NORMAL FORMS FOR RELATIONAL DATA BASES

Sylvia Louise Osborn

Research Report CS-78-06

Department of Computer Science

University of Waterloo
Waterloo, Ontario, Canada

January 1978

Normal Forms for Relational Data Bases

by

Sylvia Osborn

Abstract

Several third normal forms for relational data bases have been proposed, notably those by Codd and Kent. Related research has yielded algorithms for finding coverings of the functional dependencies of a relational data base. Since normal forms and coverings are intended to remove anomalies, each is examined to determine its relative effectiveness in doing so. It is then shown that under certain conditions a relation can be reconstructed from the collection of relations constituting its third normal form.

The concept of a key is central to normal forms. Thus an algorithm is presented which finds $\underline{K}$, the set of all keys for any subset of the given set of attribute names, A, and for a given set F of functional dependencies, in time polynomial in $|A|$, $|F|$ and $|\underline{K}|$. For a single relation, attainable upper bounds on the number of minimal keys are given, in terms of $|F|$ and $|A|$. It is shown that the problem of deciding whether or not a specified attribute name is prime (i.e. an element of some minimal key) is NP-complete. In order to prove this result, it is first shown that the problem of deciding whether or not there is a key having at most m elements, for a given parameter m, is NP-complete.

Neither coverings nor normal forms alone are found to be as effective in removing anomalies as the two together. An algorithm is presented which, given any relation, finds a Codd third normal form that covers the functional dependencies, in time polynomial in $|A|$, $|F|$ and the number of keys for each relation. Possible choices for primary keys are outlined. We suggest algorithms for deciding, based on the collection of relations output by the Codd third normal form algorithm, which joins to perform in answering queries involving an arbitrary subset of A. Finally, an algorithm is given to determine whether a covering Kent third normal form exists.

Acknowledgements

I am deeply indebted to Prof. Frank Tompa for his many words of encouragement throughout the work on this thesis. I would also like to thank Prof. Ian Munro for his help during the final stages. The financial assistance of the National Research Council of Canada is gratefully acknowledged.

Table of Contents

List of Figures

Notation

A, B, C, etc.	sets of attribute names
a, b, c, etc.	individual attribute names
$\underline{K}$, $\underline{C}$, etc.	sets of sets of attribute names
(1), (i), (j), etc.	subscripts
2**A	power set of A
$\subseteq$	subset of
$\not\subseteq$	not a subset of
$\in$	member of
$\notin$	not a member of
$\cup$	union
$\cup_{i\geq 1}$	union over all $i \geq 1$
$\cap$	intersection
$\mathcal{F}$	script F
0	order
ϕ	null set

1. Introduction

The relational model for data was suggested by Codd several years ago [Codd 70]. He showed that a conceptually simple model for data, one in which all data is represented by relations, is adequate to describe complex applications for data base management purposes. If the simple notion of functional dependency is added to the basic model, some problems arising on insertion, update and deletion, which seem to deal with the semantics of the data, can be handled syntactically. To deal with these problems, Codd suggested that a relation be decomposed into a collection of relations in what he called a third normal form; thus third normal form is intended to be a form in which none of the problems mentioned above can arise.

Since Codd's original paper, there has been a great deal of related research, including other definitions of third normal form, various ways to carry out decomposition, and the related notion of covering the functional dependencies. With the notable exception of Bernstein and Beeri [Bernstein 75a, 76b, 76c], there has been a lack of emphasis on finding efficient algorithms, or analysing the run time of any algorithms given.

Thus this thesis has three objectives:

- to clarify the theoretical aspects of normalizing a relational data base,

- to examine the feasibility of efficient algorithms for related problems, and
- to provide efficient algorithms wherever possible.

We begin, in Chapter 2, by examining in detail the need for third normal form and by establishing the terminology to be used throughout the thesis.

In Chapter 3, some properties relating to third normal form are examined. Three distinct types of third normal forms are identified, and the relationships among them established. It is shown that covering the functional dependencies, often proposed as a way of achieving third normal form, is not equivalent to third normal form. In fact, both properties are desired in a collection of relations if the insertion, update and deletion problems are to be avoided. One type of third normal form, as proposed by Kent, is shown to be successful in solving these problems; however, a Kent third normal form that also covers the functional dependencies does not always exist. One set of conditions is given under which the original relation can be reconstructed from a third normal form collection.

Chapter 4 is devoted to various aspects of finding keys for relations. Attainable upper bounds on the number of keys a single relation can have are given; this work was done jointly with F. W. Tompa. Two related problems are shown to be NP-complete. Finally an algorithm is given which, for a given set A of attribute names and a given set

F of functional dependencies, finds all of the keys for an arbitrary subset of A, in time polynomial in |A|, |F| and the number of keys actually found. The two NP-complete results and an earlier version of the key finding algorithm which finds all the keys for A were done jointly with C. L. Lucchesi [Lucchesi 77].

Since covering Kent third normal forms do not always exist, covering Codd third normal forms are studied first. In Chapter 5, an algorithm is developed which finds a covering Codd third normal form collection in time polynomial in |A|, |F| and the number of keys found for each relation in the resulting collection. The relations differ from those found by Bernstein's algorithm in that all keys for all relations are considered, not just those that happen to appear on the left-hand side of a given functional dependency. The collection can have at most one more relation than output by Bernstein's algorithm, or up to n fewer relations for an arbitrary value of n. The reason for having possibly one more relation is that our algorithm guarantees that the original relation is reconstructible, whereas Bernstein's does not. We then discuss how to choose primary keys for the resulting relations. We also suggest algorithms for using the colllection of relations output to decide mechanically which joins to perform to answer an arbitrary query on the data base.

The problem of finding Kent third normal forms is

studied in Chapter 6. It is shown through examples that determining an optimum covering Kent third normal form in an efficient way is probably not feasible. We do show, however, a method of deciding whether or not a covering Kent third normal form exists, which has not been done before.

Chapter 2

Background

2.1 Motivation

The purpose of a data base is to store information for subsequent inquiry. We assume that all data needed to answer any one query can be considered (initially) to be in a single table.

The totality of information in this one table can be considered to represent an enterprise [ANSI/SPARC]. Throughout this thesis, the arguments given and the algorithms presented pertain to designing a data base for this enterprise. One area of research is to consider how to break up this single table to make expected query processing efficient. However, we will assume that information characterizing the queries of the particular enterprise is not yet available. Thus we are concerned strictly with modelling the data and ensuring that query-independent operations such as insertion, deletion and update of pieces of information can be carried out efficiently.

To see that these operations are not necessarily straightforward, consider a sample table with column headings {Professor, Department, Course number, Room, Time, Student name, Address, Mark}. As various pieces of information become available, values may be inserted into those columns for which data is available, and each unknown value can be represented by a special symbol (e.g., "?") meaning

"unknown", as in Figure 2.1. Each row in the table represents an entity; each column can be thought of as an attribute.

Figure 2.1

Prof	Dept	Course	Room	Time	Student	Address	Mark
Smith	C.S.	395	29	9:30	?	?	?
Taylor	C.S.	395	61	10:30	?	?	?
Jonson	C.S.	405	14	12:30	?	?	?
Jones	C.S.	?	?	?	?	?	?

Suppose we now want to add a student's name, course and time to the table, where the time is 10:30 and the course number is 395. It is not clear whether this is the course taught by Taylor or whether it is another copy of 395 offered by someone else, for example Jones. Thus we do not know whether to update the "unknowns" in Taylor's or Jones's row or to add a new row in which the professor is unknown.

There are several ways to avoid this problem:

1. Break the table into sets of columns such that information would always arrive in pieces that correspond to one of these column sets. In other words, these column sets would represent insertion or update units.
2. Only allow data to be entered if all attributes have known values.

Solution 2 does not work very well as it means that

none of the information in Figure 2.1 could have been entered at all. Such inability to insert partial information has been called an insertion dependency or anomaly [Codd 71a,Codd 71b]. The reverse situation, i.e. being forced to delete known information, could also happen. For example, if everyone dropped a particular course, all rows relating to that course would have to be deleted thus causing information about the professor's department to be lost from the table. This has been called a deletion dependency or anomaly [Codd 71a,Codd 71b]. Thus one motivation for breaking up the table is to provide reasonable update units; that is, to be able to handle partial information.

In addition to the ability to add or delete some information, the work which must be done to insert, update or delete a row of the table is also a criterion to be considered. One might reasonably expect that this could be done without having to look at other rows in the table. Such might be the case if we made no attempt to maintain the integrity of the data. For example, a professor can only be in one room at one time; thus every row in the table pertaining to Taylor at 10:30 should contain room 61. There will be one such row for every student in the course. As a result, every new student added to Taylor's class would cause a check to avoid possible input errors. This illustrates another type of insertion anomaly.

The repetition of room number for each professor is clearly redundant. If a class is moved to another room, many rows will have to be changed to update what seems to be a single piece of information. This is an example of an update anomaly [Codd 71a,Codd 71b]. If the original table were broken up so that professor, room and time appeared in one table and students and their courses in another, then only one row would be updated if a room were changed.

What we seem to need, then, is some external information which indicates update units and also shows which situations should be checked for integrity. This external information is formalized in the next section by the concept of functional dependency. Functional dependency can, then, be used to decide how to break up the original table to avoid the problems described above.

Once the table has been split, we must still be able to satisfy the original purpose for storing the data, namely to answer queries. It is important that the original table can always be reconstructed so that queries posed in terms of it can still be answered.

2.2 Basic definitions

Let A = {a(1), . . . , a(n)} be a finite set of attribute names. Associated with each a(i) is a set of valid values which, together with the special symbol, ?,

which represents "unknown", is called a domain and written D(i). It will be assumed that domain values are indivisible, that is, that all domains are simple [Codd 70]. It should be noted that although the a(i) are all distinct, the D(i) need not be.

A relation R is a subset of D(1) x . . . x D(n) [Codd 70].[1] (Note we have defined R as a set, and thus there are no duplicate elements.) The table in Figure 2.1 is an example of a relation and the column headings, of attribute names. Professor and student name could have the same underlying domain, namely the set of all people's names.

In order to break R into a set of relations avoiding the problems of the previous section, we require information on the relationships among the attributes. We will consider only functional relationships and see that they are sufficient to identify the types of redundant information exemplified above and that they provide a method of specifying update units. Another type of redundancy, namely multivalued dependencies, [Beeri 77, Fagin 76b, Zanoilo 76] will not be considered.

In the timetable example, we have observed that each professor can be in only one room at a time. That is, we are modelling a situation in which, for each pair <professor, time>, there can be at most one value for room, so our tabulation should accurately reflect this. Each such

---

[1] Since all domains are simple, R is said to be in first normal form [Codd 71a, Codd 71b].

relationship among values is called a functional dependency. Functional dependencies may, of course, arise not only from physical limitations, but also because of rules established by the enterprise; e.g., a professor can be a member of only one department within the University.

To formalize this notion, let F be a user-specified binary relation on the power set of A (denoted 2**A). Each member of this relation corresponds to a functional dependency. One functional dependency in F will be denoted by X -F-> Y, where X and Y represent subsets of A. Where there is no possibility of confusion, simply X --> Y will be used. For example, one functional dependency for the timetable relation might be {professor, time} --> {room}, or pt --> r. (Single letter codes and sample F for the timetable relation are given in Figure 2.2.)

A functional dependency X --> Y means that, according to the user's model of the enterprise, at any one time there will be at most one value for Y associated with each value for X in the relation. It would be very tedious for the user to have to specify all of the members of 2**A x 2**A which agree with this model, as some of them are rather trivial, e.g. dependencies such as X --> Y where Y $\subseteq$ X are part of every model. Let us denote the complete set of functional dependencies constituting a model by $\mathcal{F}$. Armstrong has called this the dependency structure of a relation R [Armstrong 74]. That is, $\mathcal{F}$ contains only those

Figure 2.2

A for timetable relation

Single letter code	Attribute name
p	Professor
d	Department
c	Course
r	Room
t	Time
s	Student's name
m	Mark
u	Student's address

F for timetable relation

p --> d
pt --> r
sc --> m
s --> u
pt --> c
c --> d
sc --> r
sc --> p
sc --> t

functional dependencies which are guaranteed to be observable in the data at all times. This is the model that the user intends to describe in specifying a set F, assuming F is correct and contains enough information to derive $\mathcal{F}$.

Given a set of functional dependencies F, where F is not necessarily equal to $\mathcal{F}$, we define $\underline{F}(A, F)$, the projective, transitive and additive closure of (A, F), (or simply the closure of (A, F)), as follows:

$\underline{F}(A, F) = U\ i \geq 1\ F(i)$

where F(1) is defined to be the union of F and all pairs

defined by projectivity, i.e. for all subsets X and Y of A, if Y $\subseteq$ X then X -F(1)-> Y;

and the F(i), i≥2, are defined to be exactly those functional dependencies obtained by the following:

- transitivity: for all subsets X, Y and Z of A, if X -F(i-1)-> Y and Y -F(i-1)-> Z then X -F(i)-> Z;
- additivity: for all subsets X, Y and Z of A, if X -F(i-1)-> Y and X -F(i-1)-> Z then X -F(i)-> YZ.[1]

Note that for all i, F(i-1) $\subseteq$ F(i).

The reader should be able to verify that the functional dependencies in the closure are consistent with the user's model; in other words $\underline{F}(A, F) \subseteq \mathcal{F}$. Armstrong has shown that the axioms used to define $\underline{F}(A, F)$ are sufficient to characterize all of $\mathcal{F}$, i.e. that $\underline{F}(A, F) = \mathcal{F}$ [Armstrong 74]. Fagin has verified the completeness of these closure axioms for deriving all of $\mathcal{F}$ by showing the equivalence of functional dependency statements and implications in propositional calculus [Fagin 76a]. This means that the user need not specify all of $\mathcal{F}$, but may specify some F whose closure is equal to $\mathcal{F}$.

We will refer to the pair (A, F) as a relational description. Because of additivity, we can assume a cannonical form for functional dependencies in F, namely that all right-hand sides in F contain at most one attribute name. This will standardize the input to the algorithms

---

[1] YZ is a shorthand notation for Y U Z.

presented later, and thus also standardize the analysis of these algorithms.

A subset K of A is a *key* for (A, F) if K --> A is in $\underline{F}$(A, F). Intuitively, a key K is a set of attribute names which, according to the model, or according to $\underline{F}$(A, F), have the property that, given a set of values for the attribute names in K, there can be at most one row in R with these values, since K --> A. K is a *minimal key* for $\underline{F}$(A, F) if K --> A lies in $\underline{F}$(A, F) but no proper subset of K also has this property. (Minimal keys are called candidate keys in the literature [Codd 71a]). For the timetable example, sc and spt are the minimal keys.

An attribute name is *prime* relative to (A, F) if it lies in some minimal key for (A, F) [Codd 71a]. In the example, s, c, p and t are the prime attribute names and the rest of the attributes, d, r, m and u are said to be *nonprime*.

The necessity of maintaining redundant information can arise for two reasons. The first is exemplified by the student's address attribute in the timetable example. Each row in the table containing the student's name must have the same address since s --> u is in F. However, there is nothing in the model to prevent the student from taking more than one course. Each sc combination can appear only once in the table, since sc is a minimal key. However, s is only part of this key and therefore each s (and the corresponding

u value) can appear more than once.

This situation is formalized by the following definition. For subsets X and Y of A, Y is said to be fully dependent on X in $\underline{F}(A, F)$ if X --> Y is in $\underline{F}(A, F)$ but for no X', a proper subset of X, does X' --> Y lie in $\underline{F}(A, F)$ [Codd 71a]. In the example, then, the fact that u is not fully dependent on sc results in redundancy.

