Computational Problems Related to the Design of Normal Form Relational Schemas

CATRIEL BEERI
The Hebrew University

and

PHILIP A. BERNSTEIN
Harvard University

Problems related to functional dependencies and the algorithmic design of relational schemas are examined. Specifically, the following results are presented: (1) a tree model of derivations of functional dependencies from other functional dependencies; (2) a linear-time algorithm to test if a functional dependency is in the closure of a set of functional dependencies; (3) a quadratic-time implementation of Bernstein's third normal form schema synthesis algorithm.

Furthermore, it is shown that most interesting algorithmic questions about Boyce-Codd normal form and keys are $\mathcal{NP}$-complete and are therefore probably not amenable to fast algorithmic solutions.

Key Words and Phrases: database design, derivation tree, functional dependency, membership algorithm, $\mathcal{NP}$-complete, relational database, third normal form
CR Categories: 4.33, 5.25

1. INTRODUCTION

The relational model has attracted widespread interest among database practitioners and researchers for its conceptual simplicity and for its mathematical elegance. One database problem that is especially easy to formulate in terms of relations is that of logical schema design: given a description of an application, find a "good" schema that describes the structure of the corresponding database. Currently, this logical design problem is largely in the hands of database administrators (DBA's) whose tools for performing this task are quite meager relative to the complexity of the problem.

In an early paper on the relational model, Codd attempted to put some science into the logical database design process by defining certain normal forms for a

This work was supported in part by the National Research Council of Canada, by a fellowship from the Lady Davis Fellowship Fund of Jerusalem, and by the National Science Foundation under Grant MCS-77-05314.

Authors' addresses: C. Beeri, Department of Computer Science, The Hebrew University, Jerusalem, Israel; P. A. Bernstein, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138.

relational schema [12]; these normal forms were a well-defined goal towards which a DBA might strive. Codd's normal forms were based on the concept of "functional dependency," which is essentially a functional relationship among database attributes. This concept of "functional dependency" was quickly formalized by researchers and it has proven to be of importance both for the logical design of relational database schemas and for the investigation of properties of relational schemas.

The problem of schema design may be loosely stated as follows: transform a DBA-supplied set of functional dependencies that describe an application, possibly augmented by an initial guess of some relations, into a relational schema satisfying certain requirements. Some basic requirements are that the schema satisfies Codd's normal forms and that the supplied functional dependencies are appropriately represented in it. Several algorithms that synthesize a relational schema using this functional dependency approach have appeared [8, 10, 15, 19, 27].

Despite the simplicity of the concept of functional dependency, applying it to schema design requires a fairly complex mathematical treatment. For example, early synthesis algorithms were found to have errors and to run quite slowly (i.e. to have exponential worst-case time bounds) [7, 15, 16, 27]. Recently, a correct solution to the synthesis problem was presented [8]; however, detailed algorithms to manipulate functional dependencies were not included. It is the purpose of this paper to present the basic algorithms for dealing with functional dependencies and to examine the computational complexity of problems related to the algorithmic design of relational schemas.

Armstrong [2] has presented a set of axioms (i.e. inference rules) for functional dependencies. Every problem dealing with functional dependencies requires a manipulation of functional dependencies according to these axioms. Thus we begin our investigation with a look at functional dependency derivations. We provide a graph-theoretic formalism—a tree model—through which derivations of functional dependencies can be better understood. Then we present a fast (linear-time) procedure to solve the functional dependency membership problem. This leads to efficient solutions of various other problems, such as the removal of redundancy from a given set of dependencies. The procedure is also sufficient to efficiently implement all previously suggested schema design algorithms based on functional dependencies. Finally, we show that two rather natural extensions to the schema design problem are computationally infeasible (i.e. $\mathcal{NP}$-hard): the problem of determining whether a third normal form schema is in Boyce-Codd normal form and the problem of finding keys. These results indicate bounds on certain directions for future research.

While this paper is in some ways a sequel and a complement to [8], we do not assume a familiarity with earlier work on functional dependencies. A pedagogic introduction to the problems discussed here can be found in Chapter 9 of [14].

2. THE RELATIONAL MODEL

2.1 Relations and Relation Schemes

In Codd's relational database model, relations over a set of attributes are used to describe connectons among data items [11]. A *relation* is a table in which each column corresponds to a distinct *attribute* and each row to a distinct *entity*. For each attribute there is a set of possible values called the *domain* of that attribute. Different attributes may share the same domain. An ⟨entity, attribute⟩ entry in a relation is a value associated with the entity, chosen from the domain of the attribute. An entity is described by a tuple of such entries, one for each attribute in the relation.

The set of tuples that comprise a relation normally changes over time as entities are inserted, deleted, or modified. Thus a relation is a time-varying set of tuples. However, the structure of a relation—its name and attributes—remains quite static, by comparison. The notation for describing the structure of a relation is $\mathbf{R}(A_1, \ldots, A_n)$ where $\mathbf{R}$ is the name of the relation and $\{A_1, \ldots, A_n\}$ is the set of attributes appearing in it. It is the structure of relations that will be the main topic of concern in this paper.

The word relation is used in the literature both to denote the structure of the relation (its *intention*) and to denote a set of tuples having the appropriate structure (an *extension*). In this paper we will use the term *relation scheme* to refer to an intention, that is, to refer to a structural description of a relation. (A precise definition is given in Section 2.4 below.) The word *relation* will be used to refer to an extension, that is, to a set of tuples.

A database, $\mathbf{D}$, normally consists of many relations each described by its own relation scheme. For dealing with the relationships that exist between attributes that may well be in different relation schemes, it is convenient to think of the database as a single relation described by a single relation scheme. The attributes of this "universal" relation scheme are all those attributes that are of interest for the application at hand.

Let $\{A_1, \ldots, A_m\}$ be this set of attributes. The database is described by the relation scheme $\mathbf{D}(A_1, \ldots, A_m)$. An extension of $\mathbf{D}$ is a subset of the Cartesian product $\mathrm{DOM}(A_1) \times \ldots \times \mathrm{DOM}(A_m)$, where $\mathrm{DOM}(A_i)$ denotes the domain of values of the attribute A_i. At any point of time, the information content of the database is such an extension. Obviously, however, an extension of $\mathbf{D}$ is never actually constructed. If the relation schemes are $\mathbf{R}_1, \ldots, \mathbf{R}_N$, then it is the projections of the extensions on the attribute sets of these relation schemes that are actually stored in the physical database. (The *projection* of a relation on a subset of its attributes is the relation that results from excising columns corresponding to attributes not in the subset and then eliminating duplicate tuples [11].) This view of the database as a universal relation represented by its projections that are the database relations assigns a single interpretation to each attribute in the database. That is, an attribute A that appears in two relation schemes $\mathbf{R}_i$ and $\mathbf{R}_j$ refers to the same column of $\mathbf{D}$. This interpretation of the relational model has been used by many authors (e.g. [2, 15, 17, 25]) and implies, in particular, the "uniqueness assumption" discussed in [3, 8] and in the next section.

2.2 Functional Dependencies

An important type of relationship between attributes is the functional dependency. Let A and B be attributes in database $\mathbf{D}$. We say that B is *functionally dependent* on A (in $\mathbf{D}$) if, at every point of time, for a given value of $a \in \mathrm{DOM}(A)$ there corresponds at most one value of $b \in \mathrm{DOM}(B)$. That is, in every extension of $\mathbf{D}$, if two tuples have the same value for attribute A, then they have the same value for attribute B. If B is functionally dependent on A we say that A *functionally determines* B. For notational convenience, we generally leave out the "DOM"; we use the notation $f: A \rightarrow B$ to denote the fact that B is functionally dependent on A. f is called a *functional dependency* (FD).

The above definitions are generalized in the obvious way for functional dependencies over sets of attributes. If $X = \{A_1, \ldots, A_n\}$ and $Y = \{B_1, \ldots, B_p\}$ are sets of attributes in **D**, then $f: X \rightarrow Y$ means $f: \mathrm{DOM}(A_1) \times \ldots \times \mathrm{DOM}(A_n) \rightarrow \mathrm{DOM}(B_1) \times \ldots \times \mathrm{DOM}(B_p)$. We will normally leave off the set notation in FD's and write simply $f: A_1 \ldots A_n \rightarrow B_1 \ldots B_p$.

By using **D** as an underlying domain of definition for FD's, we have guaranteed that for any two sets of attributes X and Y there can be at most one FD from X to Y. (This property is the so-called "uniqueness assumption" in [8].) This follows from the definition of FD and the fact that attributes are distinct in **D**. A lengthy discussion of database design problems that result from the uniqueness assumption appears in [4] and [8] and is beyond the scope of this paper.

Functional dependencies are integrity constraints that a DBA specifies to delimit the possible relations that can exist in the database. Since FD's are constraints, the database system must prevent any update that will cause some FD to be violated. To do this, the system must have an internal representation of the FD's. The construction of such internal representations as well as the treatment of other problems related to the construction of relational schemas, all involve certain computational problems, e.g., to determine if the DBA specification of the FD's is redundant and, if so, to remove the redundancy. All these problems stem from the fact that FD's may imply additional FD's.

Given a set of FD's in a database, it can be deduced that certain other FD's also exist in the database. Armstrong [2] has presented a set of inference rules for FD's and proved this set to be complete for the family of FD's. (For a detailed treatment of the notion of completeness for dependencies in database relations, see [5].) There are several equivalent sets of rules and we present one of them here. In the rules, X, Y, Z, and W are arbitrary sets of attributes.

Rule A1: (Reflexivity). If $Y \subseteq X$ then $X \rightarrow Y$.

Rule A2: (Augmentation). If $X \rightarrow Y$ and $Z \subseteq W$ then $XW \rightarrow YZ$.

Rule A3: (Pseudotransitivity). If $X \rightarrow Y$ and $YW \rightarrow Z$ then $XW \rightarrow Z$.

(We use XY to denote the union of the sets X and Y. The sets are not assumed to be disjoint.)

Let F be a given set of FD's. We can apply these rules to the FD's in F to derive additional FD's. The set of all FD's that are derivable from F by repeated applications of the rules (including the FD's in F) is called the *closure* of F and is denoted by F^+.

Clearly, if all the FD's in F are valid in the extension of a relation scheme, then all FD's in F^+ are valid in it. Armstrong's completeness result can be stated as follows: For any set F of FD's there exists a relation in which all the FD's in F^+ are valid and no other FD is valid. Thus Rules A1, A2, and A3 are sufficient to derive any FD that is implied from the given FD's.

