attributes in one tuple of a relation. Other tuples may be influenced by this update only if the following two conditions are met. The first condition is that a given combination of values for these attributes may appear in different tuples in the relation. (This is equivalent to saying that the set of attributes whose values we want to change is not a key of the relation.) Clearly, only tuples in which these attributes have the same values as in the 'new' tuple can be influenced by the update. The second condition is that this set of attributes determines another attribute in the relation (because then the update may violate this dependency). Now, in a BCNF relation, no set of attributes can satisfy both conditions. A set is either a key or it does not determine any other attribute. Therefore, tuple updates in a BCNF relation are independent.

### III.2.2 Some Negative Results

In the first part of this report, we presented a fast algorithm for synthesizing a 3NF schema from a given set of FDs. In this section we will show why any similar approach to BCNF is very likely to fail.

The main goal, then, is to find a BCNF relational schema that represents a given set of FDs. However, this goal is impossible to fulfill, since there are sets of FDs that cannot be represented by any BCNF schema.

Lemma 6: There is a set of FDs that cannot be represented by any BCNF relational schema.

Proof Let $F = \{AB \rightarrow C, C \rightarrow A\}$ be a set of FDs. (These are exactly the FDs in POSTALDISTRICT, with A = CITY, B = ADDRESS, and C = POSTALCODE.) By a brute force examination of $F^+$, it can be shown that there are no two FDs f and g in $F^+$ that range over a proper subset of $\{A,B,C\}$ with $f \bullet g: AB \rightarrow C$. Thus, the only relation that can embody $AB \rightarrow C$ must be one that contains A, B, and C as attributes. But such a relation is not in BCNF, since $C \rightarrow A$ but $C \not\rightarrow B$. Hence, any schema that embodies F violates BCNF. □

It has been pointed out that a BCNF schema can be formed from which $AB \rightarrow C$ and $C \rightarrow A$ can be extracted [9]. However, by lemma 6 any such method of extracting these FDs from the schema must involve information in addition to the knowledge of embodied FDs (e.g., that a particular join yields $AB \rightarrow C$ as a result). That is, such a schema does not embody $AB \rightarrow C$ and $C \rightarrow A$, in our formal sense. Yet we know of no published relational system that allows such additional information to be represented in the data definition language. Only the keys are known. Hence, our formal definition of embodiment closely models what is actually feasible in present-day relational systems. This means that some FDs always lead to BCNF violations and require a special mechanism to solve the integrity problems induced by such violations.

The impact of lemma 6 is softened somewhat by the observation that sets of FDs which cannot be represented by BCNF schemas seem to be quite rare in modelling real world situations. What we would like to know is: In synthesizing a schema from a given set of FDs F, can we at least guarantee that if a BCNF schema that embodies a covering of F is possible, then our

Figure 12

Two Coverings, Only One of which Violates BCNF

| <u>FDs</u> | <u>Relations</u> |
|---|---|
| A -> B,C | R1($\underline{A}$,$\underline{B},\underline{C}$) |
| B,C -> A | |
| A,D -> E | R2($\underline{A},\underline{D}$,E) |
| E -> C | R3($\underline{E}$,C) |

(a)

| <u>FDs</u> | <u>Relations</u> |
|---|---|
| A ->B,C | S1($\underline{A}$,$\underline{B},\underline{C}$) |
| B,C -> A | |
| B,C,D -> E | S2($\underline{B},\underline{C},\underline{D}$,E) |
| E -> C | S3($\underline{E}$,C) |

(b)

In both cases the given sets of FDs are nonredundant. Also, they have the same closure. Yet, in the first case the synthesized schema is in BCNF, in the second case it is not.

synthesis algorithm will find it? That is, can be be sure to obtain a BCNF schema from algorithm 2 when such a schema is possible? Unfortunately, the answer is no. In figure 12, we present two nonredundant covering of a given set of FDs, where one covering results in a BCNF schema and the other does not. This shows that BCNF is not an invariant property of coverings; if step 2 of Algorithm 2 chooses the wrong covering, the result will violate BCNF.

It is conceivable that one could develop a method of finding a nonredundant covering that always chooses a covering that generates a BCNF schema, if such a covering exists. However, we currently know of no simple property that would guide a covering algorithm to the correct choice in polynomial time.