The second type of redundancy arises in situations analogous to the following problem with the department attribute in the timetable example. F contains the two dependencies sc --> p and p --> d. A given professor can appear in more than one entry in the table but the department associated with each such occurrence must be the same.

To formalize this notion, let X and Y be distinct subsets of A, and let z be an element of A. If X --> Y and Y --> z lie in $\underline{F}(A, F)$, Y $\not\rightarrow$ X in $\underline{F}(A, F)$, and z $\notin$ Y, then z is said to be transitively dependent on X in $\underline{F}(A, F)$ [Codd 71a]. Thus in the example, d is transitively dependent on sc. Note that z $\notin$ Y is not in Codd's original definition but is consistent with what is intended; i.e. there is no redundancy involved in c --> ab --> b, whereas there is in c --> ab -->d.

The two definitions above are stated in terms of the closure because F may or may not contain X --> Y and Y --> z in terms of the definition. Consider, for example, this set of functional dependencies:

a --> b

b --> c

c --> d

b --> d

The two dependencies a --> c and b --> d are in the closure of this set of functional dependencies and therefore d should be regarded as transitively dependent on attribute name a whether or not a --> c or b --> d is mentioned in F.

With these preliminary definitions, we are now able to start examining various (third) normal form definitions. Recall that Codd's first normal form requires that all domains be simple (i.e. contain indivisible values and not sets) [Codd 70, 71a, 71b]. His second normal form, containing only pcint 1 below, is only partially free of anomalies [Codd 71a, 71b]. Codd's third normal form is intended to describe the properties of a single relation that has none of the anomalies outlined above.

A relational description (A, F) is in third normal form if

1. every nonprime attribute name of A is fully dependent on each minimal key relative to (A, F),
2. no nonprime attribute name of A is transitively dependent on any minimal key relative to (A, F) [Codd 71a, 71b].

In the definition of fully dependent, an arbitrary subset X of A is mentioned. In the definition of Codd third

normal form, full depencencies are considered only in the case that X is a minimal key. Thus a non-full dependency violating Codd third normal form is characterized by the following: there is a minimal key K, a proper subset K' of K, (since K is minimal, K' $\not\rightarrow$ K), there is an attribute name z such that K' --> z, and since z is nonprime, z $\notin$ K'. In this light, a non-full dependency on a minimal key is just a special case of a transitive dependency on a minimal key. Thus we can restate the definition of Codd third normal form as: a relational description (A, F) is in Codd third normal form if no nonprime attribute name is transitively dependent on any minimal key relative to (A, F).

Kent has given a slightly different definition of third normal form which removes the emphasis on nonprime attributes [Kent 73]. The arguments that allowed us to remove condition 1 for the Codd definition also apply here, giving us a shorter definition than Kent's original one. A relational description (A, F) is in Kent third normal form if no attribute name in the complement of a minimal key is transitively dependent on that minimal key. In the next chapter we will see the consequences of strengthening the restrictions.

Codd gives another normal form definition which is equivalent to Kent third normal form [Codd 74]. A relational description (A, F) is in Boyce-Codd third normal form if, for every subset C of A, if any attribute name not

in C is functionally dependent on C in $\underline{F}(A, F)$, then all attribute names in A are functionally dependent on C in $\underline{F}(A, F)$. Observe that this can be rephrased as: if any attribute name not in C is functionally dependent on C in $\underline{F}(A, F)$, then C is a key relative to $\underline{F}(A, F)$.

The intent of these definitions is to break the set A into (possibly intersecting) subsets, {A(1),. . . ,A(m)}, so that each of the resulting relational descriptions obeys one or more of the above definitions. The functional dependencies associated with each A(i) in this collection would be all elements of $\underline{F}(A, F)$ whose left and right sides contain only attribute names from A(i).

Two operations have been defined for breaking up the relations into the relations R(i) that correspond to the A(i), and for reconstructing R.

Let r be a tuple of relation R and let a(i) be one of the attribute names of R. Then r.a(i) denotes the component of tuple r corresponding to attribute name a(i). This is extended to a set of attribute names B = {b(1), ..., b(k)} by r.B = (r.b(1), ..., r.b(k)). The projection of R onto the attribute name set $B \subseteq A$ is given by

$$\underline{P}(B)(R) = \{r.B \mid r \in R\}.$$ [Codd 71a]

Let R and S be two relations where R has attribute names $A \cup B$, S has attribute names $B \cup C$, A = {a(1), ..., a(n)}, B = {b(1), ..., b(m)} and C = {c(1), ..., c(k)}, and A, B and C are disjoint. That is, all common attribute

names are in B. Then the natural join of R and S is defined by R * S = {(r.A, r.B, s.C) | r ∈ R, s ∈ S and r.b(i) = s.b(i) for all i = 1,m} [Codd 71a].

Note that since our algorithms are intended to be used at the design stage, before any data is entered, the natural join is the more important operation as it will be used to reconstruct the original table for queries. Data will be entered into the appropriate R(i) output by the algorithms, not into R to be projected into the R(i).

One approach to resolving the anomaly problems has been taken by a number of people [Delobel 72, 73, Rissanen 73, Wang 75, Bernstein 75a]. It involves the notion of a covering: a set of functional dependencies over A, F', is called a covering of F if its closure is equal to F(A, F). This approach involves finding a covering for F, and then breaking A into collections {A(1), ..., A(m)} so that each A(i) consists of all attribute names in one or more functional dependencies in the covering.

Coverings can have one or both of the following properties:

- F' is a minimal covering if there is no proper subset G of F' such that G is also a covering.
- F' is a minimum covering if there is no covering H such that H has smaller cardinality than F'.

(Observe that a minimum covering is also minimal, but not necessarily the converse.)

Coverings can have other characteristics. One can insist, for example, that F' be a subset of F. This, together with minimality, would reflect a desire that the functional dependencies embodied in the final collection, which determine the attribute name sets that can be used as "update units", be a nonredundant subset of the ones originally specified by the designer.

Optimization has been introduced for covering by the notions of minimum and minimal coverings. For third normal forms, Codd originally proposed that an optimum collection of third normal form relations be maintained [Codd 71a]. A set of relations is in optimum Codd third normal form if every relation in the collection is in Codd third normal form and the collection contains the smallest number of relations satisfying this property. This is not the only optimality criterion one might want to consider. If there were a choice between several third normal form collections, and if query patterns were known, there might be one collection for which query processing would be faster, either because fewer joins were required, or because the joins took less time (e.g. the range of values present in the data for some attribute names might be much smaller than for others). Since we are analysing this at the design stage, however, the only optimality criterion we can address is the number of relations in the final collection.

with some knowledge of the functional dependencies pertaining to each relation. For these reasons, we prefer to classify attempts as either a covering approach or a third normal form approach, depending on their immediate objectives.

Most of the work has been in the covering area, attempting to find a covering of the functional dependencies and then to examine whether or not the resulting collection is also in third normal form. The first major contribution was made by Delobel and Casey who use an equivalence between functional dependency statements and Boolean functions to find minimal covers [Delobel 73b]. They state that each relation (corresponding to a single functional dependency in the cover) is in Codd third normal form. They do not optimize the number of relations generated. The algorithm, relying on techniques for finding prime implicants for Boolean functions, is not very efficient.

Wang and Wedekind find a covering using a more straightforward approach, originally suggested by Delobel [Delobel 72], which removes redundant functional dependencies from a set until no more can be removed while maintaining the same closure [Wang 75]. They then suggest optimizing the number of relations by merging functional dependencies in the covering whose left-hand sides are identical. This type of optimization is shown to guarantee that the resulting relations are in Codd third normal form.

However, the covering algorithm used generates all coverings, and so is rather inefficient.

Bernstein provides a better optimization by merging functional dependencies whose left-hand sides are equivalent in terms of what attributes are functionally dependent on them [Bernstein 76b]. However, this type of covering does not always produce Codd third normal forms, and therefore an additional step is required to remove some attributes from the resulting relations. Bernstein's algorithm is, in comparison with previous work, very efficient, requiring only $O(|A|^2 |F|^2)$ steps for a given (A, F). The algorithms given in Chapter 5 differ from Bernstein's in that they find third normal form and covering at the same time. Furthermore, they take into account all minimal keys for all relations, not just those that happen to appear on the left-hand side of a functional dependency. In general this can take a lot more time, because, as we will see in Chapter 4, the number of keys can be very large. On the other hand, in some cases, it produces fewer relations than Bernstein's algorithm.

The only related work that does not consider first finding a covering is that of Rissanen and Delobel [Rissanen 73]. They give a method for finding all decompositions of a relation and suggest that the choice among them be based on optimizing some performance criterion. Of particular interest are decompositions into irreducible relations

which, as we will see in Chapter 3, is a stronger requirement than third normal form.

Although all of these algorithms have produced Codd third normal form relations, we will see in Chapter 3 that Kent third normal forms are more successful in eliminating anomalies. However, a collection of relations in Kent third normal form that covers the functional dependencies does not always exist. No one has previously given either a test for existence of covering Kent third normal forms or an algorithm for finding them when they do exist. Both are given in Chapter 6.

Chapter 3

Normal forms and coverings

3.1 Comparison of third normal form definitions

Many anomaly-removing normal forms, which we loosly call third normal forms, have been presented in the literature. In this section we will show that three distinct third normal forms have been defined and examine the relationships among them. We will also see how a similar notion, irreducible relations, compares with third normal forms.

In section 2.2, definitions were given for Codd, Kent and Boyce-Codd third normal forms. We will also consider here definitions given in [Heath 71], [Delobel 73a], [Date 75] and [Boyce 73].

The definition of third normal form in [Heath 71], with an appropriate change in terminology, is almost identical to (our version of) Kent's. A similar definition is also found in [Delobel 73a]. The equivalence between Kent and Boyce-Codd third normal forms, often assumed in the literature [e.g. Codd 74, Date 77], is not as obvious however; therefore it will be shown more rigorously.

Lemma 3.1: A relational description (A, F) is in Kent third normal form if and only if it is in Boyce-Codd third normal form.

proof: Suppose (A, F) is in Kent third normal form but not in Boyce-Codd third normal form; i.e. there is a proper

subset C of A, an attribute name d not in C such that C is not a key, and such that C --> d is in $\underline{F}(A, F)$. (A, F) has at least one minimal key, say K. By these assumptions, K and C are distinct and C and d are distinct, but K and d are not necessarily distinct. Consider then K U C - d, which is a key but not necessarily minimal; i.e. K U C - d contains at least one minimal key, say K', and K' and d are distinct. We now have K' --> C, C $\not\rightarrow$ K', d $\in$ K', and d $\notin$ C. That is, d is in the complement of minimal key K' and it is transitively dependent on K', contradicting (A, F) being in Kent third normal form.

To prove the converse, suppose (A, F) is in Boyce-Codd third normal form but not in Kent third normal form, i.e. there is a minimal key K, and some attribute name b not in K such that b is transitively dependent on K. That is, there exists a set C of attribute names such that K --> C, C $\not\rightarrow$ K, C --> b, b $\notin$ C, b $\notin$ K. But if (A, F) is in Boyce-Codd third normal form, then C --> b and b $\notin$ C implies that C must be a key; thus C --> K should be in $\underline{F}(A, F)$, contradicting the assumption. □

The term <u>primary key</u> is used to denote one minimal key which is chosen to act as an index for a relation and therefore, in practice, is not be allowed to have unknown values. The last two definitions we will discuss emphasize the role of the primary key and make no requirements of the other minimal keys of a relational description.

A relational description (A, F) is in Boyce third normal form if, for primary key K, if any attribute name not in K is functionally dependent on K, then K --> A is in F(A, F)[1] [Boyce 73].

A relational description (A, F) is in Date third normal form if no attribute name in the complement of the primary key K is transitively dependent on K in F(A, F) [Date 75].[2]

The proof that these two definitions are equivalent is similar to the Kent, Boyce-Codd equivalence; therefore we have:

Lemma 3.2: Date third normal form and Boyce third normal form are equivalent.

We have thus identified three sets of equivalent definitions. We will now show how they are related to one another, using Codd, Kent and Date as representatives of the three groups.

It is recognised [Bernstein 75a, Codd 74, Date 77, Kent 73] that Kent third normal form is strictly stronger than Codd third normal form. However the relationship of Date and Boyce third normal forms to each other and to the other two has not been examined.

Lemma 3.3: Kent third normal form implies Date third normal form.

---

[1] This definition is not only reworded to conform to our terminology, it is renamed--Boyce calls it fourth normal form!

[2] Recall that the non-full dependency which would be the first condition in this definition is a special case of transitive dependency.

proof: If, for every minimal key X, there is no transitive dependency of attribute names in the complement of X on X, then this property clearly holds for one particular minimal key, namely the primary key. □

Lemma 3.4: Date third normal form does not imply Kent third normal form.

proof: Consider this example: let A = {a, b, c, d, e} and let F contain:

ab --> c

bce --> a

bce --> d

The minimal keys are abe and bce. If the primary key is chosen to be bce, then the description is in Date third normal form but not in Kent third normal form. □

Lemma 3.5: Date third normal form implies Codd third normal form.

proof: Suppose (A, F) is in Date third normal form but not in Codd third normal form; i.e. suppose K is a minimal key not equal to the primary key P, that H is a set of attribute names, j is an individual attribute name and that K --> H, H --> j, j $\notin$ H and H $\not\rightarrow$ K in $\underline{F}$(A, F). Since H $\not\rightarrow$ K, H is not a key. Therefore H $\not\rightarrow$ P. Therefore j is transitively dependent on P, contradicting the assumption that (A, F) is in Date third normal form. □

Corollary: Kent third normal form implies Codd third normal form.

Lemma 3.6: Codd third normal form does not imply Date third normal form.

proof: Using the example of Lemma 3.4, let the primary key be abe. Then (A, F) is in Codd third normal form but it is not in Date third normal form. □

Corollary Codd third normal form does not imply Kent third normal form.