While the closure of a given set of FD's can be viewed as the complete information contained in the set, there are good reasons to avoid dealing with it directly. First, even for relatively small sets, the closure may contain a prohibitively large number of FD's. Also, the closure contains redundant information since one can derive all the FD's in the closure from subsets of the closure (e.g. from the original set). Formally, if F is a set of FD's, then a *covering* of F is any set of FD's that has the same closure as F. (A covering is not required to be a subset of F.) A covering is *nonredundant* if it does not contain any proper subset that is also a covering. An FD f in a set of FD's F is *redundant in F* if f can be derived from $F - \{f\}$ (i.e. $f \in (F - \{f\})^+$). Thus a covering is redundant if and only if it contains a redundant FD.

In the rest of Section 2 we assume that there is a given set of FD's, F, ranging over a set of attributes $\{A_1, \ldots, A_m\}$. All relation schemes range over subsets of these attributes. Hence, the FD's in F apply to these schemes. When we write $X \rightarrow Y$ we mean that an FD from X to Y exists in F^+. Similarly, the notation $X \not\rightarrow Y$ means $X \rightarrow Y$ is not in F^+.

2.3 Superkeys and Keys

Let $\mathbf{R}(A_1, \ldots, A_n)$ be a relation scheme and let X be a subset of $\{A_1, \ldots, A_n\}$. We say that X is a *superkey* of **R** if every attribute in $\{A_1, \ldots, A_n\}$ functionally depends on X. The set X is a *key* of **R** if it is a minimal superkey, that is, if it is a superkey and it does not properly contain any superkey.

We note that to specify that X is a superkey (or a key) of a relation scheme $\mathbf{R}(A_1, \ldots, A_n)$ is equivalent to saying that each of the FD's $X \rightarrow A_1, \ldots, X \rightarrow A_n$ must hold in any extension of **R**. The specification of keys is just an alternative way of specifying FD's and, therefore, problems related to keys are actually problems about FD's.

2.4 Relation Schemes and Relational Schemas

The purpose of any data model is to allow the user of the model to describe and manipulate those objects and relationships among objects in the real world that he intends to store in the database. In the relational model this description is given by a relational schema. We define a *relation scheme* to be a description of a relation consisting of a name, a set of attributes, and a list of one or more subsets of this set that are selected as *designated keys* of the relation scheme (notationally, designated keys are underlined). A *relational schema* is simply a set of relation schemes (e.g. see Figure 1).

As we have already noted, the listing of designated keys in the schema is in fact a specification of FD's that must be valid in the database at all times. Formally, we say that an FD $X \rightarrow A$ is *embodied* in relation scheme **R** if X is a designated key of **R** and A is an attribute of **R**. The set of FD's that is embodied in a relational schema is the union of all the FD's embodied in the relation schemes of the schema. The set of FD's that is *represented* in the schema is the closure of the set of FD's that is embodied in it.

AUTHOR(AUTHORNAME, BIRTHDATE, COUNTRY)
BOOK(ISBN, TITLE, AUTHORNAME, NO_OF_EDITIONS)
LIBRARY(LIBNAME, CITY)
INSTOCK(ISBN, EDITION, LIBNAME, QTY)
INDEX(ISBN, SUBJECT)

Fig. 1. A relational schema for libraries

Note that FD's exist in a schema only insofar as they are represented by the schema. This diverges from Codd's definition where FD's are independent constraints that augment the schema [12]. We view an initial database description as containing a set of FD's (and, probably, more integrity information). A schema that describes the database has to represent the closure of the given set of FD's. Each FD specified by the DBA should be either embodied in the schema or derivable from the FD's embodied in the schema. An advantage of this approach is that it facilitates the development of the theory that is necessary for understanding the structure and properties of relational schemas (see e.g. [4, 24]). Indeed, one of the main goals of all methods published so far for constructing a schema from a set of FD's is to find a scheme that represents the given set (e.g. [4, 8, 15, 27]).

It is important to distinguish between the keys of a relation scheme **R** and the sets that are *designated keys* of **R.** A set of relation schemes is selected and a collection of sets of attributes are designated as keys to represent the given set of FD's. However, not all the keys of the relation scheme may have been among the sets designated as keys. Furthermore, some of the keys that are designated may not in fact be keys but may only be superkeys. This occurs if all attributes of a relation scheme are functionally dependent on a proper subset of the designated key. As an example of the latter case, in Figure 2 the relation scheme **EDITION** contains a designated key {ISBN, TITLE} that is only a superkey, since {ISBN} alone is a key of **EDITION.** In **BOOK,** the key {TITLE, AUTHORNAME} is not designated at all. In what follows, we will present an efficient algorithm to check if a designated key is indeed a key and, if it is only a superkey, an efficient algorithm to convert it into a key will be presented.

2.5 Normal Forms

Codd has observed that the existence of certain patterns of FD's among the attributes of a relation scheme results in undesirable properties of its extensions. This led him to define two normal forms for relations which he called second and third normal form [12]. (First normal form, also defined by Codd, restricts the possible values of attributes and has nothing to do with FD's.) To introduce these normal forms we need some definitions.

Let **R** be a relation scheme (with some keys). If an attribute A of **R** belongs to some key of **R** then A is said to be *prime* in **R.** Otherwise, A is *nonprime* in **R.** For example, in relation scheme **TENANT** of Figure 3(c), NAME is prime but ADDRESS and APT# are nonprime. Note that the fact that A belongs to a superkey of **R** does not imply that A is prime.

Given FD's: ISBN → TITLE, AUTHORNAME
TITLE, AUTHORNAME → ISBN
ISBN, TITLE → NO_OF_EDITIONS

Relation schemes:
EDITION(ISBN, TITLE, NO_OF_EDITIONS)
BOOK(ISBN, TITLE, AUTHORNAME)

In **EDITION**, the designated key {ISBN, TITLE} is only a superkey. In **BOOK**, {TITLE, AUTHORNAME} is a key that is not designated.

Fig. 2. Designated keys

Let $X \rightarrow A$ be an FD. We say that A is *partially dependent* on X if A is functionally dependent on a proper subset of X. That is, there exists a set Y such that $Y \subset X$, and $Y \rightarrow A$. Attribute A is *fully dependent* on X if it is not dependent on any proper subset of X. A relation scheme **R** is in *second normal form* (2NF) if each of its nonprime attributes is fully dependent on each of its keys. In Figure 3(b), **DWELLER** is in 2NF, but **APARTMENT_TYPE** is not, since LANDLORD is partially dependent on the key {APT_TYPE, ADDRESS}.

Let **R** be a relation scheme, let X be a subset of the set of attributes of **R**, and let A be any attribute of **R**. We say that A is *transitively dependent* (in **R**) on the set X if there exists a set Y of attributes of **R** such that $X \rightarrow Y$, $Y \not\rightarrow X$, $Y \rightarrow A$, and A does not belong to Y. A relation scheme is in *third normal form* (3NF) if none of its nonprime attributes is transitively dependent on any of its keys. In Figure 3(b), **DWELLER** is not in 3NF because RENT is transitively dependent on NAME. In Figure 3(c), all relations are in 3NF.

Note that if a nonprime attribute is partially dependent on a key then it is also transitively dependent on that key. Therefore, every relation scheme in 3NF is also in 2NF.

These two normal forms have been discussed extensively in the literature (see e.g. [8, 12, 14]), and it has been shown that in many cases certain problems relating to update anomalies and internal consistency disappear when the relation schemes are transformed into 3NF. However, since the definitions of the normal forms involve only the nonprime attributes, some of these problems still remain for prime attributes [14]. This leads to a still stronger normal form [13]. A relation scheme is in *Boyce-Codd normal form* (BCNF) if, for all disjoint nonempty sets of attributes X and Y in **R**, if $X \rightarrow Y$ then X is a superkey of **R**. In other words, if some attributes of **R** depend on X, then all attributes of **R** depend on X. It follows from the definitions that if a relation scheme is in BCNF then it is also in 3NF. The converse is not true. For example, the relation scheme **R**(ABC) with the FD's $AB \rightarrow C$ and $C \rightarrow B$ is in 3NF but is not in BCNF. The schema in Figure 3(c) is in BCNF.

Some researchers do not acknowledge a significant difference between BCNF and 3NF (e.g. [14], pp. 163-164). We disagree. In Section 6, we will show that BCNF and 3NF differ in two important ways: in their ability to represent FD's and in the computational tractability of finding schemas that satisfy them.

APT_TYPE, ADDRESS → SIZE
ADDRESS → LANDLORD
ADDRESS, APT# → RENT
NAME → ADDRESS, APT#
(a) The given set of FD's

DWELLER(NAME, ADDRESS, APT#, RENT)
APARTMENT_TYPE(APT_TYPE, ADDRESS, LANDLORD, SIZE)
(b) A schema that violates the normal forms

TENANT(NAME, ADDRESS, APT#)
APARTMENT(ADDRESS, APT#, RENT)
APT_KIND(APT_TYPE, ADDRESS, SIZE)
BUILDING(ADDRESS, LANDLORD)
(c) A third normal form schema representing the above FD's

Fig. 3. Examples of second and third normal forms

2.6 The Problems

All of the problems treated in this paper relate to FD's. We begin with some basic algorithms to solve fundamental FD problems and build on these results to examine certain schema design questions. All of the problems we attack are defined on a finite universe, and hence are trivially amenable to algorithmic solution. Thus our goal is really to investigate the *efficiency* of these algorithms.

Virtually every interesting algorithmic question relating to FD's requires manipulating "derivations" of one FD from a set of other FD's using Armstrong's inference rules. Formal derivations can be quite cumbersome to use and to understand. To facilitate the manipulation of FD's, we introduce in Section 3 a graph model, called derivation trees, representing derivations. We prove a variety of useful properties of this model, including the fact that derivation trees and formal derivations are indeed equivalent concepts.

Despite the fact that derivation trees and formal derivations are equivalent concepts, it is much easier to understand and use derivation trees than it is to use derivations. We illustrate this by using derivation trees to prove a property of FD's and coverings.