Given that the problem of synthesizing a BCNF schema seems to be quite difficult, perhaps we should simply allow Algorithm 2 to make mistakes and generate BCNF violations where they could have been avoided. Now, we are faced with the problem of examining each relation of the schema to check whether it is in BCNF.

Checking whether a relation is in BCNF is clearly decidable. The membership algorithm can be used to check every subset of attributes in the relation to see if it functionally determines some but not all attributes in the relation. This algorithm, though, is very slow, since it requires checking an exponential number of sets of attributes. That a faster algorithm is not very likely to be found follows from theorem 8, which shows the BCNF detection problem to be NP-complete.

<u>Theorem 8</u>: Given a set of attributes X, a set F of FDs over X, and a 3NF relation R over a subset of X, the question "does R violate BCNF" is NP-complete. The problem is NP-complete even when it is known that R is one of the relations produced by the synthesis algorithm from the set F.

<u>Proof</u> The problem is NP computable by nondeterministically choosing a subset of the attributes of R and verifying that this subset functionally determines some but not all attributes of R (i.e., verifying that R violated BCNF). To show the problem is NP-difficult, we reduce the hitting set problem [13] to the BCNF violation problem.

The hitting set problem is formulated as follows: We are given a family {Vi} i=1,...,n of subsets of T = {t1,...,tr}. We have to decide if there exists a set W c T such that for each 1≤i≤n the intersection of W with Vi contains exactly one element. The problem was proved to be NP-complete in [13].

We now show how to construct, in polynomial time, for a given instance of the hitting set problem a corresponding instance of the BCNF violation problem, such that the two instances have either both positive solutions or both negative solutions.

We construct the following set F of FDs:

1. For each i, for each pair u1, u2 of elements in Ui and for each $t \in T$, the set F contains the FD u1,u2 -> t.

2. For each i, let xi be a new object. The set F contains the FD x1,...,xn -> t1.

3. For each $u \in Ui$, for each i, F contains the FD u -> x.

Applying the synthesis algorithm to the set F, we obtain the following relations. From (1), we obtain a relation R1 that contains exactly the elements of T and in which the pairs of elements of any Ui are the synthesized keys. From (2), we obtain a relation R2 that contains x1,...,xn and t1 and its synthesized key is x1,...,xn. From (3) we obtain relatons R3,...,Rr+2, such that in Rj+2 the key is tj and it contains all xi such that $tj \in Ui$.

We prove now that R1 has a BCNF violation if and only if the hitting set problem has a positive solution. First, suppose W is a subset of T that intersects each Ui exactly once. Then W -> x1,...,xn follows from the FDs in (3), so applying the FD in (2) we obtain W -> t1 in the closure. However, W does not contain a key of R1, since the only FDs that have t2,...,tn on the right hand side are those of (1) and W does not contain two elements of any Ui. Conversely, suppose R1 has a BCNF violation. Let $Y \subset T$ be a subset of attributes of R that derives some $tk \in T$ but not $tj \in T$. Clearly, Y cannot contain any pair of elements of any Ui. The only way then to derive any element of T is by using x1,...,xn -> t1, which means that Y contains one element of at least each Ui and is a hitting set.

Finally, we observe that the number of pairs of elements in T is $r^2$, so the number of FDs we construct in (1) is at most $r^2$. It follows easily that the instance of the BCNF violation problem is constructed in time proportional to a polynomial function of the size of the instance of the hitting set problem. The theorem follows. □

To summarize the above results, we have:

1. Not every set of FDs can be represented by a BCNF schema,

2. Algorithm 2 does not necessarily synthesize a BCNF schema, even when such a schema is possible, and

3. the problem of determining whether a relation violates BCNF is NP-complete.

Together, these results strongly suggest that FDs are too strong a model to obtain BCNF schemas algorithmically. Perhaps a better approach is to develop a weaker model of real world relationships, a model whose additional structure (and weaker modelling power) makes the detection of BCNF violations an easier problem. For any model that is strong enough to model FDs must manifest the above problems.