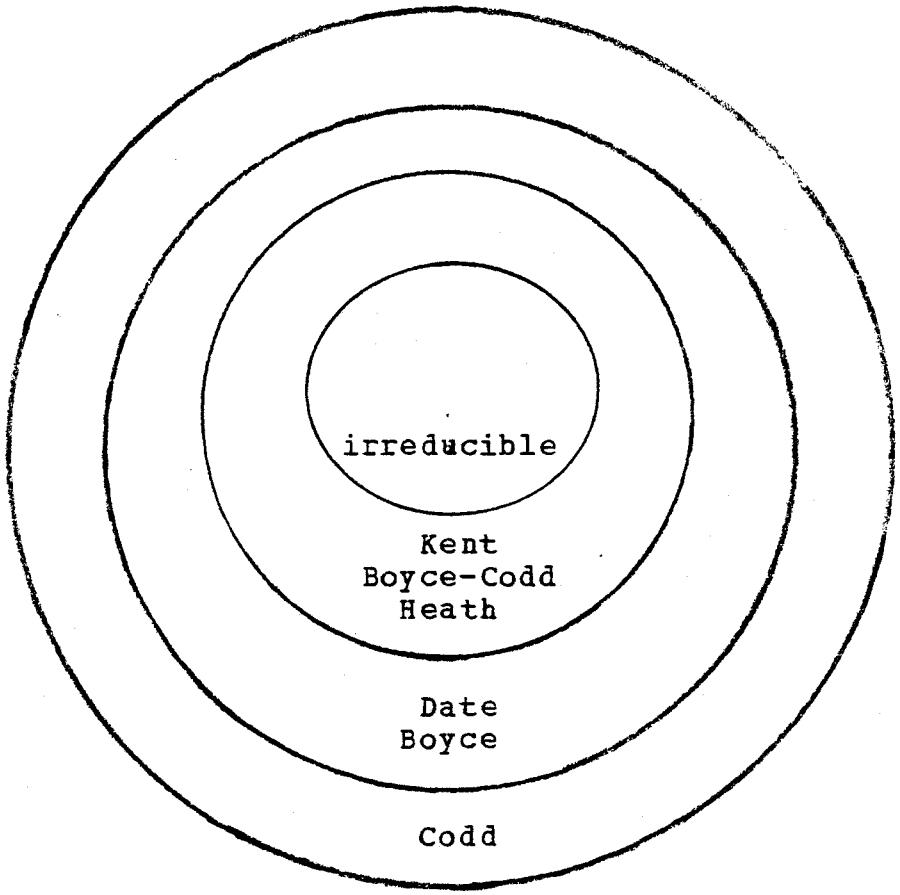
To complete this discussion we shall briefly consider another property of relations discussed in the literature. A relation R is said to be irreducible if there is no projection of R onto attribute name sets B and C such that both B and C are proper subsets of A and R = P(B)(R) * P(C)(R) (R is the natural join of these projections) [Rissanen 73]. This notion is similar to that of atomic relation [Zaniolo 76]. It is not immediately clear what relationship this property has with third normal forms, since it is not defined in terms of functional dependencies. However Rissanen and Delobel show that if a relation is irreducible, then every non-projective functional dependency is of the form A - {a(i)} --> B [Rissanen 73]. By putting these into canonical form for functional dependencies, if necessary, |B| = 0 or 1, i.e. B is either the null set or equals {a(i)}. From this it follows that irreducibility implies Kent third normal form. The converse is not true however. The relational system with A = {a, b, c} and F = {a --> b, b --> a, a --> c} is in Kent third normal form but

it is not irreducible since R is the natural join of two relations with attribute names {a, b} and {a, c}.

Corollary: Neither Codd third normal form nor Date third normal form implies irreducibility.

Figure 3.1 summarizes the results of this section. We see that of the third normal form definitions, Kent's

Figure 3.1



definition is the strongest and Codd's, the weakest. In the remainder of the thesis we will concentrate on Kent and Codd third normal forms. We will not consider irreducible rela-

tions since, in practice, we are far more likely to be able to achieve our optimality criterion with Kent or Codd third normal forms, and, as we will see in Section 3.3, Kent third normal form is strong enough to ensure that there are no anomalies. Date third normal form puts emphasis on one of possibly many minimal keys, the primary key. Keys for a given relation are functionally dependent on each other, or are in one-to-one correspondence. Thus, if possible, we would like all minimal keys to coexist in a single relation. We would certainly not want the choice of one key over another to change the collection of relations resulting from a third normal form algorithm. Thus we will not study Date third normal forms.

3.2 The relationship between covering and third normal form

In this section we will examine, for Kent and Codd third normal forms and the two types of covering algorithms, the following questions:

- Does covering imply third normal form?
- Does third normal form imply covering?

In order to answer these questions, we will first establish a framework for comparison. Coverings have been expressed as a set of functional dependencies, whereas a third normal form collection has been expressed as a set of (A(i), F(i)) pairs.

Supposedly, whenever a relational collection is stored, there is some record of these functional dependencies so the system can tell what columns (if any) it is to check for integrity. We will assume, then, that for each relation in a covering or a third normal form collection, there is enough information to deduce A(i) and the closure of (U A(i), U F(i)). In the case of a Kent third normal form, for example, this could simply mean the attribute names and the minimal keys are stored in some way, since the only functional dependencies necessary in F(i) are of the form K --> B for K a minimal key and $K \cap B = \phi$. For Codd third normal form, storing the minimal keys would not be adequate, since there may be a prime attribute in one minimal key that is not fully dependent on another minimal key. For such a functional dependency to be maintained by the system, it has to be recorded. Covering algorithms typically output the F(i), and the A(i) are assumed to be any attribute names mentioned in the F(i). With Bernstein's algorithm the input is just F; any attribute names not mentioned in F will have to be accounted for somehow in the final collection.

Thus from a covering algorithm or from a third normal form algorithm, a set of pairs (A(i), F(i)) is deducible. The third normal form collection or the whole covering is denoted by the relational description {(A(i), F(i))}, and the closure of (U A(i), U F(i)) is called the set of functional dependencies <u>embodied</u> by the collection. Clearly

it is necessary for this closure to equal $\underline{F}(A, F)$ to have a covering and for $\cup A(i) = A$. The latter is not always emphasized.

We are now ready to answer the first question: does covering imply third normal form? The answer to this depends on the type of covering. For the simple type of covering [Wang 75], where equal left-hand sides are merged, covering does imply that each relation is in Codd third normal form. Since no redundant functional dependencies are allowed in the whole set of functional dependencies, no redundant (transitive) dependencies can exist within the functional dependencies for one relation. We will see that it is not always possible to remove redundancy within a collection of relations.

For optimal covering, with merging of equivalent keys [Bernstein 76a, 76c], a purely covering approach no longer necessarily guarantees Codd third normal form. Consider the following example: let A = {a, b, c, d, e, f} and let F consist of:

ab --> c

ab --> d

ab --> e

de --> a

de --> b

de --> f

f --> c

The minimal keys are ab and de. The covering algorithm requires all of A to be put into one relation. However, c is then transitively dependent on de, one of the two minimal keys. As a result, Bernstein must look for transitivities [Bernstein 76b]. It will be seen that the algorithms presented in Chapter 5 do the merging in a different way to avoid these problems.

Before looking at this question for Kent third normal form, let us consider the following example suggested by Date [Date 75, p. 108]. Let A = {student, subject, teacher} or {s, j, t} and let F = {sj --> t, t --> j}. A covering approach would try to make both functional dependencies into relations. The relational system ({t, j}, {t --> j}) would be in Kent third normal form but ({s, j, t}, {sj --> t, t --> j}) would not be because the prime attribute name j is dependent on t and therefore t should be a key, but it is not. In fact, there is no Kent third normal form for this system that covers the functional dependencies (in particular that covers sj --> t). Thus a Kent third normal form that covers the functional dependencies does not always exist.

Note that with the first type of covering algorithm, we could conclude that the collection was in Codd third normal form without checking for transitivities. However, in light of the sjt example, once we have a covering, even a simple covering, we would still have to check for a violation of

Kent third normal form (or whether or not a Kent third normal form exists). Any merging of equivalent keys in a covering approach only increases the potential of introducing Kent third normal form violations. Thus the covering approach alone can never guarantee a Kent third normal form.

To answer the question "does a Codd or Kent third normal form imply a covering?" consider the following example: A = {person, project, department, location} or A = {p, j, d, l} and F consists of:

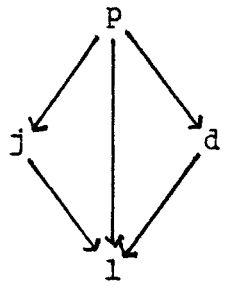
p --> j

j --> l

p --> d

d --> l

Figure 3.2



This can be represented by the diagram in Figure 3.2. Each of the three relations in the following collection is in

both Codd and Kent third normal form, and thus the collection is in both third normal forms.

A(1) = {p, j} F(1) = {p --> j}

A(2) = {j, l} F(2) = {j --> l}

A(3) = {p, d} F(3) = {p --> d}

Note that U A(i) = A is satisfied. However, d --> l is not in the closure of (U A(i), U F(i)) and thus this collection does not represent a covering. If the relationship between d and l is to be maintained, it has to be represented explicitly in the third normal form collection. Therefore A(4) = {d, l}, F(4) = {d --> l} should be added.

We should note another integrity problem illustrated in this example. The functional dependency p --> l is in the closure of (A, F). In retrieving (p, l) pairs from the third normal form tables, we could use the union of the joins {p, j} * {j, l} and {p, d} * {d, l}. Care must be taken that the two "join paths" never disagree. For single tuple insertions or updates, this can be checked by at most 4 single tuple look-ups. However the claim that non-violative operations in one relation in a collection cannot be violative in the closure [Heath 71, Delobel 72] is clearly false because p --> l could become false by changing a tuple in either A(2) or A(4).

In summary, neither third normal form nor covering is a requisite of the other. In the next section we will see that satisfying both criteria is necessary to maintain

integrity. As a consequence, in Chapter 5 we will look for a covering third normal form. It appears that Kent third normal forms, when they exist, are much harder to find than Codd third normal forms, and that finding a covering first may not help. Thus we begin, in Chapter 5, by finding covering Codd third normal forms and then, in Chapter 6, deal with the problems of finding Kent third normal forms.

3.3 Covering and third normal form with respect to anomaly removal

The original reason for third normal forms and coverings was to ensure that certain anomalies did not occur. We observed that these anomalies can arise on insertion, update or deletion of a tuple in a relation. Some of the situations described meant that an insertion or update was ambiguous, e.g. adding a student, course and time in the timetable relation. Others had to do with maintaining redundant information, e.g. the room a professor is in at a certain time.

Let us first examine ambiguity anomalies. In the timetable example in section 2.1, we could not add a student, course and time tuple, because none of the entries in the table had all the key values filled in for the minimal key {student, course}. If we establish the rule that for at least one minimal key (which would then be called the

primary key), no entry is allowed with unknown key values, then none of these ambiguous situations can arise. We are hoping that the ability to enter many forms of partial information will be provided by breaking up the original relation into subrelations, which, if possible, should correspond to update units. The above rule concerning no unknown primary key values also applies to all subrelations.

Since the Kent third normal form definition was found to be the strongest, let us see how well it handles the remaining type of anomaly, namely the redundancy anomaly. The Boyce-Codd wording is perhaps more helpful here. It says that, for one relation, the only functional dependencies that exist in the closure are those whose left sides are keys. This means that, since the values for keys are unique for all tuples, as far as we can deduce from the functional dependencies given, there are no redundant attributes in any one relation.

Thus, as far as one relation is concerned, Kent third normal form has no anomalies. However, we are usually dealing with a collection of relations, and there is nothing in the Kent third normal form definition to prevent the collection having a --> b in one relation, b --> c in a second relation, and a --> c in a third. If we were to update an (a, c) pair in the third relation, then the (a, c) derivable by joining the first two relations might not agree. It is this type of redundancy in the whole collec-

tion of relations that is prevented by insisting the U F(i) be a minimal covering. The third relation represents a redundant functional dependency and would not be allowed. This points out, however, a slight drawback to having a collection in Kent third normal form that embodies a minimal covering. If one wanted to enter an (a, c) pair and did not have a value for b, one would only be allowed to enter (a, ?) into the first relation. That is, not all update units can be accomodated if both third normal form and a minimal covering of the functional dependencies are desired.

Another problem created by requiring a minimal covering of the functional dependencies was exemplified in the previous section by figure 3.2. If an attribute name appears on the right side of more than one functional dependency in a minimal covering, then perhaps more than one relation must be consulted in order to maintain the integrity of the closure of the functional dependencies. This cannot be avoided in any type of third normal form, as long as covering the functional dependencies is also required. The designer may want to omit the maintenance of some functional dependencies when these situations arise.

Since Kent third normal form does not always exist, e.g. in the {s, j, t} relation, the designer may choose to put the relation into Codd third normal form. Again the definition describes what anomalies are allowed to exist, namely transitive dependencies of prime attribute names on

another minimal key. In the {s, j, t} example, sj and st are the minimal keys and j, a prime attribute name, is transitively dependent on st. These cases will be seen to be easily identified by the algorithms in Chapter 5. To maintain the functional dependencies in a situation such as this, whenever a tuple is added, a check must be made to see that one other j is the same for this t, assuming all data is currently correct, or that this t has no j associated with it. If {t, j} is also to be allowed as a relation, (in order to avoid deletion dependencies) two strategies could be used: all (t, j) pairs could be kept in a second relation which is consulted whenever an insertion or update is done to the first; or only those (t, j)'s not currently in {s, j, t} could be stored in this second relation (which could be considered an "overflow relation"). In either case, adding a triple to the first relation, say {s(1), j(2), t(1)} that disagrees with {t(1), j(1)} in the second, would have to be identified either as an update to the second relation as well as an insertion to the first, or as an insertion to the first to be checked for validity (and then possibly deleted from the second relation) if another type of ambiguity anomaly is to be avoided.

If update units are not important, i.e. we are not concerned that there is a place to put each functional dependency such that the left side is a key, and we are not concerned that deleting an {s, j, t} entry means losing a

(t, j) pair, then it would not be necessary to keep a relation for {t, j}. The functional dependencies are still covered as long as t --> j is mentioned in the set of functional dependencies for {s, j, t}. Some other technique or storage structure could still be used to maintain t --> j within the relation containing s, j, and t.

In conclusion, if a collection of relations {(A(i), F(i))} embodies a minimal covering of the given functional dependencies, and if each relation is in Kent third normal form, then any single tuple insertion or update can be carried out without anomalies arising, and no deletion dependencies can occur, with the following two exceptions: if there is more than one way to derive a functional dependency in the closure, then some extra work may be required to maintain this functional dependency in the closure; and if an update unit is not represented explicitly, we may not able to accomodate this update unit. If a collection of relations is in Codd third normal form and embodies a covering of the functional dependencies, then as well as the above two problems, there may be transitive dependencies of prime attribute names to maintain.

If the update units are more important to the designer than third normal form and covering (e.g., the units may be beyond his/her control) then the algorithms to be presented later could still be used to get as close as possible to third normal form and covering. Perhaps more importantly,

the algorithms allow the designer to see how many functional dependencies would have to be maintained to provide integrity checking. Since we assume that all interactions are made with arbitrary subsets of A, we will not pursue this idea further.

3.4 Conditions for reconstructing the original relation

We began in Chapter 2 by assuming that the data base exists because someone wants to query it. We also assumed that one large table might be used for any attributes that may be related in a query. The problems caused by updating and partial information forced us to break the large table into subtables, but the natural join of data in the subtables should be what would exist if the data were put in the large table.

This idea corresponds to work on decomposition [Rissanen 73, 77, Delobel 73b]. Rissanen, in particular, studies the necessary and sufficient conditions under which a relation can be projected onto two or more relations whose join equals the original relation [Rissanen 77]. One such sufficient condition is that the intersection of the two attribute name sets (which is the set over which the join is made) be a key for one of the two relations. Note that if we join in an arbitrary way, (for example, if the intersection is empty, the join results in taking the cartesian

product) the resulting table could contain "more facts" than were in the original table. Since we are not finding decompositions alone, we will use this condition as well as others to guarantee that joins can be performed to reconstruct the relation R with all of A as its attribute name set. The algorithms in Chapter 5 will produce a set of relations satisfying these conditions.

Note that decomposition does not guarantee third normal form because it could produce, for instance, two or more relations which are "not yet" in third normal form. For example, the relation given in Figure 3.2 could be decomposed into {p, j} and {p, d, l}, the latter not being in (any) third normal form. If the decomposition is carried out until irreducible relations are reached, then, by the results in Section 3.1, the relations are all in (all types of) third normal form.

A decomposition does not necessarily imply a covering of the functional dependencies. This is illustrated by Delobel and Casey's algorithm for finding all decompositions [Delobel 73b]. In the first example on page 381 of that paper, two of the four decompositions do not cover the functional dependencies.