To deal with (almost) any problem involving FD's, one must be able to determine for any given FD and any given set of FD's whether the given FD can be derived from the given set. We call this problem the *membership problem* and present a fast algorithm to solve it in Section 4. In Section 5 we present, as applications of our membership algorithm, efficient algorithms for the solution of several other problems about FD's. In particular, we outline a fast algorithm for finding nonredundant coverings and an efficient implementation of Bernstein's algorithm for synthesizing 3NF schemas [8].

Given the success in being able to construct algorithmically (and efficiently) 3NF schemas from FD's, it is natural to inquire to what extent this approach can be extended to cover BCNF, a stronger normal form. The answer we will reach is that *virtually no extensions are possible.* In particular, in Section 6 we show:

(1) Not every set of FD's can be represented by a BCNF relational schema.
(2) It is computationally very difficult (i.e. $\mathcal{NP}$-hard) to determine if a given relational schema representing a set of FD's is in BCNF.
(3) It is computationally very difficult to determine if a given set of FD's can be represented by a BCNF schema.

In Section 7 we examine a problem that turns out to be closely related, that of determining whether a relation scheme has keys in addition to the designated ones, and show this problem to be computationally difficult as well. It is interesting to note that it follows from our methods of proof that all of these problems remain computationally difficult even if one restricts one's attention just to those schemas generated by a schema synthesis algorithm.

3. PROPERTIES OF FD'S AND DERIVATIONS

3.1 Additional Inference Rules

In the previous section we presented three inference rules for FD's. While these three rules are complete for FD's, it is convenient to introduce two additional rules. These rules are, of course, logical consequences of the above-mentioned rules. They will be useful in proving later results.

Rule A4: (Union). If $X \rightarrow Y_1$ and $X \rightarrow Y_2$ then $X \rightarrow Y_1Y_2$.

Rule A5: (Decomposition). If $X \rightarrow Y$ and $Y' \subset Y$ then $X \rightarrow Y'$.

To prove the validity of Rule A4 we first augment both sides of the FD $X \rightarrow Y_2$ by Y_1 to obtain $XY_1 \rightarrow Y_1Y_2$. The FD $X \rightarrow Y_1Y_2$ now follows by pseudotransitivity from $X \rightarrow Y_1$ and $XY_1 \rightarrow Y_1Y_2$. To prove the validity of Rule A5 we first use reflexivity to obtain $Y \rightarrow Y'$; the FD $X \rightarrow Y'$ now follows by transitivity.

It follows from Rules A4 and A5 that the FD $X \rightarrow A_1 \ldots A_k$ is equivalent to the set of FD's $\{X \rightarrow A_1, \ldots, X \rightarrow A_k\}$. That is, the FD $X \rightarrow A_1 \ldots A_k$ is a short notation for $\{X \rightarrow A_1, \ldots, X \rightarrow A_k\}$. Thus in most cases, we can restrict our attention to FD's that have a single attribute on their right side. In particular, to show that $X \rightarrow A_1 \ldots A_k$ is derivable from a given set F of FD's, it is enough to show that each of the FD's $X \rightarrow A_1, \ldots, X \rightarrow A_k$ is derivable from F. Therefore, from now on we assume (without loss of generality) that all FD's have single attributes on their right sides (unless explicitly stated otherwise).

3.2 Derivation Trees as a Model for Derivations

Let F be a given set of FD's where each FD in F has a single attribute on its right side. A *derivation* of an FD f from F is a sequence $[f_1, \ldots, f_n]$ such that $f_n = f$ and for each i, $1 \leq i \leq n$, one of the following holds:

(1) The FD f_i is in F.

(2) The FD f_i is the result of invoking Rule A1, that is, it is a "reflexive" FD of the form $Y \rightarrow Y'$ where $Y' \subseteq Y$.

(3) The FD f_i is the result of applying the Rule A2 (augmentation) to one of the FD's $f_1, \ldots, f_{i-1}$.

(4) The FD f_i is the result of applying the Rule A3 (pseudotransitivity) to two of the FD's $f_1, \ldots, f_{i-1}$.

For simplicity, a derivation is defined as a sequence of FD's only. The rules that produce the FD's in the sequence are not included as part of the derivation, since it is usually easy to select a correct rule to apply in each step. Sometimes, more than one rule may be applicable in a given step to produce an FD in the sequence.

The concept of derivation as defined here is the one used in formal systems. However, for the purpose of dealing with FD's, formal derivations have several disadvantages. First, derivations are linear, so there may exist several derivations of a given FD that differ only in the order of application of the rules. These derivations are essentially equivalent, but this fact may be quite difficult to observe because of the imposed linear order. Second, a derivation may contain applications of rules that are essentially redundant. For example, in Figure 4(a) there is no need to use $B \rightarrow C$ (Step 4) followed by augmentation (Step 6) and reflexivity (Step 9) in the derivation; one can derive $X \rightarrow AB$ more directly as shown in Figure 4(b). Third, some steps of the derivation that are formally necessary do not intuitively add new information—particularly augmentation and reflexivity steps. In Figure 4(b), for example, by Step 3 the conclusion is already quite obvious; the augmentation step adds no new insight into the origin of $X \rightarrow AB$.

Given: $F = \{X \rightarrow A;\ A \rightarrow B;\ B \rightarrow C\}$
Derive: $X \rightarrow AB$

Step 1. $X \rightarrow A$	(given)
Step 2. $A \rightarrow B$	(given)
Step 3. $X \rightarrow B$	(Steps 1, 2 and Rule A3)
Step 4. $B \rightarrow C$	(given)
Step 5. $X \rightarrow C$	(Steps 3, 4 and Rule A3)
Step 6. $ABX \rightarrow ABC$	(Step 5 and Rule A2)
Step 7. $AX \rightarrow ABC$	(Steps 3, 6 and Rule A3)
Step 8. $X \rightarrow ABC$	(Steps 1, 7 and Rule A3)
Step 9. $ABC \rightarrow AB$	(Rule A1)
Step 10. $X \rightarrow AB$	(Steps 8, 9 and Rule A3)

(a) An overly long derivation of $X \rightarrow AB$ (The rules have been added to enhance readability)

Step 1. $X \rightarrow A$	(given)
Step 2. $A \rightarrow B$	(given)
Step 3. $X \rightarrow B$	(Steps 1, 2 and Rule A3)
Step 4. $AX \rightarrow AB$	(Step 3 and Rule A2)
Step 5. $X \rightarrow AB$	(Steps 1, 4 and Rule A3)

(b) A short derivation of $X \rightarrow AB$

Fig. 4. A derivation with redundant steps

To alleviate these problems, we introduce a graph model, called derivation trees, for modeling derivations. After proving that the derivation tree is sufficiently powerful to model derivations, we will argue that the above difficulties disappear if derivation trees are used.

Let F be a set of FD's. Then the set of *F-based derivation trees* (F-based DT) is defined as follows:[1]

Rule 1. If A is an attribute then a node labeled with A is an F-based DT.

Rule 2. If T is an F-based DT with a leaf node labeled with A and the FD $B_1 \ldots B_m \rightarrow A$ is in F, then the tree constructed from T by adding $B_1, \ldots, B_m$ as children of the leaf labeled with A is also an F-based DT.

Rule 3. A labeled tree is an F-based DT only if it so follows by a finite number of applications of Rules 1 and 2.

The basic step in the construction of DT's is the application of Rule 2 of the preceding definition, that is, adding children to a leaf node. As we prove below (in Lemma 1), such an application corresponds to an application of the pseudotransitivity rule in a derivation. The definition of DT's does not contain constructions that correspond to applications of the reflexivity or augmentation rules. By showing that DT's are sufficiently powerful to model derivations, we will see that the role of reflexivity and augmentation in a derivation is really quite limited.

An FD whose derivation is described by a DT is characterized by the DT's root and by its leaf set. If the set of labels of the leaves of a DT is contained in the set of attributes X then we call it an *X-DT*. An X-DT with root labeled A is called a *derivation tree for the FD* $X \rightarrow A$. This terminology is justified by Lemma 1.

LEMMA 1. *Let T be an F-based DT. Let Y be a nonempty set of labels of nodes of T and let X be the set of labels of all leaves of T. Then the FD* $X \rightarrow Y$ *is in* F^+.

PROOF. As we remarked in Section 3.1, it is enough to prove the lemma for the

[1] Derivation trees for FD's differ from those for context-free grammars primarily because the former deal with sets while the latter deal with strings. That is, FD's are like context-free grammars in which arbitrary repetitions and permutations of symbols are permitted.

case that Y is a single attribute. We also note that it is enough to prove the lemma for the case that Y is the label of the root. For if Y labels a node other than the root, then this node roots a subtree with leaf set X′, where $X' \subseteq X$. Once $X' \rightarrow Y$ is proved, $X \rightarrow Y$ follows by augmentation.

Let Y be the label of the root node. We prove the lemma by induction on the number of applications of Rule 2 in the definition of DT construction. For the basis, if the tree contains no applications of Rule 2, then it consists of a single node labeled Y. That the FD $Y \rightarrow Y$ is in F^+ follows by reflexivity.

Suppose now that we have a tree containing n applicatons of Rule 2. Choose a node of the tree whose children are all leaves. Suppose that the node is labeled with A, that X′ is the set of labels of A's children and that X″ is the set of labels of all other leaves of the tree. (Note that A may belong to X″ since more than one node may have the label A.) Now we know that $X' \rightarrow A$ is in F. By the induction hypothesis we know that $AX'' \rightarrow Y$ is in F^+. It follows by pseudotransitivity that $X'X'' \rightarrow Y$ is in F^+. But X′X″ is exactly X, the set of labels of the leaves of the tree. □

By Lemma 1, a DT with root A and leaf set X represents a derivation of $X \rightarrow A$. We now prove that the converse is also true.

THEOREM 1. *The FD $X \rightarrow Y$ is in F^+, where X, Y are sets of attributes, if and only if for each attribute A in Y there exists an F-based X-DT with root labeled A.*

PROOF. By Lemma 1 we know that if an F-based X-DT with root labeled A exists, then $X \rightarrow A$ is in F^-. Therefore, if such a tree exists for each A in Y then $X \rightarrow Y$ is also in F^+.

To prove the converse we use induction on the length of the (shortest) derivation of $X \rightarrow Y$. First, suppose that $X \rightarrow Y$ has a derivation of length 1, that is, the derivation consists of the single FD $X \rightarrow Y$. There are two cases to consider. If $X \rightarrow Y$ is in F, then Y is a single attribute and the tree with root labeled Y and children of the root labeled with the elements of X is the required DT. If $X \rightarrow Y$ is inferred by reflexivity then $Y \subseteq X$. In this case, for each A in Y, the single node labeled with A is the required tree.