## III.3 KEY FINDING

In this section we treat the problem of the possible existence of unknown keys. If the keys synthesized by Algorithm 2 (i.e., the "synthesized keys") for a particular relation always included all of the keys of the relation, then there would be no problem. However, a relation can have keys that are not synthesized by Algorithm 2. For example, given the set of FDs {AB->C, C->B}, the 3NF schema constructed by the synthesis algorithm contains the relations R1(A,B,C) and R2(C,B). Clearly, AC is an additional key of R1, although it was not synthesized. Given that these additional keys exist, the question we would like to examine is: How difficult is it to find these keys?

One approach to finding the keys of a relation is to check all subsets of the attributes in the relation starting, say, with subsets of one element, then subsets of two elements, etc. Since the number of such subsets grows very quickly with the size of the relation, it would be helpful to discover a condition that will tell us that no more subsets have to be checked, since all of the keys have already been found. One such condition might be that if all known keys have cardinality less than some integer n, and there are no keys of cardinality n, then there are no more keys to check. This condition would allow us to stop building up subsets when all the subsets of a particular cardinality turn out to yield no new keys. However, this condition fails on the example in figure 13. In this example, the attributes X1,X2,X3,X4 together constitute a key of R1, yet this key is not synthesized by the synthesis algorithm. Even though there are no additional keys of cardinality less than four (and there are no keys at all of cardinality 3), this additional key exists. It is also easy to generalize the example to an arbitrarily large cardinality gap (i.e., no keys of cardinality greater than two and less than n, for arbitrary n, yet one key of cardinality n+1). This cardinality condition, and others like it, fail for a fundamental reason.

Let the <u>additional key problem</u> be defined as follows: Given a relation R with a set of keys and a set of FDs (satisfying the conditions 1, 2a, 2b of section III.1.2), does R contain an additional key?

<u>Theorem 9</u>: The additional key problem is NP-complete.

Figure 13

A Cardinality Gap for Keys

K -> X1, X2, X3, X4

L1, L2 -> K

X1 -> M1, M5

X2 -> M2, M6

X3 -> M3, M7

X4 -> M4, M8

M1,M2,M3,M4 -> L1

M5,M6,M7,M8 -> L2

(i) a given set of FDs

R1(<u>K</u>, <u>L1,L2</u>, X1, X2, X3, X4)

R2(<u>X1</u>, M1, M5)

R3(<u>X2</u>, M2, M6)

R4(<u>X3</u>, M3, M7)

R5(<u>X4</u>, M4, M8)

R6(<u>M1,M2,M3,M4</u>, L1)

R7(<u>M5,M6,M7,M8</u>, L2)

(ii) The relations synthesized from the above FDs.

In the relation R1, X1,X2,X3,X4 is a non-synthesized key, even though there are no keys of cardinality three.

__Proof__ The problem is obviously NP-computable. Given R, we nondeterministically choose a subset of its attributes and check whether it is a key (using, say, the membership algorithm). To show that the problem is NP-hard, we again use a reduction of the hitting set problem. The construction is almost the same as that used for the BCNF violation (theorem 8) except that in (2) we construct x1,...,xn -> t for all t ∈ T. Now, if W is a solution to the hitting set problem, then W -> x1,...,xn -> T, so W is an additional key. Conversely, if W is an additional key, then it can not contain any pair of elements of any Ui. It must therefore use x1,...,xn -> T to derive all elements of T, so it is a solution to the hitting set problem. □

Lucchesi and Osborne [14] have treated some related problems in a somewhat restricted model. In their model, one is given a set of attributes and a set of FDs on them; they defined a key as a minimal subset that derives all other attributes. This essentially means that they treat all attributes of the entire schema as being collected into a single relation. In our approach attributes are collected in several relations. A key of a relation derives all attributes in that relation. It is generally accepted that relational schemas consist of many relations, as evidenced by all of the relational systems built to date. The division of the set of attributes into relations is necessary for several reasons: normalization considerations, ease of use, size of relations, etc. We will now consider the difference in approaches by comparing results in the two models.

First, Lucchesi and Osborne exhibited an algorithm that lists all keys in time that is polynomial in the number of attributes and the number of FDs, and linear in the number of keys. The algorithm does not seem to generalize to our model. Furthermore, theorem 9 strongly suggests that such a generalized algorithm does not exist. For suppose we had an algorithm that produces all of the keys of a relation, using our multiple-relation model. If we modified the algorithm so that it stops after the first unknown key is generated, we would obtain an algorithm that solves the additional key problem. If the original algorithm worked in time that is polynomial in the number of attributes, number of FDs and the number of keys generated, then one would expect the new algorithm to have the same time bound (Though it is not necessarily so). This would make it a polynomial time solution to the additional key problem, which is known to be NP-complete.