Neither covering the functional dependencies nor third normal form guarantees a decomposition since neither, alone, requires that $\cup A(i) = A$.

Consider a simple example with A = {borrower,

borrower's address, book, book's library} and functional dependencies:

borrower --> borrower's address

book --> book's library.

There is no functional relationship between books and borrowers; the relationship is many-to-many. Yet we may want to know how books and borrowers are related at any given time to answer such queries as: "Are there any books from library A borrowed by person B?"

The covering approach would be to cover the two functional dependencies by two disjoint relations R(1) {borrower, borrower's address} and R(2) {book, book's library}. A join of these two relations would show all books being borrowed by all readers, since the join operation between non-intersecting relations is equivalent to forming the cartesian product. This is definitely not what we want.

The problem here is that, considering all of (A, F), nowhere in the collection {R(1), R(2)} is there a key for all of (A, F). The only minimal key this system has is {borrower, book}. If we make this R(0), all of which is the key, then the above problem is solved. Thus one criterion for reconstructing A in a meaningful way is that there must be a (minimal) key for A in one of the relations in the collection.

In order to establish the sufficient conditions for

reconstructibility, we need the following definitions and basic results.

For a set of functional dependencies F' on 2**A and subset B of A, if F' contains L --> R such that L $\subseteq$ B but R is not a member of B, then B is F'-expansible and B U R is an F'-expansion of B.

The following lemma follows directly from the definition of the closure F(A, F):

Lemma 3.7: For subset B of A and F-expansion B' of B, B --> B' and B' --> B in the closure.

The following lemma can be proved by induction on i, where i is the induction counter in the definition of the projective, transitive and additive closure of (A, F). The induction step involves considering elements in F(i) obtained by transitivity and additivity.

Lemma 3.8: For each i, a subset of A is F(i)-expansible if and only if it is F-expansible. Consequently a subset of A is F-expansible if and only if it is F-expansible.

Thus for a relational system (A, F) and any key K, there is at least one sequence of given functional dependencies f(1), f(2), . . . f(m) called a derivation of A from K such that f(j) $\in$ F and K = A(0), A(m) = A and each A(j) is the expansion of A(j-1) obtained by applying functional dependency f(j). Note that since every application is an expansion, each f(j) can appear at most once in any derivation.

Lemma 3.9: Given a relational collection {(A(i), F(i))} that contains a key for A in some A(i) such that the functional dependencies embodied constitute a covering of (A, F) and such that for each functional dependency f in the covering, the left-hand side of f is a key for one relation in the collection, then relation R whose attribute name set is A can be reconstructed by a series of joins in which each join is over the key of one of the two relations involved.

proof: Consider a derivation of A from the key K. The derivation gives a sequence of attribute name sets K = B(0), B(1), . . . , B(m) = A corresponding to applications of functional dependencies f(1), f(2), . . . , f(m) respectively. By the hypotheses in the lemma, since U F(i) is a covering, such a derivation is possible and furthermore each such f(k) is embodied in some F(i). Let C(0) be the attribute names in the relational description containing K. Then B(0) $\subseteq$ C(0). Let R(0) be the relation whose attribute name set is C(0). Then the following algorithm gives the required join sequence.

```
For j <-- 1 until m do
    if B(j) ⊄ C(j-1)
        then let S(j) be the relation for which
                the left-hand side of f(j) is a key
             R(j) <-- R(j-1) * S(j)
        else R(j) <-- R(j-1).
```

Since every join is over the key of the S relation, the

condition that it be over the key of one of the two relations is satisfied. Also, since B(m) = A, R(m) has attribute name set A.

Corollary: If the conditions of the Lemma are satisfied, $\cup A(i) = A$.

It is interesting to note that Delobel and Casey have given a set of conditions under which R is reconstructible [Delobel 73b]. Their conditions are weaker than ours in that they do not require a covering of the functional dependencies but simply insist that each attribute name be chained to a key. This means that there must be at least one "join path" that includes every attribute name. This condition could be substituted for the covering requirement in Lemma 3.9 but, since we will be finding a covering anyway, we chose to show reconstructibility in terms of covering.

3.5 Conclusions

In this chapter we have examined three criteria for relations: minimal covering, third normal form and reconstructibility. We have seen that we want to satisfy all three conditions, and that none of them alone implies any of the others. The algorithms to be presented in Chapter 5 will generate relational systems satisfying all three conditions, which has not been done before.

Bernstein's algorithm generates a covering third normal form collection, but to guarantee reconstructibility it needs an additional requirement that one of the relations contain a key for all of A. We will also see that, by finding all minimal keys for all relations, our algorithm can, in some cases, find a much smaller collection than Bernstein's can. Rissanen's atomic relations [Rissanen 77] guarantee reconstructibility, cover the functional dependencies, and are irreducible, which is, of course, a stronger requirement than ours. However, he does not give an algorithm for generating a set of atomic relations.

Chapter 4

Computational complexity of key finding

4.1. Preliminary results

The definition of Codd third normal form is concerned with properties of nonprime attribute names in a relational system (A, F). Thus one way of finding third normal forms would be to start by finding all prime, and thus nonprime, attribute names. At first it is not clear whether or not this can be done without actually finding all the minimal keys for (A, F). In section 4.5 we will show that the problem of deciding whether or not a given attribute name is prime is NP-complete [Cook 71, Karp 72]. Problems in this class can all be solved, or more precisely the corresponding languages can be recognized, in polynomial time on a nondeterministic Turing machine. It is an open question whether or not they can be recognized in polynomial time on a deterministic Turing machine. In fact, all known algorithms for these problems are exponential. This means that it is unlikely that there is an algorithm polynomial in |A| and |F| to determine if an attribute name is prime.

Given, then, that it probably requires exponential time to determine prime attribute names, the next question we ask is "how hard is it to find all the minimal keys?" We will show below that the number of minimal keys for a given (A, F) can be exponential in |A| and factorial in |F|, and that both of these bounds are attainable. However, in

practical cases, the number of keys is typically very small. An algorithm that finds keys in time polynomial in the inputs to the problem, |A| and |F|, and in the output, i.e. the number of keys found, would therefore be considered acceptable in most cases. Such an algorithm is given in Section 4.6.

The following lemma follows directly from Lemma 3.7:

Lemma 4.1: For subset B of A and F-expansion B' of B, B is a key if and only if B' is a key.

The next lemma follows from Lemma 3.8:

Lemma 4.2: If a proper subset of A is a key for (A, F), then it is F-expansible.

In the following lemma an algorithm is presented and its time complexity is given. For this algorithm and those that follow, we assume that the elementary step being counted is the comparison of two attribute names. Thus if we assume that subsets of A are represented as sorted lists of attribute names, then a Boolean operation on two subsets of A requires at most |A| elementary steps.

The algorithm uses as a subroutine (called Key here) Bernstein and Beeri's algorithm to determine whether a single functional dependency is in the closure of a given set of functional dependencies [Bernstein 76c]. Beginning with a given left-hand side B, their algorithm constructs a set called DEPEND containing all attribute names in any right-hand side of a functional dependency in the closure

with B as its left-hand side. The algorithm ends by testing whether or not DEPEND contains a specific attribute name. To use this algorithm to determine if a given set is a key, we simply need to change this test to:

```
if DEPEND = A
    then return "true"
    else return "false".
```

Bernstein and Beeri show that this algorithm runs in time linear in the size of the input, which in their case is linear in the length of the description of the functional dependencies (in canonical form). We prefer to analyze algorithms in terms of |A| and |F|. Note that, for functional dependencies in canonical form, the length of the functional dependencies is bounded by |A| * |F|, so there is a direct correspondence between their analysis and ours.

For key K of (A, F), attribute name b of K is essential to K if K - {b} is not a key for (A, F). We then have

Lemma 4.3: For key K of (A, F), subset K' of K and attribute name b of K, if b is essential to K and K' is a key then b lies in K' and is essential to K'.

Corollary: The following algorithm determines a minimal key for (A, F) that is a subset of a specified key K. Moreover, it requires $O(|F|\ |A|^2)$ elementary operations.

```
Algorithm Minimal Key (A, F, K);
        K' <-- K;
        for each attribute name b in K do
```

```
        Comment if b is nonessential to K' then delete
           it from K';
        if Key (A, F, K' - {b})
           then K' <-- K' - {b};
     return K'.
```

4.2 An attainable upper bound in terms of F

Yu and Johnson prove that, for any d, where d is the cardinality of F, a relational system can be constructed to have d! minimal keys [Yu 76]. The following lemmas prove that this is in fact an upper bound, in terms of |F|, on the number of minimal keys. They use the notion of a derivation defined in Section 3.4.

Lemma 4.4: For a derivation of A from a key K, (f(1), f(2), . . . , f(m), where f(j) is of the form L(j) --> H(j)) and an attribute name b $\in$ A - K, there is a value J such that $1 \leq J \leq m$, b $\notin$ L(j) for $j \leq J$, and b $\in$ H(J).

proof: If b $\in$ A - K and f(1), f(2), . . . , f(m) is a derivation of A from K, then b $\notin$ A(0) and b $\in$ A(m). Let J be the first f(j)-expansion such that b $\in$ A(J); that is b $\notin$ A(j) for $j < J$. The definition of expansible requires that $L(i) \subseteq A(i-1)$ and that $A(i-1) \cup H(i) = A(i)$ for all $1 \leq i \leq m$. Therefore b $\notin$ L(j) for $j \leq J$ and b $\in$ H(J). □

Let P be the set of all permutations of a given set of functional dependencies F. For any key K for the system

(A, F), let P(K) be the subset of P having the sequence of functional dependencies comprising one derivation of A from K as its prefix. That is, P(K) = {p ∈ P | p = f(1),f(2), . . .,f(m),f(m+1), . . . ,f(k) and f(1), . . .,f(m) is the given derivation of A from K and f(m+1), . . .,f(k) forms an arbitrary permutation of the remaining elements in F}.

Lemma 4.5: If K1 and K2 are distinct minimal keys for (A, F), then the intersection of P(K1) and P(K2) is null.

proof: Assume f(1), f(2), . . . ,f(k) ∈ the intersection of P(K1) and P(K2), where f(1), f(2), . . . ,f(m) is the derivation of A from K1 and f(1), f(2), . . . , f(m1), f(m1+1), . . . , f(m2) is the derivation of A from K2 (m2 ≥ m1). Further assume that K1 = {a(i) | 1 ≤ i ≤ x} U {b(i) | 1 ≤ i ≤ y} and that K2 = {a(i) | 1 ≤ i ≤ x} U {c(i) | 1 ≤ i ≤ z}; that is, the intersection of K1 and K2 is {a(i)}. Let f(S):L(S) --> H(S) be the first functional dependency in the derivation of A from K1 where the intersection of {c(i)} and H(S) is not empty. The existence of S is guaranteed by Lemma 4.4. Consider T such that f(T):L(T) --> R(T) is the first functional dependency in that same derivation where the intersection of {b(i)} and L(T) is not empty.

- Case 1: T ≤ S. Consider the derivation of A from K2. By Lemma 4.4, there must be a t < T such that the intersection of {b(i)} and H(t) is not empty. Let t' be the smallest such t; that is, for all j < t, the intersection of {b(i)} and H(j) is null, and the

intersection of {b(i)} and L(j) is null. Thus f(1), f(2), . . . , f(t') is a derivation of the set A(t') from {a(i)}, where some b(i) ∈ A(t'). This implies that K1 is not minimal and thus is not a minimal key.

- Case 2: T is not ≤ S (that is, T does not exist or T > S). This implies that for all j < S, the intersection of {b(i)} and L(j) is empty. Therefore f(1), f(2), . . . , f(S) is a derivation of the set A(S) from {a(i)}, where some c(i) ∈ A(S). This implies that K2 is not minimal.

In conclusion, if the intersection of P(K1) and P(K2) is not empty, for K1 ≠ K2, then either K1 or K2 is not a minimal key. □

Corollary: For any relational system (A, F), there can be at most d! minimal keys, where d is the cardinality of F.

proof: Each minimal key can be associated with one or more subsets of all permutations of F such that no two such subsets intersect. There are at most d! such subsets (i.e., when each permutation is in its own subset). □

4.3 An attainable upper bound in terms of A

If two sets of attribute names are minimal keys, then one of them cannot be a subset of the other, or the minimality condition of the keys would be violated. Thus the number of minimal keys for a given (A, F) cannot be

greater than the maximum number of subsets of A, no one of which is a subset of another. Let $\underline{K}$ = {K(1), K(2), . . . , K(m)} where each K(i) is a subset of A, such that K(i) $\not\subseteq$ K(j) for any i, j. Consider the (undirected) graph G whose vertex set is A, and whose edge set is $\underline{E} = \bigcup_{1 \le i \le m} E(i)$, where each E(i) is given by: E(i) = {(a,b) | a, b $\in$ K(i)}. That is, the vertex set corresponding to each K(i) is completely connected in G. Furthermore, since K(i) $\not\subseteq$ K(j) for any i, j, it follows that each K(i) is a maximal completely connected set or a clique in G. The following result from graph theory places an upper bound on the number of cliques in a graph with n vertices [Even 73]:

$$f(n) = \begin{cases} 3^{**}r & \text{if } n = 3r \\ 4 \cdot 3^{**}(r-1) & \text{if } n = 3r + 1 \\ 2 \cdot 3^{**}r & \text{if } n = 3r + 2 \end{cases}$$

Thus, f(n) is also an upper bound on the cardinality of $\underline{K}$, and therefore on the number of minimal keys for a given A.

This upper bound is achieved by the following relational systems (A, F). For n = 3r, let A = {a(i) | 1≤i≤r} U {b(i) | 1≤i≤r} U {c(i) | 1≤i≤r} and let F consist of {a(i) --> b(i), b(i) --> c(i), c(i) --> a(i) | 1≤i≤r}. Then for this relational system, every minimal key must contain, for all i, either a(i), b(i) or c(i). Thus, (A, F) has 3**r distinct minimal keys.

For n = 3r + 1, let A = {a(i) | 1≤i≤r-1} U {b(i) | 1≤i≤r-1} U {c(i) | 1≤i≤r-1} U {d, e, f, g}, and let F

consist of {a(i) --> b(i), b(i) --> c(i), c(i) --> a(i) | 1≤i≤r-1} U {d --> e, e --> f, f --> g, g --> d}. Every minimal key for this system must contain, for all i, either a(i), b(i) or c(i) as well as one of d, e, f or g. Therefore this (A, F) has 4•3**(r-1) minimal keys.

For n = 3r + 2, let A = {a(i) | 1≤i≤r} U {b(i) | 1≤i≤r} U {c(i) | 1≤i≤r} U {d, e} and let F consist of {a(i) --> b(i), b(i) --> c(i), c(i) --> a(i) | 1≤i≤r} U {d --> e, e --> d}. This system has 2•3**r minimal keys, each containing, for all i, either a(i), b(i) or c(i) and either d or e.

4.4 NP-completeness of the key of cardinality m problem

In the next two sections we show that two problems related to candidate keys are NP-complete. The first problem is that of deciding whether or not there is a key of cardinality less than or equal to a specified integer m is NP-complete. We refer to this problem as the key of cardinality m problem. The second is that of deciding whether or not a specified attribute name is prime, which we call the prime attribute name problem, is NP-complete. The key of cardinality m result is an intermediate result for the proof that the prime attribute name problem is NP-complete. It is also interesting in its own right.

There are two aspects to proving that a language L is

NP-complete. The first is to exhibit a nondeterministic polynomial algorithm for recognizing L. This shows that L lies in NP. The second part of such a proof, showing completeness, is typically done by transforming a known NP-complete problem into L. Such a transformation must be performed in polynomial time so that a polynomial time algorithm for recognizing L, if ever found, would yield a polynomial algorithm for all NP-complete languages.