Suppose now that $X \rightarrow Y$ has a shortest derivation of length n. We consider the rules that may be used in the last step to produce $X \rightarrow Y$ (which is the last FD in the derivation). Rule A1 is not used since otherwise $X \rightarrow Y$ would have a derivation of length 1. If Rule A2 is used then there exists in the derivation an FD $X' \rightarrow Y'$ and there exists sets Z and W such that $Z \subseteq W$ and $X = X'W$ and $Y = Y'Z$. Now, the FD $X' \rightarrow Y'$ has a shorter derivation so, by the induction hypothesis, for each A in Y′, there exists an F-based X′-DT with root labeled A. Since $X' \subset X$, this is also an X-DT. For an attribute $A \in Z\text{-}Y'$, since $Z \subseteq W \subseteq X$, the single node labeled with A is an F-based X-DT with root labeled A. So, for all A in Y, there is an F-based X-DT with root labeled A.

The last case to consider is that $X \rightarrow Y$ is produced by an application of Rule A3. That is, there exist in the derivation two FD's $X' \rightarrow Z$ and $ZW \rightarrow Y$ such that $X'W = X$. Let $A \in Y$ be given. Since $ZW \rightarrow Y$ has a shorter derivation, we know that there exists an F-based ZW-DT with root labeled A. Let $B_1, \ldots, B_k$ be the labels of leaves of this tree that belong to Z. Applying the induction hypothesis to the FD $X \rightarrow Z$ we obtain the existence of F-based X-DT's with roots labeled with $B_1, \ldots, B_k$, respectively. If we replace each leaf labeled B_i in the first tree by the X′-DT with root labeled B_i, we obtain an F-based DT with root labeled A and leaf set contained in $X'W = X$. This is the required tree. □

(Note: In order for the proof to carry through, we needed to be able to use the induction hypothesis to infer the existence of an X′-based DT for each B in Z. This is why we formulated the lemma with a set and not with a single attribute on the right side of the given FD.)

COROLLARY 1. *Let $f{:}X \rightarrow A$ be in F^+, where A is not in X. Then there exists a derivation of f from F in which reflexivity is not used and augmentation is either not used at all or used only in the last step.*

PROOF. Since $X \rightarrow A$ is in F^+, there exists an F-based X-DT with root labeled A. Let the set of labels of the leaves of the tree be X′, where $X' \subseteq X$. Using the construction described in the proof of Lemma 1 we obtain a derivation of $X' \rightarrow A$ from F in which only pseudotransitivity is used. If X′ is a proper subset of X then we need one application of augmentation, augmenting the left side of $X' \rightarrow A$ by $(X\text{-}X')$, to obtain a derivation for $X \rightarrow A$. □

We note that the corollary could be proved directly for derivations, without using DT's. However, because of the linear nature of derivations, the proof would be very tedious and cumbersome.

We see, therefore, that the DT, being nonlinear in nature, can serve as a tool for proving properties of and dealing with derivations. Going back to the disadvantages we listed for derivations, we can now see how they disappear when DT's are used. First, derivations that differ only by the order of application of the rules are modeled by the same DT. Furthermore, looking at the derivations that one can construct from a given DT by going from the root to the leaves or from the leaves to the root, we see that the same DT represents derivations that actually use different rules and therefore may not superficially be considered to be equivalent. For this reason, the concept of DT will probably prove to be very useful for the investigation of problems like "Does a given FD have two essentially different derivations from a given set?"

Another disadvantage we mentioned was that a derivation could contain redundant applications of the rules. Looking back at the example of redundancy we gave there, we can now see that if a DT was constructed for that derivation (using the method in the proof of Theorem 1), this redundancy would be eliminated. In general, DT's may still contain some redundancy; however, the redundancy can usually be easily discovered and eliminated (for example, see Lemma 3 in the next subsection).

We have seen that if we construct a derivation from a given DT going from the root to the leaves then only Rule A3 needs to be used. We now give a characterization of the derivation that results by going from the leaves to the root. Let us call an application of augmentation *restricted* if attributes are added only to the left side of the FD.

COROLLARY 2. *Let $f{:}X \rightarrow A$ be in F^+, where A is not in X. Then there exists a derivation of f from F in which only the rules pseudotransitivity, union (Rule A4), and restricted augmentation are used.*

PROOF. If one constructs a derivation from a given DT, starting at the leaves, then only these rules need to be used. □

3.3 Additional Properties of Derivation Trees

We conclude this section with three useful lemmas about derivation trees.

Let an occurrence of an FD f in a derivation of an FD g be *redundant* if it is possible to eliminate this occurrence of f (and, possibly, some other FD's) from the derivation and obtain a derivation for the same FD, g.

LEMMA 2. *Let F be a set of FD's and let $f{:}X \rightarrow A$ be in F. If f is used nonredundantly in some derivation from F of an FD $g{:}\, V \rightarrow W$ in F^+, then $V \rightarrow X$ is also in F^+* [8].

PROOF. We may asssume that W is a single attribute. Given a derivation of g from F we can construct an F-based V-DT with root labeled W that represents it, as in the proof of Theorem 1. This construction may eliminate FD's that are redundant in the derivation. However, every FD from F that is not redundant will appear in the DT. It follows that the FD X → A appears in the DT and the result now follows by Lemma 1. □

A priori, derivation trees can be arbitrarily large. The following lemma states that, for all practical purposes, we can restrict our attention to "small" trees. The lemma can be viewed as a way of removing redundancy from derivation trees. It is essentially the same as a well-known result about derivation trees in the theory of context-free languages.

LEMMA 3. *If $f \in F^+$ then there exists an F-based DT for f in which no path from the root to a leaf contains more than one occurrence of any attribute.*

PROOF. See for example, the proof of Theorem 4.1 in [20]. □

An FD X → Y is *reduced* in F^+ if there is no X′ ⊂ X such that X′ → Y is in F^+. Using this concept, the following lemma shows how DT's can be used for proving properties of coverings of sets of FD's.

LEMMA 4. *Let F be a set of FD's with $X \rightarrow A$ $(A \notin X)$ in F^+.*

(*a*) *If for each FD $Y \rightarrow A$ $(A \notin Y)$ in F^+, $X \rightarrow Y$ implies $X \subseteq Y$, then $X \rightarrow A$ is a member of every covering of F that contains only reduced FD's.*

(*b*) *If for all $Y \subset X$ $(Y \neq X)$, F^+ does not contain any FD $Y \rightarrow B$, then each covering of F contains an FD with left side X.*

PROOF. Let G be a covering of F where X → A is not in G. By Theorem 1, there exists a G-based X-DT, T, for X → A.

Suppose that the hypothesis of condition (a) holds and that G contains only reduced FD's. Let us look at the root FD of T, Y → A. By Lemma 1, X → Y. So, X ⊆ Y. But since G contains only reduced FD's, and Y → A is in G, it follows that X = Y, thereby proving condition (a).

To prove condition (b), observe that each FD whose left side is contained in the leaves of T has a left side that is contained in X. By the hypothesis of condition (b), no FD has a left side properly contained in X. So, their left sides must be precisely X. □

Condition (a) of Lemma 4 is only a sufficient condition for X → A to be in every covering, but it is not a necessary one. For a necessary and sufficient condition for an FD to belong to every covering see [23].

4. THE MEMBERSHIP PROBLEM

4.1 Definitions

The *membership problem for FD's* is: Given a set of FD's, F, and an FD, f, determine if $f \in F^+$. In this section we develop an efficient *membership algorithm* to solve this problem.

4.2 A Naive Membership Algorithm

By Theorem 1, $f \in F^+$ if and only if there exists an F-based DT for f. By Lemma 3, such a DT (if it exists) is of bounded size. Therefore, we can solve the membership problem simply by enumerating the "small" trees and checking each one to see if it is a DT for the given FD. However, this enumeration algorithm is too time consuming. To improve upon it, we begin with a fairly straightforward quadratic-time membership algorithm that we will later refine into a linear-time version.

Let $F = \{f_1, \ldots, f_n\}$ be a set of FD's defined over the set of attributes $\{A_1, \ldots, A_m\}$. To obtain a shorter description of F, we assume that all FD's in F that have the same left side are grouped into a single FD (by applying the union rule, Rule A4). The set F is represented as a string of pairs, where each pair represents an FD and consists of a left side (LS) and a right side (RS). Each side is a set of attributes, where attributes are represented by integers in the set $\{1, \ldots, m\}$. The length of the representation of F is denoted by $|F|$.

Let f be $X \rightarrow A$ where both X and $\{A\}$ are subsets of $\{A_1, \ldots, A_m\}$. Since each attribute in f appears in at least one FD of F, we can assume that $|f| \leq |F|$.

Our method to check if $f \in F^+$ is to compute the set of attributes that are functionally dependent on X. Let DEPEND be a set-valued variable to hold these attributes. Initially, we assign the value X to DEPEND, since by reflexivity X is functionally dependent on X. To find new attributes to add to DEPEND, we select an FD, f', in F whose left side is contained in DEPEND, but whose right side is not. By pseudotransitivity, the right side of f' is functionally dependent on X and can be added to DEPEND; that is, if $X \rightarrow$ DEPEND and f': $Y \rightarrow B$ where $Y \subseteq$ DEPEND, then $X \rightarrow B$. We can continue selecting FD's in this way, adding their right sides to DEPEND, until no new FD's qualify.

Conceptually, DEPEND contains a set of attributes each of which is the root of an F-based X-DT. Each time we find an FD, f', whose left side is contained in DEPEND, the second DT construction rule states that each attribute on the right side of f' can root an F-based X-DT. That is, the process is a bottom-up construction of X-DT's. The process stops only when no new roots for an X-DT can be found. We know that $f \in F^+$ if and only if A is in DEPEND at the completion of the process.

The method is formally implemented as Algorithm 1. Its proof of correctness easily follows, say by induction on the depth of the DT for f, and is left to the reader.