Lucchesi and Osborne [14] proved two other problems to be NP-complete (in the one-relation model). These are:

1. The __prime attribute problem__: Given an attribute A, decide whether it belongs to any key.

2. The __key of cardinality m problem__: Given an integer m>1, decide if there exists a key of cardinality less than m.

Since their one-relation model is a special case of the multiple-relation model, these problems are also NP-complete in the latter model. However, it should be noted that these problems are NP-complete even if we restrict ourselves only to multiple-relation schemas synthesized by Algorithm 2. This is not a direct consequence of their results. To see this, let us consider the prime attribute problem for synthesized schemas.

Suppose we are given a set of FDs G over the attributes A1,...,Am. Let R(A1,...,An) where $n<m$ be one of the relations synthesized from G by the synthesis algorithm. We want to know if A1 is prime in R. We map this problem to a similiar problem for the one-relational model. To do this, we add a new attribute D and we add the FD D -> A1,...,Am and the FD A1,...,An -> D to G. If we apply the synthesis algorithm to the new set of FDs, there will be one additional relation containing the attributes D,A1,...,Am. The keys for this relation will be D and any key of R. Thus, A1 is prime in R iff it is prime in this relation. But this relation contains all the attributes in the schema so, by the Lucchesi and Osborne result, the problem is NP-complete. A similiar reduction can be used for the key of cardinality m problem.

The results in this section strongly suggest that key finding is an inherently difficult problem. From theorem 9 it follows that if NP ¬= P then there is no algorithm that lists all keys in time polynomial in the size of the relation and the set of FDs. It is true that even if NP ¬= P none of the results implies that an algorithm that lists all keys of a relation in time polynomial in the number of keys does not exist. It the difficulty of the additional key problem lies in the cases where the number of additional keys is exponential in the size of the relation. However, we conjecture that such an algorithm does not exist.

## CONCLUSION

Our goal in this paper was to analyze the possibility of an efficient algorithmic treatment of problems related to normalization in relational schemas. The main achievement is the presentation of an algorithm that solves the synthesis problem. To our knowledge, this is the first correct algorithm for this problem. Our efforts to find an efficient implementation of the algorithm led us to an improvement of previous algorithms for the membership problem of functional dependencies. We have presented here a linear time membership algorithm and using it we have presented a quadratic time implementation of the synthesis algorithm. We doubt if these time bounds can be significantly improved.

Our results about the Boyce-Codd normal form and about the key finding problems have been all negative. We have shown that the problems of whether a schema is in Boyce-Codd normal form and of whether a relation contains additional keys are NP-complete. On the basis of these and other results we argued (though we could not prove) that an efficient algorithm that produces Boyce-Codd normal form schemas and an efficient algorithm that lists all keys of a relation do not exist.

In view of the synthesis algorithm, the concept of functional dependency has proven to be a useful tool for the construction of relational schemas. Recently, generalizations of this concept have been suggested. We believe that functional dependencies (and their generalizations) may prove to be useful in the treatment of other problems of relational data base systems.

References

[1] Armstrong, W. W., "Dependency Structures of Data Base Relationships", _Proc. of IFIP '74_, North-Holland, 1974, pp. 580-583.

[2] Bernstein, P. A., _Normalization and Functional Dependencies in the Relational Data Base Model_, Tech. Rep. CSRG-60 (Ph.D. Diss.), Computer Systems Research Group, Dept. of Computer Science, Univ. of Toronto, Nov. 1975.

[3] Bernstein, P. A., J. R. Swenson, and D. C. Tsichritzis, "A Unified Approach to Functional Dependencies and Relations", _Proc. of 1975 ACM SIGMOD Conference_, San Jose, May, pp. 237 - 245.

[4] Bernstein, P. A., "A Comment on 'Segment Synthesis in Logical Data Base Design,'" _IBM J. of Res. and Dev._, Vol. 21 # 7 (July 1976).