The key of cardinality m problem can be stated as follows: given a set A of attribute names, binary relation F on 2**A and integer m, decide whether or not there exists a key for (A, F) having cardinality less than or equal to m.

Theorem 4.1: The key of cardinality m problem is NP-complete.

proof: The problem lies in NP. Nondeterministically generate a subset of A, say K, and then verify whether K is a key containing no more than m attributes. Since algorithm Key is polyncmial so is this algorithm.

To complete the proof of the theorem, it now suffices to prove that the vertex cover problem, an NP-complete problem stated below, is polynomially transformable into the key of cardinality m problem.

The vertex cover problem: given integer m and graph G having vertex set V(G) and edge set E(G), decide whether or not G has a vertex cover of E(G) having cardinality not greater than m. A graph G consists of elements called

edges, together with a relation of incidence that associates two vertices of V(G) with each edge α of E(G), called the ends of α. Each edge α in E(G) is incident upon its ends. A vertex cover K of E(G) is a subset of V(G) such that each edge of E(G) is incident upon some vertex in K. Vertex b of V(G) is adjacent to vertex c of V(G) if E(G) contains an edge having b and c as its ends. The vertex cover problem is NP-complete [Karp 72].

To transform this into the corresponding key of cardinality m problem, define A to be V(G) and F to be {N(v) --> {v}: v lies in V(G)} where N(v) denotes the set of vertices in V(G) that are adjacent to v.

Observe that (A, F) can be determined in time polynomial in |V(G)| and |E(G)|. Note also that |A| = |V(G)| = |F|. Moreover, by Lemma 4.6, asserted below, the vertex cover problem is polynomially transformable into the key of cardinality m problem.

Lemma 4.6: Subset K of A is a key for (A, F) if and only if it is a vertex cover of E(G) in G.

proof: Assume as inductive hypothesis that the assertion holds for each subset K' of A that properly includes K. If K is equal to A then the assertion holds trivially. Assume therefore that K is a proper subset of A. By Lemmas 4.1 and 4.2, K is a key for (A, F) if and only if it has a F-expansion that is a key for (A, F). By construction of F, K has a F-expansion if and only if V(G) - K contains a

vertex, say v, such that K includes N(v). By the induction hypothesis, K U {v} is a vertex cover of E(G) in G. That is, proper subset K of A is a key for (A, F) if and only if it is a vertex cover of E(G) in G, as asserted. □

4.5 NP-completeness of the prime attribute name problem

Given a set A of attribute names, binary relation F on 2**A and attribute name b in A, decide whether or not b is prime relative to (A, F).

Theorem 4.2: The prime attribute name problem is NP-complete.

proof: The problem lies in NP. Nondeterministically generate a subset of A and then verify whether it is a minimal key that contains b. To complete the proof, it suffices to prove that the key of cardinality m problem is polynomially transformable into the prime attribute name problem. For this, assume that set A', binary relation F' and integer m have been given as input to the key of cardinality m problem. Define sets A, F and {b} for the corresponding prime attribute name problem as follows: Define A to be A' U [A" x A'] U {b}, where A" is a set whose cardinality is the smaller of |A'| and m, and b is a "new" attribute name not in A' U [A" x A']. Define binary relation F on 2**A as follows:

(i) for each pair E, F of subsets of A', if E -F'-> F

then {b} U E --> F,

(ii) {b} U A' --> A" x A',

(iii) for each element i of A" and each element e of A', {b} U {(i, e)} --> {e},

(iv) for each element i of A" and for each pair e, f of distinct elements of A', {(i, e), (i, f)} --> {b},

(v) for each element e of A', {e} --> {b},

(vi) no ordered pairs other than those given by (i) - (v) lie in F.

Observe that $|A"| \leq |A'|$, $|A| \leq |A'|^2 + |A'| + 1$ and $|F| \leq |F'| + 1 + |A'|^2 + |A'|^3 + |A'|$. Thus, by Lemma 4.7 stated below, the key of cardinality m problem is polynomially transformable into the prime attribute name problem.

Lemma 4.7: Attribute name b is prime relative to (A, F) if and only if (A', F') has a key of cardinality not greater than m.

proof: Consider first the case where (A, F) has a minimal key, K, that contains b. By the minimality of K, and in view of (iv) and (v) above, K = {b} U {(i(1), c(1)),...,(i(n), c(n))}, where i(1),...,i(n) are n distinct elements of A", c(1),...,c(n) are elements of A' and $n \geq 0$. We assert that {c(1),...,c(n)}, denoted by K', is a key for (A', F'). To prove this, let L' denote a maximal subset of A' such that (K', L') lies in the closure of F'. If L' is equal to A', then K' is indeed a key for (A', F'). Assume thus that L' is a proper subset of A'. By the choice of L'

and by Lemmas 4.1 and 4.2, L' is a proper subset of A' that includes K' but is not F'-expansible. Thus, K U L' is a proper subset of A that includes K but is not F-expansible. By Lemma 4.2, K is not a key for (A, F), a contradiction. As asserted, K' is a key for (A', F'). Indeed, $|K'| \leq n \leq |A''| \leq m$. As asserted, if b is prime relative to (A, F), then (A', F') has a key having cardinality not greater than m.

To prove the converse, assume that (A', F') has a key, denoted K', having cardinality n not greater than m. Denote the elements of K' by c(1),...,c(n). Since $n \leq |A'|$ and $|A''| = \min(|A'|, m)$, A'' contains n distinct elements: denote them i(1),...,i(n). We assert that {b} U {(i(1), c(1)),...,(i(n), c(n))}, denoted K, is a key for (A, F). To prove this, define L to be a maximal subset of A such that (K, L) lies in the closure of F. By Lemmas 4.1 and 4.2, by the choice of L and in view of (iii) above, L is a subset of A that includes K U K' but is not F-expansible. Thus, L $\cap$ A' is a subset of A' that includes K' but is not F'-expansible. Since K' is a key for (A', F'), so is L $\cap$ A'. By Lemma 4.2, L $\cap$ A' = A'. Thus L includes {b} U A', whence L = A by (ii) above. As asserted, K is a key for (A, F).

Finally, K - {b} is a proper subset of A that is not F-expansible, whence b is essential to K. Thus K includes a minimal key for (A, F) that includes b, by Lemma 4.3. That

is, b is prime relative to (A, F). □

4.6 An algorithm for finding all minimal keys

Several algorithms for finding all minimal keys have been presented in the literature [Bernstein 75a, Bernstein 75b, Delobel 73b, Fadous 75]. The following example has exactly one minimal key and yet each of these algorithms requires 2**A steps or more to find the one key. This particular example has 8 attribute names and 4 functional dependencies; it can be generalized to larger examples where |A| = 2 |F|. Let A = {a,...,h} and let F contain:

a --> b,

c --> d,

e --> f,

g --> h.

This system has exactly one minimal key, namely {a, c, e, g}.

Bernstein's approach is to test, for all subsets A' of A, whether A' is a key and whether it is minimal. This algorithm takes 2**A steps regardless of the number of keys found.

For this example, Delobel and Casey's algorithm involves finding the prime implicants for the following Boolean function:

F = abcdefgh + ab' + cd' + ef' + gh'.

They show that the prime implicants with no complemented variables correspond to minimal keys. Known algorithms for generating prime implicants run in time exponential in the number of variables, or in this case, in time exponential in |A|.

Fadous and Forsyth's algorithm is not analyzed nor is it easy to deduce its worst-case behaviour. For examples in the family described above, one of the temporary sets used by this algorithm, T2, gives some idea of how it works. The first time T2 is constructed, it contains Choose(|F|, 2) elements after elements which are supersets of others have been discarded. The next time T2 is constructed it contains Choose(|F|, 3) elements after supersets are discarded. The pattern continues, and in general, the number of operations required to maintain T2 alone is more than

$$\sum_{i=2}^{|F|} \text{Choose}(|F|, i)$$

or more than 2**|F| steps.

In [Lucchesi 77] an algorithm is presented which finds all minimal keys for a relational system in time polynomial in |A|, |F| and in the number of keys found. As pointed out by Bernstein and Beeri, this is not really what we need when finding the collection {(A(i), F(i))} of covering third normal form relations. The covering approach, for example, yields such sets A(i) together with some of the minimal keys

for (A(i), F(i)). What we then want to find is all minimal sets of attribute names K such that K --> A(i) and A(i) --> K. These could be called minimal keys for A(i) except that they are not necessarily in A(i). They are in some sense equivalent to A(i) with respect to all of F in what they can "derive".

Let us define the maximal expansion A(i)* of a subset A(i) of A to be all attribute names that are in any F-expansion of A(i). Intuitively A(i)* contains everything "derivable" from A(i). When we speak of finding all minimal keys for A(i), what we really mean is finding all K $\subseteq$ A(i)* such that K is a minimal key for A(i)*.

Maximal expansions can be found using Bernstein and Beeri's membership algorithm, again changing the output from the algorithm to return the set DEPEND as the expansion. This version or their algorithm will be called Expansion below.

Using this algorithm to find A(i)* for the A(i) in question, we can then restrict our attention to the functional dependencies whose left-hand sides are subsets of A(i)*, by Lemma 4.8 below. This lemma follows directly from Lemma 3.8.

Lemma 4.8: If a subset B of A is F-expansible, then B or a proper subset of B is a left-hand side of a functional dependency in F.

These functicnal dependencies, then, constitute F(i).

Note that F(i) can be determined in 0(|F| |A(i)*|) steps. By definition of A(i)* we also know that the right-hand sides of all functional dependencies in F(i) are subsets of A(i)*.

We are now ready to show the conditions under which an additional key for A(i)* exists.

Lemma 4.9: Given an arbitrary subset A(i) of A, F(i) and a nonnull set K of minimal keys for A(i)*, 2**A - K contains a minimal key for A(i)* if and only if K contains a minimal key K and F(i) contains L --> R such that L U (K - R) does not include any key in K.

proof: To prove that under these conditions 2**A - K contains a minimal key for A(i)*, assume there is a K ∈ K and L --> R in F(i) such that L U (K - R) does not include any other key in K. Since L U K U R includes K and K is a key for A(i)*, L U K U R is a key for A(i)*. It is also true that L U (K - R) --> L U K U R; therefore L U (K - R) is also a key for A(i)*. Thus L U (K - R) contains a minimal key, say K'. Since L U (K - R) does not include any key in K, K' is not already in K.

To prove the converse, assume that 2**A - K contains a minimal key K' for A(i)*. Define K" to be the maximal subset of A that includes K' but does not include any key in K. Since K is nonnull, K" is a proper subset of A(i)*. Since K' is a key for A(i)*, so is K"; i.e. K" --> A(i)* is in F(A, F). By Lemmas 4.1 and 4.2, K" is F-expansible. By

definition of A(i)*, it is F(i)-expansible; i.e. F(i) contains L --> R such that L $\subseteq$ K" and R $\not\subseteq$ K". By the choice of K", K" U R includes a key in $\underline{K}$, say K. That is, K" includes L and K - R, or L U (K - R) $\subseteq$ K". Since K" does not include any key in $\underline{K}$, neither does L U (K - R). □

If A(i) ≠ A, then before we can start finding keys, we must first find A(i)* and F(i). If A(i) = A, we simply rename A and F. Then, before we can apply the lemma, we must have a nonnull set of minimal keys. This can be accomplished by applying the Minimal Key algorithm to all of A(i)* to isolate one minimal key. Then the test in the lemma is used to determine if there are any remaining keys, again using Minimal Key to isolate one such key whenever the test is satisfied. In the algorithm, it is assumed that $\underline{K}$ is accumulated as a sequence which can be scanned in the order in which the keys are entered.

Corollary: The following algorithm determines the set of minimal keys for an arbitrary subset A(i) of A, given a set F of functional dependencies over A.

Algorithm Set of Minimal Keys (A(i), F);

```
if A(i) ≠ A
     then B <-- Expansion (A(i), F)
          F(i) <-- all functional dependencies
               in F whose LHS ⊆ B
     else B <-- A;
          F(i) <-- F;
```

```
K <-- {Minimal Key (B, F(i), B)};
for each K in K do
     for each pair L --> R in F(i) do
          S <-- L U (K - R);
          test <-- true;
          for each J in K do
             if S includes J then test <-- false;
          if test
             then K <-- K U {Minimal Key (B, F(i), B)};
return K.
```

Observe that the algorithm takes $O(e + |F| + |F(i)|\ |K|\ (|B| + |K|\ |B|) + |K|\ m)$ elementary operations where e is the complexity of Expansion and m is the complexity of Minimal Key. That is, the algorithm has time complexity $O(|F|\ |B| + |F(i)|\ |B|\ |K|\ (|K| + |B|))$. For any A(i), this is no greater than $O(|F|\ |A|\ |K|\ (|K| + |A|))$.

Bernstein and Beeri have shown that finding an additional key for a set A(i), a sub-relation of a larger system (A, F), is NP-complete with respect to the size of A(i). This result does not contradict the analysis of algorithm Set of Minimal Keys because the algorithm finds one more key in time polynomial in $|A(i)|$, $|F(i)|$ and the number of keys already found, where F(i) is determined after A(i)* has been found, and there is no reason for $|A(i)^*|$ to be small with respect to $|A(i)|$. The point is that this algorithm will be used when finding a third normal form

collection for (A, F), and any algorithm that is polynomial in |A|, |F| and anything else we want to know, e.g. keys, is considered acceptable. In finding third normal forms, this algorithm will be invoked at most r times, where r is the number of relations in the final collection; for the covering algorithms, and for the algorithm in Chapter 5, r is bounded by |F|.

Chapter 5

Algorithms for finding covering Codd third normal forms

5.1 The graphical approach

In order to develop algorithms for covering Codd third normal form, we will use an approach based on directed graphs. This section is devoted to establishing an appropriate model.

A directed graph G(V, E) consists of a finite, non-empty set of vertices V and a set of ordered pairs of vertices called edges. For a relational system (A, F), the equivalent directed graph has labeled vertex set V = 2**A, the labels being subsets of A. Furthermore, for each functional dependency L -> R in F, there is an edge (L, R) in the equivalent directed graph. Where there is no possibility of confusion, we will simply call this the graph for a relational system.

Since the third normal form and covering definitions pertaining to relational systems are in terms of the (projective, transitive and additive) closure F(A, F), we will now define a closure operation for the equivalent directed graph. Given a set V of vertices and a set E of edges, the equivalent-graph closure G* of G(V, E) is a directed graph consisting of the same vertices V, and all edges in

$$\bigcup_{i \geq 1} E(i)$$

where E(1) is the union of E and all edges obtained by

• graph projectivity: for X and Y in V, if the label for Y is a subset of the label for X, then (X, Y) is in E(1);

and the E(i), i ≥ 2, are defined to be exactly those edges obtained by the following:

• graph transitivity: for all vertices X, Y, and Z in V, if (X, Y) and (Y, Z) are in E(i-1) then (X, Z) is in E(i);

• graph additivity: for all vertices X, Y and Z of V, if (X, Y) and (X, Z) are in E(i-1) and Y and Z are not both the empty set, then (X, YZ)[1] is in E(i).

Lemma 5.1: For a relational system (A, F) and its equivalent graph G(V, E), an edge (L, R) is in the equivalent-graph closure iff L --> R is in the projective, transitive and additive closure of (A, F).