Given a set of attributes, X, we define the *closure of* X (relative to F), denoted by X^+, to be the set of attributes that are functionally dependent (in F^+) on X. That is, X^+ contains the set of all attributes A such that $X \rightarrow A \in F^+$. Algorithm 1 first computes X^+ (called DEPEND in the algorithm) and then checks if

$A \in X^+$. Therefore, the only change in the algorithm required to handle a compound right side of f (i.e. $f{:}X \to A_1 \dots A_p$) is to check that each of $A_1 \dots A_p$ is in DEPEND (i.e. X^+) in the PRINT step.

To obtain a worst-case time bound for Algorithm 1, we observe that in each iteration of the loop labeled FIND_NEW_ATTR (except the final iteration) at least one new attribute is added to DEPEND. So, in the worst case the number of iterations of FIND_NEW_ATTR may be close to m. In each such iteration, the loop labeled CHECK_FDS scans the input string F. Therefore, in the worst case, the total time spent by the algorithm is proportional to $m|F|$ [i.e. $0(m|F|)$].

ALGORITHM 1. A QUADRATIC-TIME MEMBERSHIP ALGORITHM FOR FD'S

INPUT: A set F of n FD's on attributes $\{A_1, \dots, A_m\}$ and an FD $f{:}X \to A$.
OUTPUT: "YES" if $f \in F^+$; "NO" if $f \notin F^+$.
DATA STRUCTURES:

(1) Attributes are represented by integers between 1 and m.
(2) FD's in F are represented by integers between 1 and n.
(3) LS[1:n], RS[1:n] are arrays of sets, containing the attributes in the left and right side of each FD.
(4) DEPEND is a set of attributes found to be functionally dependent on X so far.
(5) DONE is a Boolean variable.

```
ALGORITHM:
begin
           INITIALIZE: DEPEND = X;
                       DONE = FALSE;
FIND_NEW_ATTR: do while (not DONE);

                       DONE = TRUE;
        CHECK_FDS:     do i = 1 to n;
                         if ((LS[i] ⊆ DEPEND) &
                                 (RS[i] ⊈ DEPEND))
                         then begin
                                 DEPEND = DEPEND ∪ RS[i];
                                 DONE = FALSE
                               end
                           end CHECK_FDS
                       end FIND_NEW_ATTR;
                PRINT: if A ∈ DEPEND then print 'YES'
                                     else print 'NO'
end
```

4.3 A Linear-Time Membership Algorithm

Algorithm 1 has several inefficiencies all of which revolve around a central problem: In each iteration of FIND_NEW_ATTR all of F is scanned even though only a small part of it is involved in any operation. For example, once an f_i satisfies the condition (LS[i] $\subseteq$ DEPEND & RS[i] $\subseteq$ DEPEND), it need never be examined again. In fact, once an attribute in LS[i] is found to be in DEPEND, that *attribute* need never be examined again. Similarly, an attribute in RS[i] need only be examined once, namely, when all of LS[i] is found to be in DEPEND. By taking advantage of these observations, we can arrange to examine each occurrence of an attribute in F only once in FIND_NEW_ATTR.

To avoid reexamining attributes in F more than once, we need auxiliary data structures. We construct a linked-list for each attribute appearing in F; the linked-list for an attribute contains a pointer to each FD that has that attribute on its left side. In addition, we associate a counter with each FD; the counter initially specifies the number of attributes on the left side of the FD. The linked-lists and counters can be constructed in a single pass over F.

The fast membership algorithm (see Algorithm 2) operates essentially as in Algorithm 1 by successively adding new attributes to DEPEND. When a new member of DEPEND is found, it is removed from all of the left sides of FD's on which it appears by following its linked-list and decrementing the counter of each FD on the list. If any such counter is decremented to zero, then the left side of the associated FD is in DEPEND; hence, its right side can be added to DEPEND as well. (Actually, an attribute is added to DEPEND only if it is not already a member of DEPEND.) The algorithm continues until no new members of DEPEND can be found.

4.4 Analyzing the Membership Algorithm

To prove the correctness of Algorithm 2, we first examine the INITIALIZE step. This step consists primarily of a scan of F, performing a constant number of operations for each attribute on the left side of an FD. Therefore, this part terminates (and takes time $0(|F|)$). At the end of INITIALIZE, the following hold:

(1) For each f_i in F, COUNTER[i] = $|$LS[i]$|$.
(2) For each A_j in $\{A_1, \ldots, A_m\}$, ATTRLIST[j] contains a list of all FD's with A_j on its left side.
(3) The sets DEPEND and NEWDEPEND are initialized to X.

The main body of the algorithm is the loop labeled FIND_NEW_ATTR. To prove termination of this loop, we note that the loop is executed once for each member of NEWDEPEND. An attribute can only be added to NEWDEPEND by satisfying the if statement in the loop CHECK_FDS. The condition ($j \notin$ DEPEND) prevents an attribute from being added to NEWDEPEND more than once. Thus since at most m attributes can be added to NEWDEPEND, the loop FIND_NEW_ATTR can execute at most m times and therefore must terminate. Hence the entire algorithm terminates.

We now prove that at the beginning of PRINT, DEPEND contains exactly X^+. We first observe that in the beginning of each iteration of FIND_NEW_ATTR, for each i COUNTER[i] equals the number of attributes in LS[i] that are not in (DEPEND-NEWDEPEND). This claim is true at the first iteration because of (1) and (3) above. In an iteration of FIND_NEW_ATTR, one attribute is removed from NEWDEPEND and each FD that contains that attribute on its left side has its COUNTER decremented; so the claim is true after the iteration.

We now show that every attribute added to DEPEND is in X^+. This is true by reflexivity for X. An attribute is added to DEPEND only if it is on the right side

ALGORITHM 2. A LINEAR-TIME MEMBERSHIP ALGORITHM FOR FD'S

INPUT: A set F of n FD's on attributes $\{A_1, \ldots, A_m\}$ and an FD $f{:}X \rightarrow A$.
OUTPUT: "YES" if $f \in F^+$; "NO" if $f \notin F^+$.
DATA STRUCTURES:

(1) Attributes are represented by integers between 1 and m.
(2) FD's in F are represented by integers between 1 and n.
(3) LS[1:n], RS[1:n] are arrays of sets containing the attributes on the left and right sides of each FD.
(4) DEPEND is a set of attributes found to be functionally dependent on X so far.
(5) NEWDEPEND is a subset of DEPEND that has not yet been examined.
(6) COUNTER[1:n] is an array containing the number of attributes on the left side of each FD that have not yet been found to be in DEPEND.
(7) ATTRLIST[1:m] is an array of lists of FD's specifying for each attribute the FD's with that attribute on their left sides.

ALGORITHM:

```
begin
          INITIALIZE: do i = 1 to m;
                           ATTRLIST[m] = 0
                         end;
                         do i = 1 to n;
                              COUNTER[i] = 0;
                              do for each j ∈ LS[i];
                                ATTRLIST[j] = ATTRLIST[j] ∪ {i};
                                COUNTER[i] = COUNTER[i] + 1
                              end
                         end;
                         DEPEND = X;
                         NEWDEPEND = DEPEND;
  FIND_NEW_ATTR: do while (NEWDEPEND ≠ ∅);
                              select NEXT_TO_CHECK from NEWDEPEND;
                              NEWDEPEND = NEWDEPEND-{NEXT_TO_CHECK};
        CHECK_FDS:            do for each i ∈ ATTRLIST(NEXT_TO_CHECK);
                                COUNTER[i] = COUNTER[i] - 1;
                                if(COUNTER[i] = 0)
                                then do for each j ∈ RS[i];
                                     if (j ∉ DEPEND)
                                     then begin
                                       DEPEND = DEPEND ∪ {j};
                                       NEWDEPEND = NEWDEPEND ∪ {j}
                                     end
                                   end
                              end CHECK_FDS
                            end FIND_NEW_ATTRS;
                PRINT: if A ∈ DEPEND
                              then print "YES"
                              else print "NO"
end
```

of an FD whose COUNTER is zero. By the above claim, if the COUNTER is zero, then the left side of the FD is in DEPEND. Thus by the induction hypothesis and pseudotransitivity, the attribute is in X^+.

Finally, we show that every member of X^+ is eventually added to DEPEND by induction on the depth of DT's. (The *depth* of a DT is the length of the longest path from the root to a leaf.) For DT's of depth zero, we need only consider members of X and they are all in DEPEND. Given an attribute C that has a DT of depth i + 1, we examine the root FD of the tree, say $fj{:}B_1 \ldots B_p \rightarrow C$. Each of $\{B_1, \ldots, B_p\}$ roots an X-DT of depth $\leq$i and is, by the induction hypothesis, in DEPEND. When the last member of $\{B_1, \ldots, B_p\}$ is removed from NEWDEPEND, COUNTER[j] will be decremented to zero, thereby adding C to DEPEND (if it is not already there). So, DEPEND contains exactly X^+ as desired.

The time complexity of Algorithm 2 is derived by adding the complexity of INITIALIZE (shown to be $0(|F|)$ above) to that of FIND_NEW_ATTR. For each attribute in NEWDEPEND, the FIND_NEW_ATTR loop follows a constant number of steps for each occurrence of that attribute on the left side of an FD in F. Similarly, each right side of an FD in F is visited at most once in FIND_NEW_ATTR. Thus FIND_NEW_ATTR is also $0(|F|)$ as is the entire Algorithm 2. (Recall that we have assumed $|f| \leq |F|$.)

While the worst-case time of Algorithm 2 is $0(|F|)$, the running time will frequently be much better. First, if F contains many FD's whose left sides are disjoint from X^+, then these FD's will never be visited in FIND_NEW_ATTR. Also, since the running time is a function of the depth of DT's of f, if f has a shallow DT its right side will be added to DEPEND after an early iteration of FIND_NEW_ATTR. To take advantage of such situations, we can add a shortcut by exiting the FIND_NEW_ATTR loop as soon as C is found to be in DEPEND. (All attributes that are the roots of X-DT's of depth $\leq$ N are added to DEPEND before any attribute that is only the root of X-DT's of depth > N.) All of these considerations are important when many membership tests based on one set of FD's are performed. Furthermore, in this case INITIALIZE need only be done once and the data structure reused for each membership test.

THEOREM 2. *The membership problem for FD's is solvable in linear time.*

5. APPLICATIONS OF THE MEMBERSHIP ALGORITHM

5.1 Introduction

The membership algorithm is sufficient to solve several other FD problems that are related to automatic schema synthesis. We present solutions to these problems in Section 5.2 and show their application to one schema synthesis algorithm in Section 5.3.