[5] Casey, R. G., C. Delobel, and P. A. Bernstein, "A Correction to 'Decomposition of a Data Base and the Theory of Boolean Switching Functions'", _IBM J. of Res. and Dev._ (to appear).

[6] Codd, E. F. "A Relational Model for Large Shared Data Bases", _CACM_, Vol. 13, #6 (June 1970), pp. 377 - 387.

[7] Codd, E.F., "Further Normalization of the Data Base Relational Model", in _Data Base Systems_ (Courant Computer Science Symposium 6, R. Rustin ed.), Prentice-Hall, 1972, pp. 33-64.

[8] Codd, E. F., "Recent Investigations in Relational Data Base Systems", _Proc. f IFIP '74_, North-Holland, 1974, pp. 1017 - 1021.

[9] Date, C. J., _An Introduction to Database Systems_, Addison-Wesley, Reading, Mass., 1975.

[10] Delobel, C. and R. G. Casey, "Decomposition of a Data Base and the Theory of Boolean Switching Functions", _IBM J. of Res. and Dev._, Vol. 17, #5 (Sept. 1972), pp. 374-386.

[11] Fagin R., "Dependency in a Relational Database and Propositional Logic", IBM Research Report, San Jose, California, 1976.

[12] Forsyth, J. and R. Fadous, "Finding Candidate Keys for Relational Data Bases", _Proc. f ACM SIGMOD Conf._, San Jose, May 1975, pp. 203 - 210.

[13] Karp, R. M., "Reducibility among Combinatorial Problems", Technical Report 3, Dept. of Computer Science, University of California at Berkeley, April 1972.

[14] Lucchesi, C. L. and S. L. Osborne, "Candidate keys for relations", unpublished paper, Univ. of Waterloo, 1976.

[15] Rissanen, J. and C. Delobel, "Decomposition of Files, A Basis for Data Storage and Retrieval", IBM Research Report RJ 1220 (May), San Jose, California, 1973.

[16] Schmid, H.A. and J. R. Swenson, "On the Semantics of the Relational Data Model," Proc. of ACM SIGMOD Conf., San Jose, May 1975, pp. 211 - 223.

[17] Wang, C. P. and H. H. Wedekind, "Segment Synthesis in Logical Data Base Design", IBM J. of Res. and Dev., Vol. 19, # 1 (Jan 1975), pp. 71-77.

UNIVERSITY OF TORONTO

COMPUTER SYSTEMS RESEARCH GROUP

<u>BIBLIOGRAPHY OF CSRG TECHNICAL REPORTS</u>+

* CSRG-1 EMPIRICAL COMPARISON OF LR(k) AND PRECEDENCE PARSERS
J.J. Horning and W.R. Lalonde, September 1970
[ACM SIGPLAN Notices, November 1970]

CSRG-2 AN EFFICIENT LALR PARSER GENERATOR
W.R. Lalonde, February 1971 [M.A.Sc. Thesis, EE 1971]

* CSRG-3 A PROCESSOR GENERATOR SYSTEM
J.D. Gorrie, February 1971 [M.A.Sc. Thesis, EE 1971]

* CSRG-4 DYLAN USER'S MANUAL
P.E. Bonzon, March 1971

CSRG-5 DIAL - A PROGRAMMING SYSTEM FOR INTERACTIVE ALGEBRAIC MANIPULATION
Alan C.M. Brown and J.J. Horning, March 1971

* CSRG-6 ON DEADLOCK IN COMPUTER SYSTEMS
Richard C. Holt, April 1971
[Ph.D. Thesis, Dept. of Computer Science, Cornell University, 1971]

CSRG-7 THE STAR-RING SYSTEM OF LOOSELY COUPLED DIGITAL DEVICES
John Neill Thomas Potvin, August 1971
[M.A.Sc. Thesis, EE 1971]

* CSRG-8 FILE ORGANIZATION AND STRUCTURE
G.M. Stacey, August 1971

CSRG-9 DESIGN STUDY FOR A TWO-DIMENSIONAL COMPUTER-ASSISTED ANIMATION SYSTEM
Kenneth B. Evans, January 1972 [M.Sc. Thesis, DCS, 1972]