We will now associate labels with the edges of the equivalent graphs, in the following way:

label	meaning
given	a functional dependency given in F
projectivity	derived by graph projectivity
transitivity	derived by graph transitivity
additivity	derived by graph additivity

In all figures, the underlined letter will be used as the label.

Note that an edge could be derived in more than one way. For example, ({bc}, {b}) could be a given functional dependency or it could be derived by graph projectivity.

---

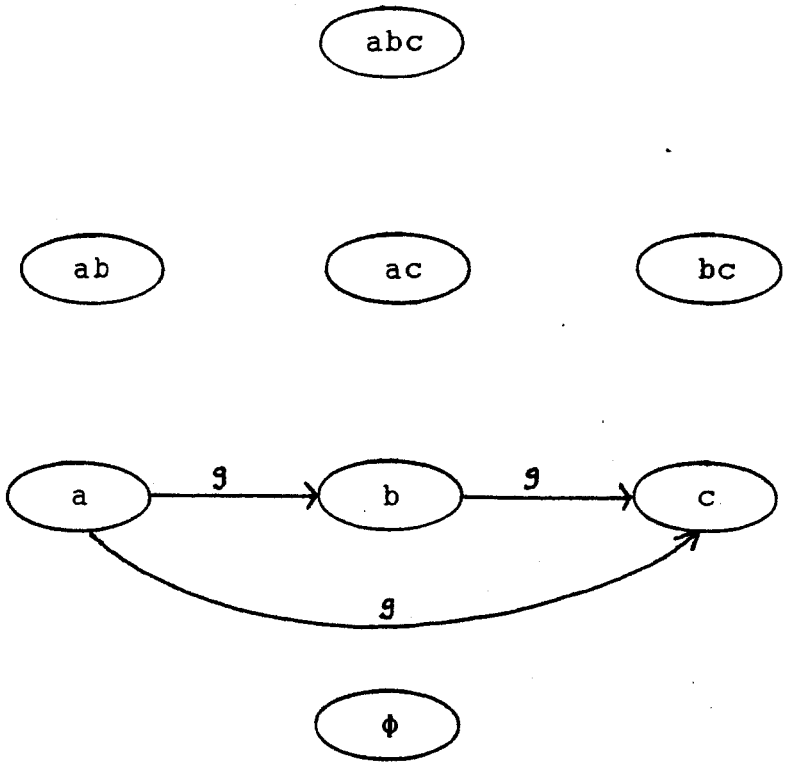
[1] YZ is the label of the vertex corresponding to Y U Z.

Whenever this happens, we will choose the edge label as follows: if any one of them is derived by projectivity, then the label is projectivity; otherwise if one of them is a given edge, then take given. In the only other cases having a choice, the edge was derived by both transitivity and additivity, the choice is made as follows: if the label of the terminus of the edge is a single attribute name, the edge is a transitivity edge; otherwise the edge is an additivity edge. We choose a projectivity label above all others because it reflects the set structure of the attribute name sets, and the fact that one element of 2**A is a subset of another can never be "deleted" from whatever we are considering. The above ranking of transitivity and additivity is needed to distinguish between vertices corresponding to single attribute names and those corresponding to sets of attribute names in characterizing third normal forms in a graph. In this way, priority is given to the behaviour of single attribute names because it is the single attribute name behaviour that is specified by the third normal form definitions.

To illustrate the graphical approach, consider the relational system with A = {a, b, c} and F = {a --> b, b --> c, a --> c}. The equivalent directed graph is given in Figure 5.1, and the equivalent-graph closure is shown in Figure 5.2, where projectivity edges are dashed for greater

Figure 5.1

equivalent directed graph



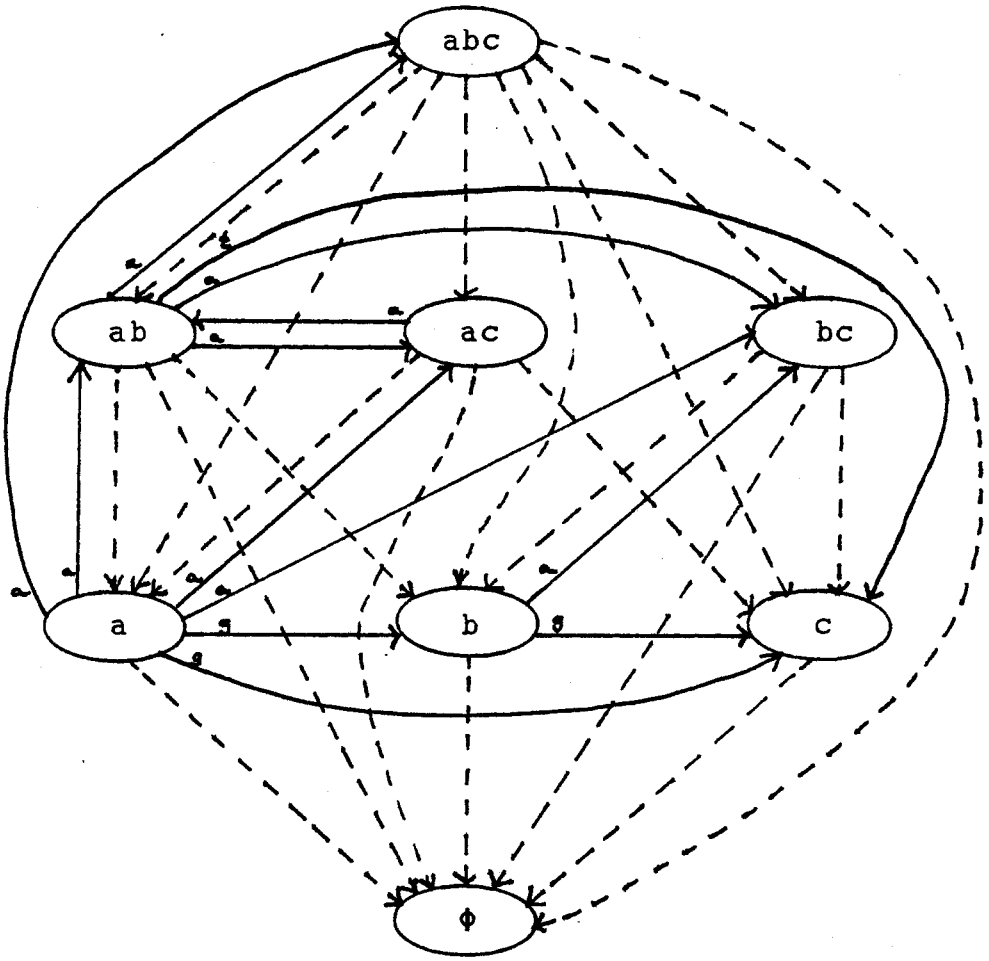
legibility.

In this example, the set of vertices with labels {abc}, {ab}, {ac} and {a} are pairwise connected by edges in the equivalent-graph closure. Since the edge ({a}, {abc}) is in the equivalent-graph closure, then by Lemma 5.1, {a} --> {abc} in $\underline{F}$(A, F). Thus {a} is a key for (A, F); in fact it is a minimal key.

In an equivalent graph G(V, E), V can be divided into

Figure 5.2

equivalent-graph closure



equivalence classes V(i), $1 \le i \le r$, such that vertices v and w are equivalent iff there is an edge (v, w) and an edge (w, v) in the equivalent-graph closure G*. Let E(i), $1 \le i \le r$, be the set of edges connecting vertices in V(i). The graphs G(i) (V(i), E(i)) are called the strongly connected components of G*.

We can make the following observations concerning the strongly connected components of the equivalent-graph closure for a relational system (A, F):

- Since the V(i) in the definition of strongly connected components are equivalence classes, A belongs to exactly one of them.
- Since X --> A for every vertex in the strongly connected component containing A, by Lemma 5.1 every vertex in the strongly connected component containing A corresponds to a key for (A, F).
- Since A --> X for all X in 2**A, by projectivity, for no X outside of A's strongly connected component does X --> A. In other words, there are no edges entering A's strongly connected component. Thus A's strongly connected component contains all the keys for (A, F).

We conclude this section with the following definition and observation. If X and Y represent vertex labels, then X is said to be a minimal member of a strongly connected component if no Y which is a proper subset of X is also a member of the strongly connected component. Thus the minimal members of A's strongly connected component are the minimal keys of the relational system (A, F).

5.2 Covering Codd third normal forms

In this section we will discuss a characterization of covering Codd third normal form but ignore algorithm efficiency. Recall that there are two main properties to satisfy, namely covering the functional dependencies and making sure that each relation in a collection is in third normal form. We will begin with the definitions describing the graph parallel to covering the functional dependencies, and then we will discuss the properties of a graph corresponding to a single relation in third normal form. Finally we will combine the two requirements by giving an algorithm to find a collection of relations satisfying both properties and show that the resulting collection also allows the reconstruction of the original relation.

A reduction G^+ of an equivalent graph $G(V, E)$ is a subset of the equivalent graph whose equivalent-graph closure equals the closure of $G(V, E)$. A reduction G^+ is said to be minimal if, for no proper subset H of G^+, does H^* [1] equal G^{+*}. A reduction G^+ is said to be minimum if there is no reduction J with fewer edges than G^+ such that J^* equals G^{+*}.

By Lemma 5.1 and the definitions in Chapter 2, we have the following basic results:

Lemma 5.2: If $G(V, E)$ is the graph of a relational system (A, F), then G^+ is a minimal reduction of G iff the edge set

---

[1] This notation will be used for closure where the context indicates what closure operation is involved.

of G^+ corresponds to a minimal covering of F.

Lemma 5.3: If G(V, E) is the graph of a relational system (A, F), then G^+ is a minimum reduction of G iff the edge set of G^+ corresponds to a minimum covering of F.

We observed at the end of the last section that all the minimal keys for the relational system (A, F) are in the same strongly connected component of the equivalent graph. The same is true for any subrelation (A(i), F(i)): if K and L are contained in A(i) and are two minimal keys for A(i), then K --> L and L --> K in the closure, and thus they are pairwise ccnnected in the equivalent-graph closure.

Any attribute name in a minimal key of a subrelation is prime with respect to that subrelation. Recall that Codd third normal form requires no transitive dependencies of nonprime attribute names on minimal keys. Any violations of Codd third normal form will be caused by nonprime attribute names which are minimal members of some strongly connected component other than the one containing all the minimal keys. These potential violations will be found by examining the interactions between the strongly connected components, not within them. Thus we define below the acyclic equivalent graph which collapses strongly connected components and allows us to look only at an acyclic representation of the graph.

The acyclic equivalent graph G' of an equivalent graph G(V, E) is constructed from the equivalent-graph closure G*

as follows: Let G(i) (V(i), E(i)), 1≤i≤r, be the strongly connected components of G*. Then G' has r vertices, one for each set V(i), such that each vertex is labeled by a distinct set L(i) = {x | x is the label of v - V(i)}. The edges of G' are given by: (V(i), V(j)) is an edge of G' if and only if there is an edge (u, v) in G* such that u ∈ V(i) and v ∈ V(j). The labels on the edges of G' are given by the following rules:

- If there is a given edge from any member of V(i) to any member of V(j), then (V(i), V(j)) is labeled given.
- Otherwise, if there is a transitivity edge from any member of V(i) to any member of V(j), then (V(i), V(j)) is a transitivity edge.
- Otherwise, if there is an additivity edge from any member of V(i) to any member of V(j), then (V(i), V(j)) is an additivity edge.
- Otherwise (all edges are projective), (V(i), V(j)) is a projectivity edge.

For the example in Figure 5.2, the acyclic equivalent graph is given in Figure 5.3. The minimal members of each strongly connected component are underlined.

Closure and reduction have been defined in terms of G whose vertex set corresponds to 2**A. All cycles in the closure G* have been removed in constructing the acyclic equivalent graph, but the information has been retained in the labels. The closure of the acyclic equivalent graph can

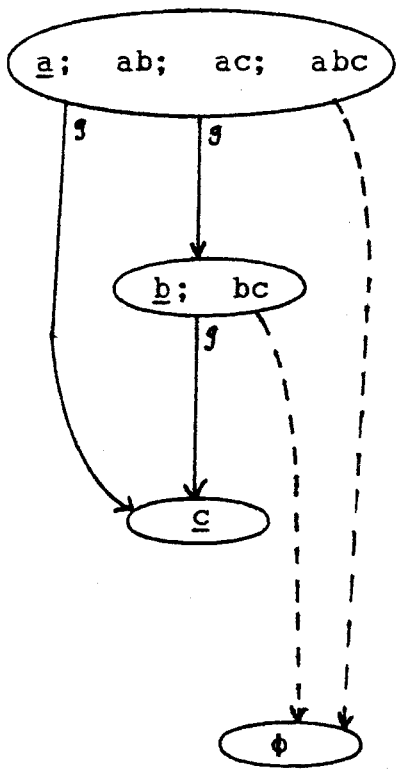


Figure 5.3

be found by replacing each vertex in the acyclic equivalent graph by the set of vertices in the corresponding strongly connected component, and adding edges to form a simple cycle incident to all vertices in the strongly connected component. The closure of this graph is clearly the same as G* (without edge labels). Therefore a reduction G^+ of an acyclic equivalent graph G' is a subset of G' such that $(G^+)^* = (G')^*$, where both closure operations are as

described above.

Aho, Garey and Ullman have proved that for conventional acyclic directed graphs, the minimal reduction is unique [Aho 72]. The following example shows that this is not true for acyclic equivalent graphs. Let A = {u, v, x, y} and F = {u --> v, u --> x, u --> y, xy --> v, v --> y}. The acyclic

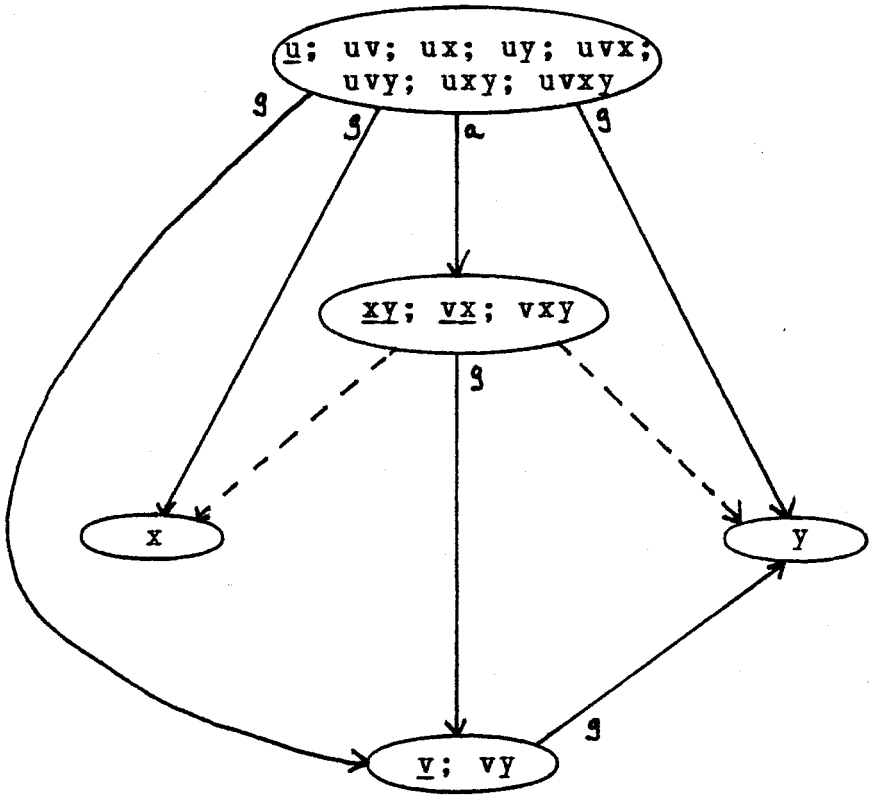


Figure 5.4

equivalent graph is shown in Figure 5.4. Figure 5.5 shows two minimal reductions for this example. In all such

figures, the null set is omitted, projectivity edges (dashed) are included for clarity, and only minimal members of each strongly connected component are shown. The reader can verify that they both have the same closure.