We note in passing that FD problems are closely related to certain problems in propositional calculus. For example, the FD membership algorithm also provides a linear-time test to decide if a disjunctive normal form propositional formula with at most one negated literal per clause is a tautology. The connection between FD's and propositional logic is developed in [18].

5.2 Redundancy Tests

Let F be a given set of FD's and let $f{:}X \rightarrow A$ be an FD in F^+. An attribute B is *extraneous in f* (relative to F) if $B \in X$ and $(X - \{B\}) \rightarrow A$ is in F^+. If f contains no extraneous attributes, then it is reduced.

To determine if B ϵ X is extraneous in *f*, one can apply the linear-time membership algorithm to test if $(X - \{B\}) \rightarrow A$ is in F^+. A subset of X that determines A but does not contain extraneous attributes can be found using the following simple procedure:

```
X' = LS[f];
do for each B ∈ LS[f];
  if (X' − {B}) → A is in F+
    then X' = X' − {B}
end
```

The final value of X′ may depend on the order in which elements of LS[*f*] are selected, since A may be functionally dependent on several incomparable proper subsets of X.

This reduction procedure is effective for FD's in *F* (as well as those in $F^+ - F$). Therefore, it can be used to find a set of reduced FD's, say F_r, with $F_r^+ = F^+$. This is accomplished by eliminating extraneous attributes from all $f_i \in F$. Since each extraneous attribute is eliminated in time $0(|F|)$, the entire reduction procedure for finding F_r takes time $0(|F|^2)$.

Another application of the membership test is to eliminate redundant FD's. Recall that an FD, *f*, is *redundant* in a set of FD's, *F*, if $(F - \{f\})^+ = F^+$ (see Section 2). A *nonredundant cover* of *F* is a set of FD's, *G*, where $G^+ = F^+$ and *G* contains no redundant FD's.

To determine if *f* is redundant in *F*, one can apply the linear-time membership algorithm to test if *f* is in $(F - \{f\})^+$. A subset of *F* that is a nonredundant cover of *F* can be found using the following procedure:

```
G = F;
do for each f ∈ F;
if f ∈ (G − {f})+ then G = G − {f}
end
```

As in the case of extraneous attributes, the final value of *G* may depend on the order in which FD's in *F* are selected in the loop. Note also that *G* is always a subset of *F* (although this is not a requirement of nonredundant covering algorithms). The running time of the covering algorithm is bounded by $0(n|F|)$, where n is the number of FD's in *F*.

Many other problems related to FD's are solved by using the membership algorithm. Key finding problems are a good example. Given a relation scheme $R(A_1, \ldots, A_r)$, we can test if $X \subseteq \{A_1, \ldots, A_r\}$ is a superkey by testing if $X \rightarrow A_1, \ldots, A_r$ is in F^+. To test if a superkey is a key is equivalent to testing if $X \rightarrow A_1, \ldots, A_r$ is reduced. Also, this method can be used to efficiently test if a designated key is really a key or simply a superkey.

5.3 An Implementation of a Schema Synthesis Algorithm

The primary application of FD algorithms is in the implementation of schema synthesis algorithms. A schema synthesis algorithm takes a given set of FD's that describes an enterprise and produces a relational schema that represents these FD's. One such algorithm that produces a 3NF schema was recently proposed in [8] and is shown in Algorithm 3. The essential technique used by Algorithm 3 (and by most other synthesis algorithms) is to distill the given set of FD's down to a highly nonredundant version, and then to form a relation scheme for each set of FD's that have the same left sides. This technique can be implemented using membership tests of various sorts and is evidence for the power of the FD membership algorithm.

The time complexity of Algorithm 3 follows from the discussion in the last section. Step 1 can be accomplished by first eliminating extraneous attributes from FD's in G producing, say G', and then finding a nonredundant covering $H \subseteq G'$. These two steps take time $0(|F|^2)$ and $0(n|F|)$, respectively. Step 2 partitions a set of FD's based on common left sides and is certainly bounded by $0(|F|^2)$ (a faster implementation is described in [9]). The covering algorithm in Step 3 runs in time $0(n|F|)$. Under a suitable representation of H and J, $H + J$ is of size $0(|F|)$, so the covering algorithm in Step 4 requires time $0(n|F|)$. Step 5 requires time $0(|F|)$. Since $n|F| \leq |F|^2$, the entire algorithm has a worst-case time of $0(|F|^2)$. The details of this analysis appear in [9].

ALGORITHM 3. SYNTHESIZING A RELATIONAL SCHEMA FROM A SET OF FD'S

Step 1: (Find covering). Find a nonredundant covering, H, of G that consists of reduced FD's.

Step 2: (Partition). Partition H into *groups* such that all of the FD's in each group have identical left sides.

Step 3: (Merge equivalent keys). Let $J = \emptyset$. For each pair of groups, say Hi and Hj, with left sides X and Y, respectively, merge Hi and Hj together if there is a bijection $X \rightarrow Y$ and $Y \rightarrow X$ in H^+. For each such bijection, add $X \rightarrow Y$ and $Y \rightarrow X$ to J. For each $A \in Y$ if $X \rightarrow A$ is in H, then delete it from H. Do the same for each $Y \rightarrow B$ in H with $B \in X$.

Step 4: (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 the corresponding group of H' from which it was derived in Step 3.

Step 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 *synthesized.* The set of constructed relations constitutes a schema for the given set of FD's.

6. BOYCE-CODD NORMAL FORM

6.1 Updates in BCNF Relation Schemes

Third normal form was introduced to solve certain kinds of update anomalies and consistency difficulties among nonprime attributes in a relation scheme. However, 3NF does not eliminate such problems among prime attributes. For this reason, the strictly stronger Boyce-Codd normal form was introduced [13]. Recall the definition given in Section 2.5: a relation scheme **R** is in BCNF if for all disjoint nonempty sets of attributes X and Y in **R**, if $X \rightarrow Y$ then X is a superkey of **R**.

As we remarked in Section 2.5, every relation scheme in BCNF is also in 3NF, but the converse is not true. The example given there was the relation scheme R(A, B, C) with the FD's AB $\rightarrow$ C, C $\rightarrow$ B. This relation scheme is in 3NF but is not in BCNF. This is a realistic example, as can be seen by choosing for A the attribute ADDRESS, for B the attribute CITY, and for C the attribute POSTALCODE.

In non-BCNF relations, problems arise that are basically the same as those caused by transitive dependencies of nonprime attributes on keys. The reason is that the extensions of FD's in a non-BCNF relation cannot be independently updated. For example, consider the FD's CITY, ADDRESS $\rightarrow$ POSTALCODE and POSTALCODE $\rightarrow$ CITY. One cannot arbitrarily change the POSTALCODE for a particular CITY and ADDRESS, because by doing so one can violate the FD POSTALCODE $\rightarrow$ CITY. This is essentially the consistency problem found in 3NF violations. Also, the insertion of the first CITY, ADDRESS combinaton for a particular POSTALCODE creates a new POSTALCODE $\rightarrow$ 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 [26]), 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 scheme.

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 multiple 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.

6.2 Some Negative Results

We know that given any set F of FD's we can find a 3NF schema that represents it [8]. Since BCNF schemas are apparently preferable to 3NF schemas, we would like to be able to find a BCNF schema that represents F. However, this goal is impossible to fulfill, since there are sets of FD's that cannot be represented by any BCNF schema.

THEOREM 3. *There is a set of FD's that cannot be represented by any BCNF relational schema.*

PROOF. Let F be the set of FD's {AB $\rightarrow$ C, C $\rightarrow$ B}. By a brute force examination of F^+, it can be shown that AB $\rightarrow$ C is in every nonredundant covering of F. Therefore, in any relational schema that represents F, one of the relation schemes must contain A, B, and C. This relation scheme is not in BCNF. □

It has been pointed out that, given the relation scheme **R**(A, B, C) with the FD's AB → C, C → B, it is possible to decompose **R** to obtain relation schemes in BCNF (see [14], and also the definition of 4NF and the accompanying theorem in [17]). The relation schemes would be $\mathbf{R}_1$(B, C) with key C, and $\mathbf{R}_2$(A, C) with key {AC}. We note, however, that the relational schema consisting of $\mathbf{R}_1$ and $\mathbf{R}_2$ represents only the FD C → B. The FD AB → C is not represented. Thus there is a price to be paid for obtaining a BCNF schema in this case, namely, losing the direct representation of the given set of FD's. It is our opinion that this price is not justifiable and that the given set of FD's *should* be represented in the relational schema. However, a discussion of this topic is outside the scope of this paper [4].

The impact of Theorem 3 is that some sets of FD's always lead to non-BCNF schemas and require a special mechanism to solve the integrity problems that arise in databases described by such schemas. What we would like to know is: When a set of FD's can be represented by a BCNF schema, what is the cost involved in finding or constructing such a schema to represent it. We note that a set of FD's may have two coverings such that for one of them the schema that embodies it is in BCNF, but for the other covering the schema that embodies it is not in BCNF (see Figure 5). Thus even when a set of FD's can be represented by a BCNF schema, the task of finding such a schema is not necessarily trivial.

It is obvious that there is an algorithm for constructing a BCNF schema that represents a given set of FD's whenever such a schema exists. This observation is based on the fact that one can check if a given relation scheme is in BCNF (hence also if a given relational schema is in BCNF). This is done by using the membership algorithm to check if there exists a subset of the attributes in the relation scheme such that it functionally determines some but not all of the other attributes in the relation scheme. Thus one can enumerate all relational schemas that represent the given set of FD's and check each one to see if it is in BCNF. However, this algorithm is very slow and may take, in the worst case, exponential time. In the rest of this section we present results that indicate that a polynomial time algorithm for this problem is not likely to be found.

FD's	Relations
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)

FD's	Relations
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 FD's 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.

Fig. 5. Two coverings, only one of which violates BCNF

Clearly, any algorithm that generates a BCNF schema for a given set of FD's when such a schema exists can also be used to decide if the given set of FD's can be represented at all by a BCNF schema. In other words the problem of generating a BCNF schema from a given set of FD's is at least as difficult as the problem of deciding if the given set can be represented by a BCNF schema. Our next theorem states that the complement of this decision problem is $\mathcal{NP}$-hard. It follows that a polynomial time algorithm for generating BCNF schemas from sets of FD's is not likely to be found.