* CSRG-10 HOW A PROGRAMMING LANGUAGE IS USED
William Gregg Alexander, February 1972
[M.Sc. Thesis, DCS 1971; Computer, v.8, n.11, November 1975]

CSRG-11 PROJECT SUE STATUS REPORT
J.W. Atwood (ed.), April 1972

* CSRG-12 THREE DIMENSIONAL DATA DISPLAY WITH HIDDEN LINE REMOVAL
Rupert Bramall, April 1972 [M.Sc. Thesis, DCS, 1971]

CSRG-13 A SYNTAX DIRECTED ERROR RECOVERY METHOD
Lewis R. James, May 1972 [M.Sc. Thesis, DCS, 1972]

---

\+ Abbreviations:
DCS - Department of Computer Science, University of Toronto
EE - Department of Electrical Engineering, University of Toronto

* - Out of print

CSRG-14 THE USE OF SERVICE TIME DISTRIBUTIONS IN SCHEDULING
Kenneth C. Sevcik, May 1972
[Ph.D. Thesis, Committee on Information Sciences, University of Chicago, 1971; JACM, January 1974]

CSRG-15 PROCESS STRUCTURING
J.J. Horning and B. Randell, June 1972
[ACM Computing Surveys, March 1973]

CSRG-16 OPTIMAL PROCESSOR SCHEDULING WHEN SERVICE TIMES ARE HYPEREXPONENTIALLY DISTRIBUTED AND PREEMTION OVERHEAD IS NOT NEGLIGIBLE
Kenneth C. Sevcik, June 1972
[Proceedings of the Symposium on Computer-Communication, Networks and Teletraffic, Polytechnic Institute of Brooklyn, 1972]

CSRG-17 PROGRAMMING LANGUAGE TRANSLATION TECHNIQUES
W.M. McKeeman, July 1972

CSRG-18 A COMPARATIVE ANALYSIS OF SEVERAL DISK SCHEDULING ALGORITHMS
C.J.M. Turnbull, September 1972

CSRG-19 PROJECT SUE AS A LEARNING EXPERIENCE
K.C. Sevcik et al, September 1972
[Proceedings AFIPS Fall Joint Computer Conference. v. 41, December 1972]

* CSRG-20 A STUDY OF LANGUAGE DIRECTED COMPUTER DESIGN
David B. Wortman, December 1972
[Ph.D. Thesis, Computer Science Department, Stanford University, 1972]

CSRG-21 AN APL TERMINAL APPROACH TO COMPUTER MAPPING
R. Kvaternik, December 1972 [M.Sc. Thesis, DCS, 1972]

* CSRG-22 AN IMPLEMENTATION LANGUAGE FOR MINICOMPUTERS
G.G. Kalmar, January 1973 [M.Sc. Thesis, DCS, 1972]

CSRG-23 COMPILER STRUCTURE
W.M. McKeeman, January 1973
[Proceedings of the USA-Japan Computer Conference, 1972]

* CSRG-24 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
J.D. Gannon (ed.), March 1973

CSRG-25 THE INVESTIGATION OF SERVICE TIME DISTRIBUTIONS
Eleanor A. Lester, April 1973 [M.Sc. Thesis, DCS, 1973]

* CSRG-26 PSYCHOLOGICAL COMPLEXITY OF COMPUTER PROGRAMS: AN INITIAL EXPERIMENT
Larry Weissman, August 1973

* CSRG-27 STRUCTURED SUBSETS OF THE PL/I LANGUAGE
Richard C. Holt and David B. Wortman, October 1973

* CSRG-28 ON THE REDUCED MATRIX REPRESENTATION OF LR(k) PARSER TABLES
  Marc Louis Joliat, October 1973 [Ph.D. Thesis, EE 1973]

* CSRG-29 A STUDENT PROJECT FOR AN OPERATING SYSTEMS COURSE
  B. Czarnik and D. Tsichritzis (eds.), November 1973

* CSRG-30 A PSEUDO-MACHINE FOR CODE GENERATION
  Henry John Pasko, December 1973 [M.Sc. Thesis, DCS 1973]

* CSRG-31 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
  J.D. Gannon (ed.), Second Edition, March 1974