The above example shows that, because of the properties of graph additivity, the minimal reduction is not unique, and also in this case the minimum reduction is not unique. However, it is important to remember that what we want to optimize is the number of relations in a Codd third normal form collection. From the discussion so far, it is not clear whether or not all minimal reductions give rise to equal size sets of relations.

Although we have defined a minimal reduction to be a subset of the given graph, we should consider briefly other graphs, perhaps containing transitivity and additivity edges, which represent minimal sets of functional dependencies having the same closure. Two such examples are shown in Figures 5.6 and 5.7, each with two possible minimal representations. We will see at the end of Section 5.3 that the number of relations produced by the Codd third normal form algorithm does not depend on the cardinality of the minimal covering of the functional dependencies embodied. Thus, since we have a choice, we choose a covering that is a subset of the given functional dependencies, since they are likely to be update units. The following lemma shows that a minimal representation of the acyclic graph corresponding to

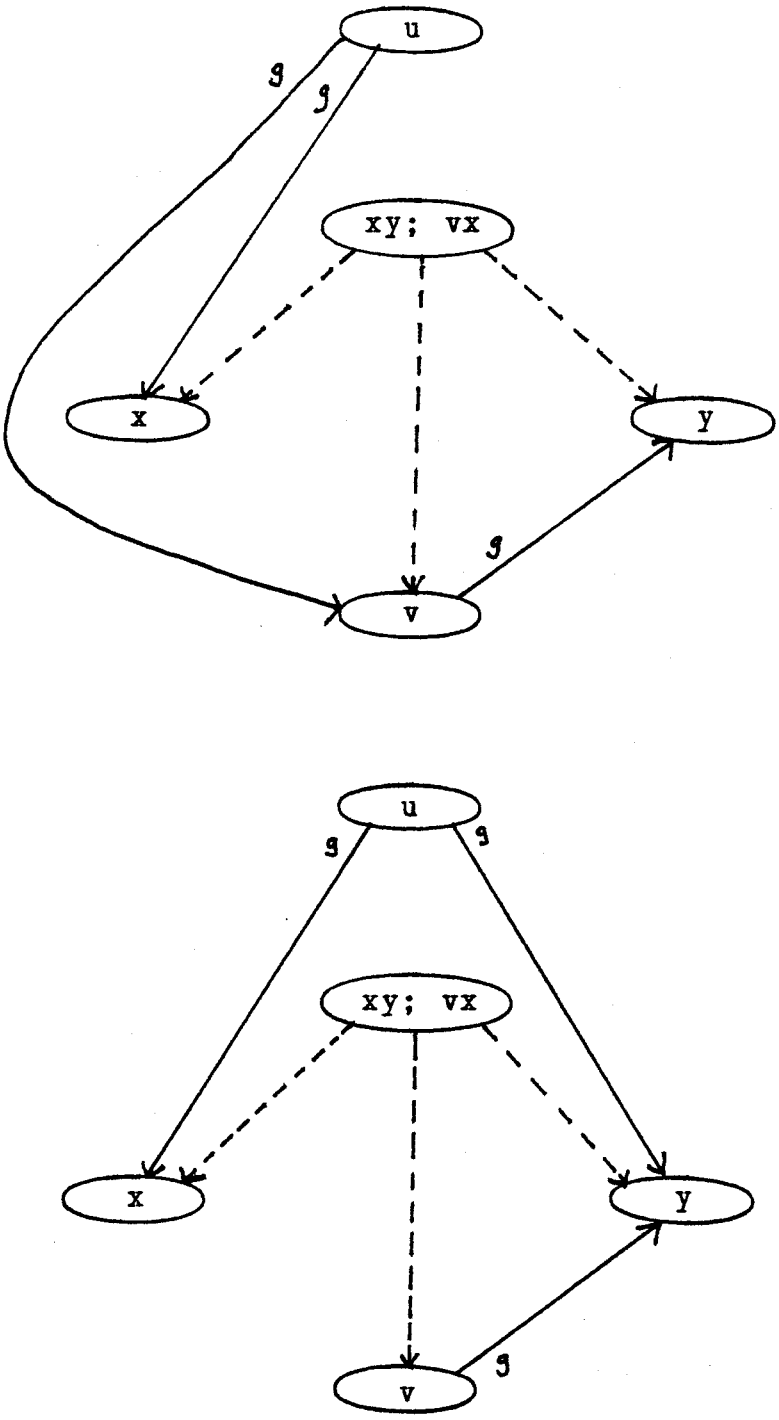


Figure 5.5

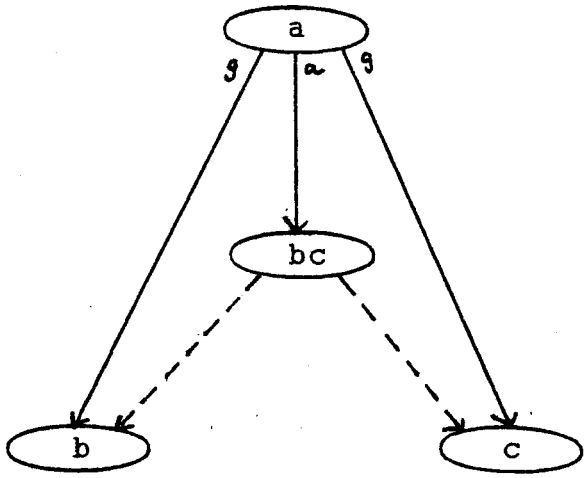
given functional dependencies does always exist.

Lemma 5.4: For the acyclic equivalent graph G' of a relational system (A, F), there is at least one minimal representation that contains only given edges.

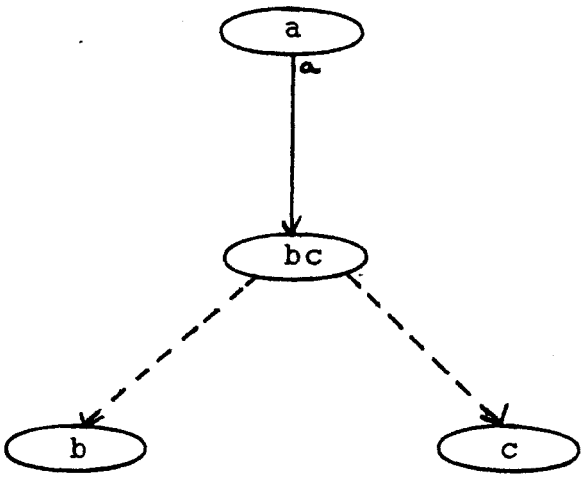
proof: Recall that there will not be projectivity edges in a minimal representation since they are always generated in forming the closure. Also recall that in the acyclic equivalent graph G', a given edge appears wherever there is a given edge between any vertex in V(i) and any vertex in V(j). Thus the given edges of the acyclic equivalent graph form one representation, although not necessarily a minimal one. To form a minimal one from these given edges, temporarily delete a given edge and see if the closure operation yields the same result as before. If it does, then permanently delete this given edge. When no more edges can be permanently deleted in this manner, the resulting set of given edges is a minimal representation, and thus, a

Figure 5.6

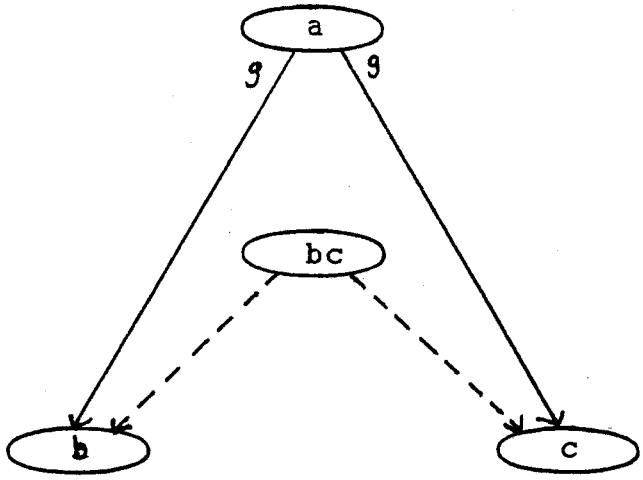
A = {a, b, c} F = {a --> b, a --> c}



5.6 (a) acyclic equivalent graph



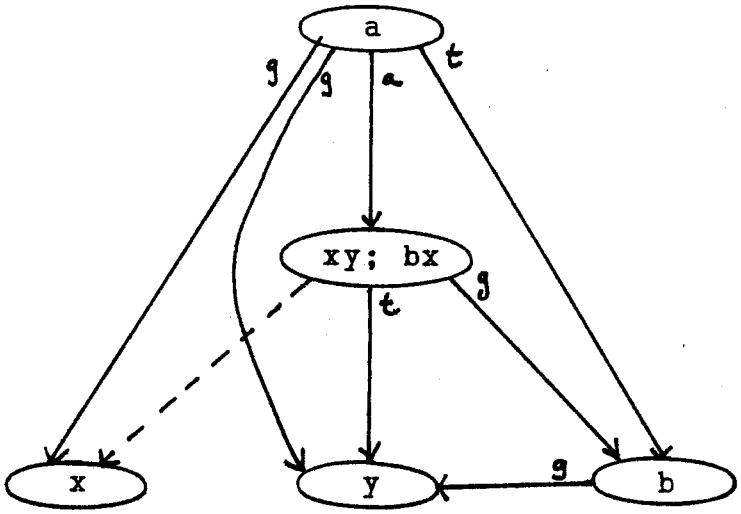
5.6 (b) minimal representation not consisting of all given edges (projectivity edges shown for convenience)



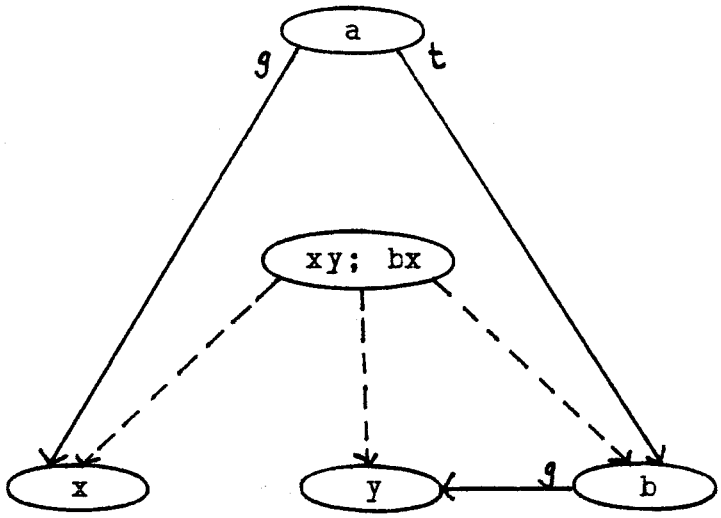
5.6 (c) minimal representation consisting only of given edges (projectivity edges shown for convenience)

Figure 5.7

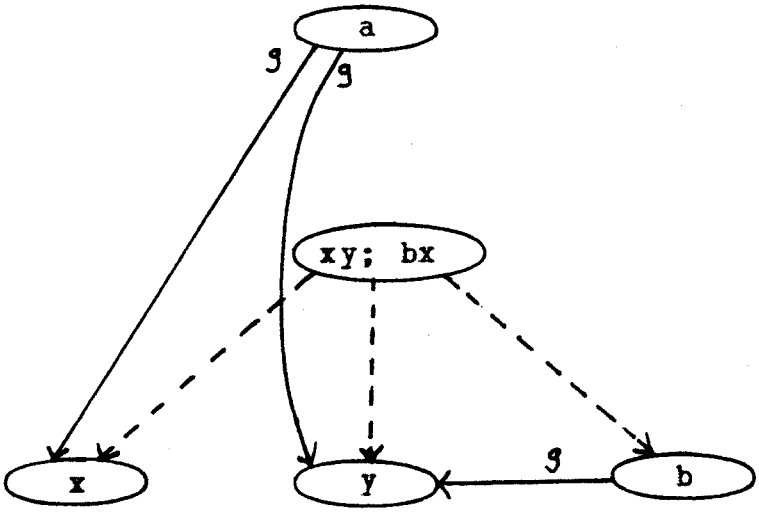
A = {a, b, x, y}
F = {a --> x, a --> y, xy --> b, b --> y}



5.7 (a) acyclic equivalent graph



5.7 (b) minimal representation not all given edges



5.7 (c) minimal representation containing only given edges

minimal reduction. □

This completes the discussion of how covering is

manifested in a graph. We will now turn to third normal forms before combining the two concepts. In order to talk about third normal form, we need a means of identifying potential third normal form violations; i.e. we need a notion similar to prime and nonprime for equivalent graphs and for acyclic equivalent graphs.

A vertex V in the equivalent graph G(V, E) of a relational system (A, F) is all nonprime if all the elements of A in its label are nonprime relative to (A, F). Otherwise it is graph prime. For example, in Figure 5.2, the vertices labeled b, c and bc are all nonprime and all other vertices are graph prime.

In the acyclic equivalent graph G' for a relational system, a vertex is prime if all of its minimal members are graph prime. Otherwise, (at least one minimal member is all nonprime) it is said to be nonprime.

All vertices in the acyclic equivalent graph have at least one minimal member. A vertex is said to be simple if the strongly connected component it represents has exactly one minimal member. Otherwise a vertex in the acyclic equivalent graph is compound. With these definitions, we can now characterize a graph representing a single relation in Codd third normal form.

Theorem 5.1: The acyclic equivalent graph G' of a relational system (A, F) corresponds to a relation in Codd third normal form iff

P1: every nonprime vertex is simple and

P2: there is no path of length greater than one to a nonprime (simple) vertex such that the last edge of the path is a given edge.

proof: Assume (A, F) is in Codd third normal form but that P1 is not true. That is, G' contains a nonprime, compound vertex, say B. Since B is nonprime, it is not the strongly connected component containing all the minimal keys. By definition of nonprime for acyclic equivalent graphs, B contains at least one minimal member, say X, that is all nonprime in G. By the assumption that B is compound, B contains at least one other minimal member, say Y.

If X is a single attribute name, then, being nonprime, it is transitively dependent on any minimal key K; i.e. K --> Y, Y -/-> K, Y --> X and, since Y is minimal, X $\notin$ Y. This contradicts the assumption that (A, F) is in Codd third normal form.

If |X| > 1, then Y --> X is an additivity edge in the closure, or Y --> x(i) for all x(i) $\in$ X. Since X is a minimal member of B, none of the x(i) are in B. Since X is all nonprime, all the x(i) are nonprime. Y $\not\subseteq$ X since X is minimal in B. Therefore there is at least one x(i) $\notin$ Y. This x(i) is transitively dependent on any minimal key, contradicting the Codd third normal form assumption.

Now let us assume that (A, F) is in third normal form but that P2 is not true. That is, G' contains a path of

length greater than one ending in a given edge into a nonprime simple node. Because this ends in a given edge, by canonical form for functional dependencies the simple node in fact contains only a single attribute name, say z. Let this path be C --> B --> z where C and B are two other strongly connected components. B is not the strongly connected component containing the keys since there is an edge coming into it. Thus if K is the strongly connected component containing all the keys, K --> B --> z. Since B --> z is a given edge, there is at least one member Y in B such that Y --> z is a given edge in the closure and thus z $\notin$ Y. Thus z is a nonprime attribute name transitively dependent on a minimal key, contradicting (A, F) being in Codd third normal form.