For the statement of the theorem we assume some restrictions on relational schemas. We will assume that the word "schema" means a schema in 3NF in which no two relation schemes have equivalent keys. (Sets X and Y are *equivalent* if $X \rightarrow Y$ and $Y \rightarrow X$ are in the closure of the given set of FD's.) Recall that every set of FD's can be represented by a 3NF schema. Recall also that the equivalence of keys of relation schemes can be tested using the membership algorithm. Thus these restrictions do not exclude any set of FD's from consideration.

Let $\mathcal{P}$ denote the class of problems that can be solved in deterministic polynomial time and let $\mathcal{NP}$ denote the class of problems that can be solved in nondeterministic polynomial time. It is an open problem whether $\mathcal{P} = \mathcal{NP}$. A problem is *$\mathcal{NP}$-hard* if it is provable that if the problem belongs to $\mathcal{P}$ then $\mathcal{P} = \mathcal{NP}$. A problem is *$\mathcal{NP}$-complete* if it is in $\mathcal{NP}$ and is $\mathcal{NP}$-hard. A standard method for showing that a problem is $\mathcal{NP}$-hard is to present a polynomial time reduction of a known $\mathcal{NP}$-complete problem to the given problem. No polynomial time algorithm currently exists for any $\mathcal{NP}$-hard or $\mathcal{NP}$-complete problem, or for the complement of such a problem. Furthermore, if such an algorithm were shown to exist, this would be a proof that $\mathcal{P} = \mathcal{NP}$. So, a proof of $\mathcal{NP}$-completeness of a problem or its complement is tantamount to showing that the problem probably requires exponential time to be solved. For further discussion of $\mathcal{NP}$-completeness see [1].

THEOREM 4. *Given a set U of attributes and a set F of FD's over U. Then: (a) The problem "Does there exist a non-BCNF schema that represents F?" is $\mathcal{NP}$-complete; (b) The problem "Is there no BCNF schema that represents F?" is $\mathcal{NP}$-hard. (Remark: These two problems are not the same. As we have seen, it is possible that a set F can be represented by two schemas exactly one of which is in BCNF. We are interested mainly in the complement of the second problem, namely, does there exist a BCNF schema that represents F.)*

PROOF. See Appendix.

COROLLARY 3. *The following problem is $\mathcal{NP}$-complete: Given a set of FD's and a relational schema that embodies it, does the schema violate BCNF? The problem is $\mathcal{NP}$-complete even when the schema is known to be in 3NF and even when it is produced by Algorithm 3.*

PROOF. See Appendix.

7. KEY FINDING

In this section we treat the problem of checking the possible existence of unknown keys of a relation. Some of the keys of a relation may be designated by a schema synthesis algorithm or by a database designer. However, a relation may have keys in addition to those that are designated. For example, given the set of FD's {AB

→ C, C → B}, the 3NF schema constructed by the synthesis algorithm contains the relations **R1**($\underline{A}$, $\underline{B}$, C) and **R2**($\underline{C}$, B). Clearly, AC is an additonal 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 6. 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 three), 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.

We define the *additional key problem* as follows: Let *F* be a set of FD's defined over $\{A_1, \ldots, A_m\}$, let **R** be a relation scheme consisting of a subset of $\{A_1, \ldots, A_m\}$ and a (possibly empty) set of keys. Does **R** contain an additional key?

THEOREM 5. *The additional key problem is $\mathcal{NP}$-complete.*

PROOF. See Appendix.

Lucchesi and Osborne [22] have treated some related problems in a somewhat restricted model. In their model, one is given a set of attributes and a set of FD's on them; they define 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

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

(a) A given set of FD's

R1($\underline{K}$, $\underline{L1}$, $\underline{L2}$, X1, X2, X3, X4)
R2($\underline{X1}$, M1, M5)
R3($\underline{X2}$, M2, M6)
R4($\underline{X3}$, M3, M7)
R5($\underline{X4}$, M4, M8)
R6($\underline{M1}$, $\underline{M2}$, $\underline{M3}$, $\underline{M4}$, L1)
R7($\underline{M5}$, $\underline{M6}$, $\underline{M7}$, $\underline{M8}$, L2)

(b) The relations synthesized from the above FD's. In the relation **R1**, {X1, X2, X3, X4} is a nonsynthesized key, even though there are no keys of cardinality three.

Fig. 6. A cardinality gap for keys

collected into a single relation scheme. In our approach attributes are collected in several relation schemes. A key of a relation scheme derives all attributes in that relation scheme. It is generally accepted that relational schemas consist of many relation schemes as evidenced by all of the relational systems built to date. The division of the set of attributes into relation schemes 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 exhibit an algorithm that lists all keys in time that is polynomial in the number of attributes and the number of FD's, and linear in the number of keys. (Note that the number of keys may be exponential in the number of attributes and of FD's, in which case the algorithm will take time exponential in the size of the input. For such cases, we doubt that *any* algorithm for generating keys can be considered to be useful.) The algorithm does not seem to generalize to our model. Furthermore, Theorem 5 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 scheme, 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, the number of FD's 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 $\mathcal{NP}$-complete.

Lucchesi and Osborne [22] proved two other problems to be $\mathcal{NP}$-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 $\mathcal{NP}$-complete in the latter model.

The results in this section strongly suggest that key finding is an inherently difficult problem. From Theorem 5 it follows that if $\mathcal{NP} \neq \mathcal{P}$ then there is no algorithm that lists all keys in time polynomial in the size of the relation and the set of FD's. It is true that even if $\mathcal{NP} \neq \mathcal{P}$ none of the results implies that there is no algorithm that lists all keys of a relation in time polynomial in the number of keys. 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. (As we remarked earlier, even if it existed its usefulness is dubious.)

8. CONCLUSIONS

In the first part of the paper we dealt with the basic algorithmic properties of functional dependencies. Starting from the concept of a derivation, we presented a tree model for derivations and illustrated its usefulness. We then presented a linear time membership algorithm for functional dependencies. Using this result, we showed that redundancy can be removed from a set of given functional dependencies in polynomial time. The usefulness of these results was illustrated by an efficient implementation of Bernstein's algorithm for synthesizing relational schemas from functional dependencies. Since a membership test is an essential part of any algorithmic solution to any problem concerning functional dependencies, we expect the results in this part of the paper to serve as a basis for all further work on functional dependencies.

In the second part of the paper we showed that two problems of logical schema design that can be stated in terms of functional dependencies are $\mathcal{NP}$-hard or are the complement of $\mathcal{NP}$-hard problems, that is, are computationally infeasible. Similar results about related problems were obtained by other workers.

We note that the problems in this paper are treated in order of increasing conceptual complexity. The first problem is the membership problem, that is, given a set of functional dependencies F and a functional dependency f, does there exist a derivation for f from F. This problem has an efficient algorithmic solution. The next class of problems is the class of redundancy problems. For example, given a set F, is there a functional dependency f in F that is derivable from $F-\{f\}$. Here, we have an additional bounded existential quantifier. The algorithm we present still works in polynomial time. The next two problems, the BCNF problem and the key finding problem, can both be loosely stated as follows: given a set of functional dependencies F, does there exist a function dependency f, satisfying some general condition, in F^+. (The condition in the case of the key finding problem is that the left side be a subset of a given set of attributes and the right side is this given set.) These problems are $\mathcal{NP}$-hard. Thus we see that the distinction between problems that can be solved in polynomial time and problems that are $\mathcal{NP}$-hard or $\mathcal{NP}$-complete is in the type of the condition that bounds the additional existential quantifier. If this quantifier is not strongly bound so that the required functional dependency can be found in a set of polynomial size then the problem is $\mathcal{NP}$-hard. This applies to all problems that we know of about functional dependencies that were shown to be $\mathcal{NP}$-hard. In particular, this implies that we should avoid algorithms that compute large subsets of the closure since such algorithms seem to be doomed to be inefficient.

APPENDIX

PROOF: Theorem 4. That the first problem is in $\mathcal{NP}$ is quite obvious. One only has to construct nondeterministically a schema that represents the given set F, and nondeterministically select a set of attributes in one of the relations and show it violates BCNF. To prove $\mathcal{NP}$-completeness of (a) and $\mathcal{NP}$-hardness of (b), we now present a polynomial time reduction of a known $\mathcal{NP}$-complete problem to each of the given two problems.

The hitting set problem is formulated as follows: Given a family $\{V_i\}$ $i = 1, \ldots, n$ of subsets of a set $T = \{t_1, \ldots, t_p\}$, one has to decide if there exists a set $W \subseteq T$ such that for each i, $1 \leq i \leq n$, the intersection of W with V_i contains exactly one element. Such a set W is called a *hitting set.* The problem was proved to be $\mathcal{NP}$-complete in [19].

We now show a polynomial-time algorithm that maps each instance of the hitting set problem into a corresponding set of FD's such that the following holds: If the hitting set problem has a positive solution then all schemas representing the set of FD's violate BCNF; if the hitting set problem has a negative solution then all schemas representing the set are in BCNF. This construction is therefore a polynomial reduction of the hitting set problem to each of our two problems. Since the construction is complex, we start by giving some intuitive explanation. Then we will present the full construction and a formal proof.

Suppose that the elements of T were attributes and that by a suitable choice of FD's all these attributes were contained in a single-relation scheme **R**. We could create a BCNF violation in **R** by taking a hitting set W and making some but not all attributes of **R** functionally dependent on W. However, we do not know if a hitting set really does exist and clearly we do not want to compute a hitting set since this would solve the hitting set problem. Thus we need a mechanism that will create a BCNF violation when a hitting set exists, but which does not require a particular hitting set to be known.

Let $x_1, \ldots, x_n$ be new attributes (the attribute x_i is, in a sense, a "representative" of the set V_i). We create the FD's $\{t \rightarrow x_i\text{: all } t \in V_i\}$. It follows that if W intersects each V_i and, in particular, if W is a hitting set, then $W \rightarrow x_1 \ldots x_n$. If we now make some but not all attributes of the relation scheme **R** functionally dependent on $x_1 \ldots x_n$, then we have the desired BCNF violation.