CSRG-32 SCHEDULING MULTIPLE RESOURCE COMPUTER SYSTEMS
E.D. Lazowska, May 1974 [M.Sc. Thesis, DCS, 1974]

* CSRG-33 AN EDUCATIONAL DATA BASE MANAGEMENT SYSTEM
  F. Lochovsky and D. Tsichritzis, May 1974 [INFOR, to appear]

* CSRG-34 ALLOCATING STORAGE IN HIERARCHICAL DATA BASES
  P. Bernstein and D. Tsichritzis, May 1974 [Information Systems Journal, v.1, pp.133-140]

* CSRG-35 ON IMPLEMENTATION OF RELATIONS
  D. Tsichritzis, May 1974

* CSRG-36 SIX PL/I COMPILERS
  D.B. Wortman, P.J. Khaiat, and D.M. Lasker, August 1974 [Software Practice and Experience, v.6, n.3, July-Sept. 1976]

* CSRG-37 A METHODOLOGY FOR STUDYING THE PSYCHOLOGICAL COMPLEXITY OF COMPUTER PROGRAMS
  Laurence M. Weissman, August 1974 [Ph.D. Thesis, DCS, 1974]

* CSRG-38 AN INVESTIGATION OF A NEW METHOD OF CONSTRUCTING SOFTWARE
  David M. Lasker, September 1974 [M.Sc. Thesis, DCS, 1974]

CSRG-39 AN ALGEBRAIC MODEL FOR STRING PATTERNS
Glenn F. Stewart, September 1974 [M.Sc. Thesis, DCS, 1974]

CSRG-40 EDUCATIONAL DATA BASE SYSTEM USER'S MANUAL
J. Klebanoff, F. Lochovsky, A. Rozitis, and D. Tsichritzis, September 1974

* CSRG-41 NOTES FROM A WORKSHOP ON THE ATTAINMENT OF RELIABLE SOFTWARE
  David B. Wortman (ed.), September 1974

* CSRG-42 THE PROJECT SUE SYSTEM LANGUAGE REFERENCE MANUAL
  B.L. Clark and F.J.B. Ham, September 1974

CSRG-43 A DATA BASE PROCESSOR
E.A. Ozkarahan, S.A. Schuster and K.C. Smith, November 1974 [Proceedings National Computer Conference 1975, v.44, pp.379-388]

* CSRG-44 MATCHING PROGRAM AND DATA REPRESENTATION TO A COMPUTING ENVIRONMENT
Eric C.R. Hehner, November 1974 [Ph.D. Thesis, DCS, 1974]

* CSRG-45 THREE APPROACHES TO RELIABLE SOFTWARE; LANGUAGE DESIGN, DYADIC SPECIFICATION, COMPLEMENTARY SEMANTICS
J.E. Donahue, J.D. Gannon, J.V. Guttag and J.J. Horning, December 1974

CSRG-46 THE SYNTHESIS OF OPTIMAL DECISION TREES FROM DECISION TABLES
Helmut Schumacher, December 1974 [M.Sc. Thesis, DCS, 1974]

CSRG-47 LANGUAGE DESIGN TO ENHANCE PROGRAMMING RELIABILITY
John D. Gannon, January 1975 [Ph.D. Thesis, DCS, 1975]

* CSRG-48 DETERMINISTIC LEFT TO RIGHT PARSING
Christopher J.M. Turnbull, January 1975 [Ph.D. Thesis, EE, 1974]

CSRG-49 A NETWORK FRAMEWORK FOR RELATIONAL IMPLEMENTATION
D. Tsichritzis, February 1975 [in Data Base Description, Dongue and Nijssen (eds.), North Holland Publishing Co.]

* CSRG-50 A UNIFIED APPROACH TO FUNCTIONAL DEPENDENCIES AND RELATIONS
P.A. Bernstein, J.R. Swenson and D.C. Tsichritzis February 1975 [Proceedings of the ACM SIGMOD Conference, 1975]

* CSRG-51 ZETA: A PROTOTYPE RELATIONAL DATA BASE MANAGEMENT SYSTEM
M. Brodie (ed). February 1975 [Proceedings Pacific ACM Conference, 1975]

* CSRG-52 AUTOMATIC GENERATION OF SYNTAX-REPAIRING AND PARAGRAPHING PARSERS-
David T. Barnard, March 1975 [M.Sc. Thesis, DCS, 1975]