Before presenting the full construction we point to some difficulties that still remain. First, if W intersects each V_i but contains more than one element of some V_i then we also have $W \rightarrow x_1 \ldots x_n$. However, we do not want this dependency to create a BCNF violation in **R**. Therefore, we will have to introduce FD's that make any such set a superkey of **R**. Now, suppose W is a hitting set so $W \rightarrow x_1 \ldots x_n$. Let us look at an attribute of **R** that depends on $x_1 \ldots x_n$. If it is some $t \in W$ then $W \rightarrow t$ is not a BCNF violation. If it is some $t \in T - W$, then clearly $W \cup \{t\}$ contains at least two elements of some V_i and is a superkey of **R**. Since $W \rightarrow (W \cup \{t\})$, W is also a superkey of **R** and again we do not have a BCNF violation. Thus **R** must contain some additional attribute $Z \notin T$ such that $W \rightarrow x_1 \ldots x_n \rightarrow Z$ is a BCNF violation. We now present the construction.

Let $t_1t_1', t_2t_2', \ldots, t_jt_j'$ be a list of all pairs of elements of T that belong to some V_i, for any i. (Note that their number is a polynomial in p.) Our set of attributes contains the elements of T and the new attributes $x_1, \ldots, x_n$, y, z. We create a set of FD's over this set of attributes as follows:

(1) For each $t \in T$ and for each i such that $t \in V_i$, the set contains the FD $t \rightarrow x_i$.
(2) The set contains the FD $x_1 \ldots x_n \rightarrow z$.
(3) For each k, $1 \leq k \leq j$, the set contains the FD $t_kt_k' \rightarrow yz$.
(4) The set contains the FD $yz \rightarrow T$.

This set of FD's is easily seen to be a nonredundant covering of itself. Let us now study the relational schema that results when thee FD's are embodied by relation schemes.

The FD's of (1) will be embodied in a set of relation schemes, one relation scheme (with key t) for each $t \in T$. The FD of (2) will be embodied in a relation scheme with the set $x_1 \ldots x_n$ as a key. The FD's in (3) and (4) have equivalent left sides. Therefore, because of the restriction we placed on relational schemas, they will be embodied in a single-relation scheme **R** $(t_1, \ldots t_p, y, z)$. The keys of **R** are yz and the pairs t_kt_k', $1 \leq k \leq j$. The reader can easily verify that, with the exception of **R**, none of these relation schemes contain a BCNF violation. As for **R**, we observe that if $W \subseteq T$ intersects each V_i then $W \rightarrow x_1 \ldots x_n \rightarrow z$. However, if W contains a pair of elements of some V_i then W is a superkey of **R**. Thus **R** contains a BCNF violation if and only if T contains a subset which is a hitting set.

To conclude the proof of the theorem we now show that no matter what nonredundant covering we choose for the given set of FD's, we always obtain the schema just described. Thus a hitting set exists in T if and only if "all" schemas representing this set contain a BCNF violation.

Let us then look at an arbitrary nonredundant covering of the given set of FD's. We assume that all the left sides are reduced, that is, they do not contain extraneous attributes. We now list some facts about this covering and about the given FD's. The proof of each fact is either outlined briefly in parentheses or left to the reader.

Fact 1. Each one of the FD's of (1) and (2) belongs to every covering (apply Lemma 4).

Fact 2. Each one of the sets yz, $t_1t_1', \ldots, t_jt_j'$ functionally determines all other attributes; furthermore, any set that functionally determines all other attributes is a superset of one of these sets. (Compute closures.)

Fact 3. If the left side of an FD f in the covering does not determine all the attributes then f is one of the FD's in (1) and (2).

Fact 4. If the left side of an FD f does determine all other attributes then it is either yz or a pair t_kt_k'. (Follows from Fact 2 and the assumption that left sides are reduced.)

Fact 5. If the left side of an FD in the covering is yz then its right side is a member of t. (Since $yz \rightarrow T$ is in the closure, it is derivable from the given covering. Clearly no FD of the form $yz \rightarrow x_i$ is used in this derivation. It follows from Fact 1 that $yz \rightarrow x_i$ is redundant.)

Fact 6. If the left side of an FD in the covering is a pair t_kt_k' then its right side is either y or z or a member of T. (Similar argument.)

Fact 7. If the right side of an FD in the covering is a member of T then its left side is either yz or a pair t_kt_k' for some k. (Except for the FD's of (1) and (2), all FD's in the covering have yz or some t_kt_k' in their left side.)

Fact 8. The covering includes at least one FD with right side t for each $t \in T$.

Fact 9. The covering does not include any FD's except those mentioned in Facts 1-9.

It follows easily from these facts that for any covering of the given set of FD's the schema that embodies it is the one described above. It is also easy to verify that the construction can be done in polynomial time. The theorem follows. □

PROOF: Corollary 3. The problem is obviously in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce it to the hitting set problem. For any given instance of the hitting set problem we construct the schema described in the proof of the theorem. This schema is in 3NF and it is also the schema produced by applying Bernstein's algorithm to the given FD's. Since the hitting set problem has a positive solution if and only if the schema is not in BCNF, the corollary follows. □

PROOF: Theorem 5. The problem is obviously $\mathcal{NP}$-computable. Given **R**, we nondeterministically choose a subset of its attributes and check that it is a key (using, say, the membership algorithm). To show the problem is $\mathcal{NP}$-hard, we again use a reduction of the hitting set problem. The construction is the same as that used for the BCNF violation (Theorem 4). Now, if W is a solution to the hitting set problem, then $W \rightarrow X_1 \ldots X_n \rightarrow Z$, so WY is an additional key. Conversely, if W is an additional key, then it cannot contain any pair of elements of any V_i (because all such pairs are already known to be keys). It therefore must use $X_1, \ldots, X_n \rightarrow Z$ to derive all attributes, so it is a solution to the hitting set problem. □

REFERENCES

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. Armstrong, W.W. Dependency structures of data base relationships. Information Processing 74, North-Holland Pub. Co., Amsterdam, 1974, pp. 580-583.
3. Beeri, C. On the membership problem for multivalued dependencies in relational databases. TR-229, Dept. EE and Comptr. Sci., Princeton U., Princeton, N.J., Sept. 1977.
4. Beeri, C., Bernstein, P.A., and Goodman, N. A sophisticate's introduction to database normalization theory. Proc. Int. Conf. on Very Large Data Bases, Berlin, 1978, pp. 113-124. (available from ACM, New York)
5. Beeri, C., Fagin, R., and Howard, J.H. A complete axiomization for functional and multivalued dependencies in database relations. Proc. ACM SIGMOD. Int. Conf. on Manage. of Data, Toronto, Canada, 1977, pp. 47-61.
6. Bernstein, P.A. Normalization and functional dependencies in the relational data base model. Tech. Rep. CSRG-60, Ph.D. Diss., Comptr. Syst. Res. Group, Dept. Comptr. Sci., U. of Toronto, Toronto, Canada, Nov. 1975.
7. Bernstein, P.A. Comment on 'Segment synthesis in logical data base design'. *IBM J. Res. and Develop. 20,* 4 (July 1976), 412.
8. Bernstein, P.A. Synthesizing third normal form relations from functional dependencies. *ACM Trans. Database Syst. 1,* 4 (Dec. 1976), 272-298.
9. Bernstein, P.A., and Beeri, C. An algorithmic approach to normalization of relational data base schemas. Tech. Rep. CSRG-73, Comptr. Syst. Res. Group, Dept. Comptr. Sci., U. of Toronto, Toronto, Canada, Sept. 1976.
10. Bernstein, P.A., Swenson, J.R., and Tzichritzis, D.C. A unified approach to functional dependencies and relations. Proc. ACM SIGMOD Conf. on Manage. of Data, San Jose, Calif., 1975, pp. 237-245.
11. Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM 13,* 6 (June 1970), 377-387.
12. Codd, E.F. Further normalization of the data base relational model. In *Data Base Systems,* Courant Inst. Comptr. Sci. Symp. 6, R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 33-64.
13. Codd, E.F. Recent investigations in relational database systems. Information Processing 74, North-Holland Pub. Co., Amsterdam, 1974, pp. 1017-1021.
14. Date, C.J. *An Introduction to Database Systems.* Addison-Wesley, Reading, Mass., 1977.
15. Delobel, C., and Casey, R.G. Decomposition of a data base and the theory of Boolean switching functions. *IBM J. Res. and Develop. 17,* 5 (Sept. 1972), 374-386.
16. Delobel, C., Casey, R.G., and Bernstein, P.A. Comment on 'Decomposition of a data base and the theory of Boolean switching functions'. *IBM J. Res. and Develop. 21,* 5 (Sept. 1977), 484-485.
17. Fagin, R. Multivalued dependencies and a new normal form for relational databases. *ACM Trans. Database Syst. 2,* 3 (Sept. 1977), 262-278.
18. Fagin, R. Functional dependencies in a relational database and propositional logic. *IBM J. Res. and Develop. 21,* 6 (Nov. 1977), 534-544.
19. Fagin, R. The decomposition versus synthetic approach to relational database design. Proc.

Third Int. Conf. on Very Large Data Bases, Tokyo, Oct. 1977, pp. 441–446. (available from ACM, New York)

20. HOPCROFT, J.E., AND ULLMAN, J.D. *Formal Languages and Their Relation to Automata.* Addison-Wesley, Reading, Mass., 1969.
21. KARP, R.M. Reducibility among combinatorial problems. In *Complexity of Computer Computations,* Plenum Press, New York, 1972, pp. 85–104.
22. LUCCHESI, C.L., AND OSBORNE, S.L. Candidate keys for relations. U. of Waterloo, Waterloo, Ont., Canada, 1976. To appear in *SIAM J. Comptng.*
23. PARADAENS, J. About functional dependencies in a data base structure and their coverings. Rep. R342, M.B.L.E. Res. Lab., Brussels, March 1977.
24. RISSANEN, J. Independent components of relations. *ACM Trans. Database Syst. 2,* 4 (Dec. 1977), 317–325.
25. RISSANEN, J., AND DELOBEL, C. Decomposition of files, a basis for data storage and retrieval. Res. Rep. RJ1220, IBM Res. Lab., San Jose, Calif., May 1975, pp. 211–223.
26. SCHMID, H.A., AND SWENSON, J.R. On the semantics of the relational data model. Proc. ACM SIGMOD Conf. on Manage. of Data, San Jose, Calif., May 1975, pp. 211–223.
27. WANG, C.P., AND WEDEKIND, H.H. Segment synthesis in logical data base design. *IBM J. Res. and Develop. 19,* 1 (Jan. 1975), 71–77.

Received September 1977; revised May 1978