* CSRG-53 QUERY EXECUTION AND INDEX SELECTION FOR RELATIONAL DATA BASES
J.H. Gilles Farley and Stewart A. Schuster, March 1975

CSRG-54 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
J.V. Guttag (ed.), Third Edition, April 1975

CSRG-55 STRUCTURED SUBSETS OF THE PL/1 LANGUAGE
Richard C. Holt and David B. Wortman, May 1975

CSRG-56 FEATURES OF A CONCEPTUAL SCHEMA
D. Tsichritzis, June 1975 [Proceedings Very Large Data Base Conference, 1975]

* CSRG-57 MERLIN: TOWARDS AN IDEAL PROGRAMMING LANGUAGE
Eric C.R. Hehner, July 1975

CSRG-58 ON THE SEMANTICS OF THE RELATIONAL DATA MODEL
Hans Albrecht Schmid and J. Richard Swenson, July 1975 [Proceedings of the ACM SIGMOD Conference, 1975]

CSRG-59 THE SPECIFICATION AND APPLICATION TO PROGRAMMING OF ABSTRACT DATA TYPES
John V. Guttag, September 1975 [Ph.D. Thesis, DCS, 1975]

CSRG-60 NORMALIZATION AND FUNCTIONAL DEPENDENCIES IN THE RELATIONAL DATA BASE MODEL
Phillip Alan Bernstein, October 1975 [Ph.D. Thesis, DCS, 1975]

CSRG-61 LSL: A LINK AND SELECTION LANGUAGE
D. Tsichritzis, November 1975 [Proceedings ACM SIGMOD Conference, 1976]

CSRG-62 COMPLEMENTARY DEFINITIONS OF PROGRAMMING LANGUAGE SEMANTICS
James E. Donahue, November 1975 [Ph.D. Thesis, DCS, 1975]

CSRG-63 AN EXPERIMENTAL EVALUATION OF CHESS PLAYING HEURISTICS
Lazlo Sugar, December 1975 [M.Sc. Thesis, DCS, 1975]

CSRG-64 A VIRTUAL MEMORY SYSTEM FOR A RELATIONAL ASSOCIATIVE PROCESSOR
S.A. Schuster, E.A. Ozkarahan, and K.C. Smith, February 1976 [Proceedings National Computer Conference 1976, v.45, pp.855-862]

CSRG-65 PERFORMANCE EVALUATION OF A RELATIONAL ASSOCIATIVE PROCESSOR
E.A. Ozkarahan, S.A. Schuster, and K.C. Sevcik, February 1976 [ACM Transactions on Database Systems, v.1, n:4, December 1976]

CSRG-66 EDITING COMPUTER ANIMATED FILM
Michael D. Tilson, February 1976 [M.Sc. Thesis, DCS, 1975]

CSRG-67 A DIAGRAMMATIC APPROACH TO PROGRAMMING LANGUAGE SEMANTICS
James R. Cordy, March 1976 [M.Sc. Thesis, DCS, 1976]

CSRG-68 A SYNTHETIC ENGLISH QUERY LANGUAGE FOR A RELATIONAL ASSOCIATIVE PROCESSOR
L.Kerschberg, E.A. Ozkarahan, and J.E.S. Pacheco April 1976

CSRG-69 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
D. Barnard and D. Thompson (Eds.), Fourth Edition, May 1976

CSRG-70 A TAXONOMY OF DATA MODELS
L. Kerschberg, A. Klug, and D. Tsichritzis, May 1976
[Proceedings Very Large Data Base Conference, 1976]

CSRG-71 OPTIMIZATION FEATURES FOR THE ARCHITECTURE OF A DATA BASE MACHINE
E.A. Ozkarahan and K.C. Sevcik, May 1976

CSRG-72 THE RELATIONAL DATA BASE SYSTEM OMEGA - PROGRESS REPORT
H.A. Schmid (ed.), P.A. Bernstein (ed.), B. Arlow, R. Baker and S. Pozgaj, July 1976

CSRG-73 AN ALGORITHMIC APPROACH TO NORMALIZATION OF RELATIONAL DATA BASE SCHEMAS
P.A. Bernstein and C. Beeri, September 1976