To prove the converse, assume we have an acyclic equivalent graph satisfying the two conditions in the theorem but that the corresponding relational system is not in Codd third normal form. That is, there is a transitive dependency of a nonprime attribute name, say z, on a minimal key, say X. In other words, there exists some Y $\subseteq$ A such that X --> Y, Y $\not\rightarrow$ X, Y --> z and z $\notin$ Y. Since Y $\not\rightarrow$ X, X and Y are in different strongly connected components. Since z is nonprime, by P1, z is the only minimal member of its strongly connected component. Since z $\notin$ Y, z and Y are in different strongly connected components. Since Y --> z, there is an edge from Y to z in the equivalent graph closure

which cannot be a projectivity edge since z ∉ Y, cannot be an additivity edge because |z| = 1, and therefore is either a given edge or a transitivity edge. Therefore, by definition of the acyclic equivalent graph, the edge from Y's strongly connected component to z's is either a given edge or a transitivity edge. If it is a given edge, then P2 is contradicted. If it is a transitivity edge, then by the two lemmas below, we also have a contradiction to P2.

Lemma 5.5: In the equivalent-graph closure, G*, of the graph for a relational system (A, F), if there is a transitivity edge from vertex M to vertex N, then there is also a path from M to N of length greater than one ending in a given edge.

proof: Let M* be the maximal expansion of M. Then in the closure, M --> M*. Consider M* - N. Since this is a proper subset of M*, M* --> M* - N by projectivity. Thus, by transitivity, M --> M* - N is in the closure. Since in the closure, M --> N is not a projectivity edge, then in forming the maximal expansion M*, there is some point at which N is the right-hand side of a functional dependency in F which is used. Since M* is maximal, the left-hand side of this functional dependency, call it L, must be a subset of M* - N. Therefore in the equivalent-graph closure, there is a path M --> M* - N --> L --> N, the last edge of which is given in F. □

Lemma 5.6: In the acyclic equivalent graph for a relational

system, if z is the only member of its strongly connected component and B --> z is a transitivity edge, then there is a path from B to z of length greater than one ending in a given edge.

proof: Since in the acyclic graph B --> z is a transitivity edge, there exists at least one X in B such that X --> z is a transitivity edge in the equivalent-graph closure. By Lemma 5.5, there is a path in the equivalent-graph closure from X to z ending in a given edge. Let this given edge be W --> z. If W were in B, then B --> z would have been a given edge. Since W is on a path from X to z in the closure, X --> W is in the closure by transitivity. Thus there is an edge from B to W's strongly connected component in the acyclic equivalent graph. Since z is the only minimal member of its strongly connected component, W and z are in different strongly connected components and thus there is a path B --> W's strongly connected component --> z ending in a given edge. □

The proof of these two lemmas completes the proof of Theorem 5.1. □

We have shown how Codd third normal form is characterized in these graphs, and we know that a minimal reduction corresponds to a minimal covering of the functional dependencies. Thus to achieve a set of relational descriptions {(A(i), F(i))} each in Codd third normal form, such that U A(i) = A and the closure of U F(i)

= the closure of F, i.e. a covering Codd third normal form, we are going to extract subgraphs from a minimal reduction of the acyclic equivalent graph in such a way that each subgraph is guaranteed to correspond to a relation in Codd third normal form.

The input to the algorithm is a minimal reduction of the acyclic equivalent graph of a relational system, consisting only of given edges; i.e. a vertex set V^+ which contains the strongly connected components, and an edge set E^+ consisting of the given edges in the reduction.

Algorithm Extract Relations (V^+, E^+);

```
1. i <-- 1
2. While the graph is nonempty do
3.     V <-- any source in V+
4.     A(i) <-- A(i)^0 <-- union of all minimal members of V
5.     F(i) <-- F(i)^0 <-- enough functional dependencies to
           generate the pairwise connections between minimal
           members of V
6.     for each edge (V, W) in E+ do
7.         if W contains no minimal member that is a subset
                of A(i)
8.             then A(i) <-- A(i) U one minimal member of W
9.                  F(i) <-- F(i) U the edge
                        (one minimal member of V, the member of
                        W chosen above)
10.        delete edge (V, W)
```

```
11.    if F(i) is empty and V is not the key strongly
           connected component
12.        then discard (A(i), F(i))
13.        else i <-- i + 1
14.    delete V
15. for all i do
16.    F(i) <-- F(i) U all functional dependencies in
           E+ U F' whose left and right sides are
           contained in A(i)
```

Remark 1: $F(i)^0$ can be constructed in the following way: for each pair X, Y of minimal members in the strongly connected component, $F(i)^0$ contains X --> y(j) for each y(j) $\in$ Y and Y --> x(k) for each x(k) $\in$ X. These are in canonical form and are also sufficient to generate the strongly connected component. Of course any projectivity edges thus created should be deleted from $F(i)^0$.

Remark 2: for each (A(i), F(i)), the closure of (A(i), F(i)) is a subset of the closure of (A, F). That is nothing is added by the algorithm in terms of functional dependencies that was not known in the context of all of (A, F).

Remark 3: in the acyclic equivalent graph for any (A(i), F(i)) generated by the algorithm, every strongly connected component is a subset of a strongly connected component in the acyclic equivalent graph for (A, F). This follows from remark 2: since we have not added any

functional dependencies, nothing is pairwise connected that was not pairwise connected in G*. Thus strongly connected components in G* may get split up, but they cannot be merged together.

Remark 4: A(i)^0 is a strongly connected component in the acyclic equivalent graph for (A(i), F(i)). Furthermore it contains all the keys for (A(i), F(i)).

Recall from section 5.1 that there are no edges in G* entering the strongly connected component containing all the minimal keys for A. Thus this strongly connected component is a source in G*, and in G$^+$. Therefore line 3 of the algorithm chooses this strongly connected component as A(i)^0 for some i. This gives the following basic lemma.

Lemma 5.7: One A(i) contains a key for (A, F).

Since each F(i)^0 reconstructs the strongly connected component for A(i)^0, and all the given edges in a minimal reduction are included by line 6 in at least one F(i), we also have the following basic result:

Lemma 5.8: For every given edge (and thus functional dependency) in the minimal reduction, there exists an (A(i), F(i)) for which the left-hand side of the corresponding functional dependency is a key.

Lemma 5.9: The closure of (U A(i), U F(i)) equals F(A, F). That is, {(A(i), F(i))} embodies a covering of (A, F).

The above three lemmas satisfy the conditions of Lemma 3.9. Therefore we have:

Corollary: The relation R with attribute name set A is reconstructible from {(A(i), F(i))}.

To show that each relational description (A(i), F(i)) is in Codd third normal form, we must show that its corresponding graph obeys the properties of Theorem 5.1.

Lemma 5.10: For each (A(i), F(i)) constructed by the algorithm, all nonprime nodes are simple.

proof: This follows directly from line 8 of the algorithm and from remark 3. □

Lemma 5.11: For each (A(i), F(i)) constructed by the algorithm, in the acyclic equivalent graph for each system there are no paths of length greater than one to a nonprime node ending in a given edge.

proof: By remarks 1 and 3, A(i)^0 is a strongly connected component in the acyclic equivalent graph for (A(i), F(i)).

Suppose there is such a path in the acyclic equivalent graph for (A(i), F(i)). Then it would be of the form $\underline{B}$ --> $\underline{C}$ --> z where $\underline{C}$ --> z is a given edge, z is simple and z $\notin$ A(i)^0. By remark 4, if A(i)^0 contains all the keys for (A(i), F(i)) then there are no edges leading into the strongly connected component corresponding to A(i)^0. Thus $\underline{C}$ is distinct from A(i)^0. Furthermore, since A(i)^0 contains all the keys, there is an edge from A(i)^0's strongly connected component to $\underline{C}$ in this acyclic equivalent graph. Thus there is a path A(i)^0 --> $\underline{C}$ --> z ending in a given edge. Now in G^+, A(i)^0 and $\underline{C}$ are distinct. $\underline{C}$ and z are

also distinct strongly connected components in G+ or there would not have been a given edge to add to F(i). Thus by remark 1, this path is also in G+. Now, since z is nonprime, A(i)^0 --> z is a given edge in G+ by construction of A(i). Thus, A(i) --> z is also in G+. This functional dependency is redundant in G+, contradicting the assumption that G+ is a minimal reduction. □

5.3 Efficient algorithms for covering Codd third normal forms

Our objective in this section is to give algorithms for carrying out the theoretical results given in Section 5.2. All of the graphs discussed in 5.2 have 2**A as their vertex set. Any polynomial graph algorithms on such a graph would result in covering third normal form algorithms which are exponential in the size of the inputs to our problem, namely in |A| and |F|. Thus, if possible, we would like to avoid such a large graph, and rather manipulate a graph whose size is polynomial in |A| and |F|.

Notice, for example, that the algorithm for extracting {(A(i), F(i))} from the graph only refers to the minimal members of the strongly connected components and given edges between the strongly connected components. Thus one improvement we might make is to ignore completely those members of 2**A which are not minimal in their strongly

connected components.

We can also economize by reducing the number of relations we keep: the algorithm, as it stands, produces too many relations. Consider the following example: A = {a, b, x, y, z} and F:

x --> y

x --> z

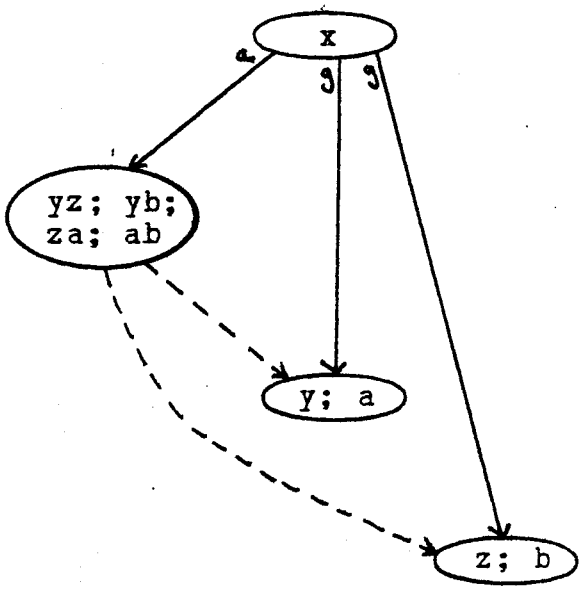
y --> a

a --> y

z --> b

b --> z

Figure 5.8



The acyclic equivalent graph is shown in Figure 5.8. The relations extracted by the algorithm would be:

A(1) = {x, y, z} F(1) = {x --> y, x --> z},

A(2) = {a, b, y, z} F(2) = {ab --> y, ab --> z, yz --> a,
yz --> b, yb --> z, yb --> a, za --> b, za --> y},

A(3) = {a, y} F(3) = {y --> a, a --> y},

A(4) = {b, z} F(4) = {b --> z, z --> b}.

Note that in F(2), all of the functional dependencies are non-full dependencies and thus none of them are necessary in the minimal covering. However, since all of the attribute names are prime with respect to (A(2), F(2)), this relation is in Codd third normal form. If the given functional dependencies indicate update units, then information will be inserted as (a, y) or (b, z) pairs, or as (x, y, z) triples. Only when y and z are both known can a (y, z) pair be inserted into R(2). Any retrievals involving the associations between y and z could be answered using R(1) (as long as there is an x value). Thus R(2) is completely useless and unnecessary.

One characteristic of this unnecessary relation is that the strongly connected component that constitutes its A(i)^0 has no given edges leaving it. Another characteristic is that none of its minimal members are the left-hand side of any given functional dependency. Thus if we restrict ourselves to strongly connected components containing the left-hand side of a functional dependency given in F, then we will certainly cover all the given functional dependencies and thus cover $\underline{F}$(A, F). If we delete non-full

dependencies from F before constructing the graph, then we will avoid strongly connected components with all non-full left-hand sides and thus avoid these unnecessary relations while still covering $\underline{F}(A, F)$. For example, yz --> a could be a given functional dependency in the above example, and thus to avoid such a strongly connected component we would want to remove such functional dependencies from F.

In Section 4.6 we gave an algorithm for finding all minimal keys for a given B*, where $B \subseteq A$.

Lemma 5.12: M is a minimal key for B* iff B --> M and M --> B in the closure and M is minimal with respect to M --> B.

proof: If M is a minimal key for B*, then $M \subseteq B^*$ and therefore B* --> M by projectivity. B --> B* by definition of B*. Therefore B --> M. If M is a minimal key for B*, then M --> B* and B* --> B by projectivity. Thus by transitivity M --> B. Suppose M is not minimal with this property. That is, suppose that M' is a proper subset of M and that M' --> B. Then M' --> B --> B*, contradicting the assumption that M is a minimal key for B*.

To prove the converse, if M --> B then M --> B* by B --> B* and transitivity. Thus M is a key for B*. The minimality of M with respect to M --> B implies that M is a minimal key for B*. □

Corollary: If M is a minimal key for B*, then M is a minimal member of B's strongly connected component in the corresponding acyclic equivalent graph. (Note that B itself

need not be a minimal member).

Using this result, then, we can find all the minimal members of strongly connected components containing left-hand sides L, by finding all minimal keys for each L*.

Lemma 5.13: If B is the strongly connected component obtained by finding all minimal keys for B*, given B, then B contains exactly those minimal members in the strongly connected component for B that would be found by taking the closure of G(V, E) and the corresponding acyclic equivalent graph.

proof: This follows from Lemma 5.12 and the fact that the algorithm in Section 4.6 finds all minimal keys for B*. □

We are now ready to give an efficient algorithm to generate a reconstructible covering Codd third normal form collection for a given (A, F).

Algorithm Covering Codd third normal form (A, F)

```
1. delete non-full functional dependencies from F;
2. for each L(i) --> r(i) in F do
       L(i)* <-- Expansion (L(i), F);
3. Comment delete duplicates from the set of expansions.
   C <-- {L(i) found in step 2};
4. if no L(i)* = A
       then C <-- C U {A};
5. for each L(i)* in C do
       K(i) <-- Set of minimal keys (L(i)*, F);
6. Comment define the vertex set for G+;
```

V^+ <-- {$\underline{K}$(i)} U {any single attribute name
z ∈ A that is not a minimal key in one of the $\underline{K}$(i)};

7. Comment define the edge set for G^+;
for each L --> r in F do
if L ∈ $\underline{V}$ and r ∈ $\underline{W}$, $\underline{V}$ and $\underline{W}$ - V^+,
then E^+ <-- E^+ U {($\underline{V}$, $\underline{W}$)};
8. Comment define F', the functional dependencies necessary to regenerate the strongly connected component;
F' <-- {X --> y | X, Y ∈ $\underline{V}$, y ∈ Y, for all $\underline{V}$ in V^+};
9. Comment find a minimal reduction of E^+ using Bernstein and Beeri's membership algorithm;
for each edge e in E^+ do
if Membership (F' U E^+ - e, e)
then E^+ <-- E^+ - e;
10. Comment extract a covering Codd third normal form collection;
Extract relations (V^+, E^+).

Lemma 5.14: The above algorithm produces a collection {(A(i), F(i))} such that U F(i) covers $\underline{F}$(A, F).

proof: Each functional dependency in F is either entered as an edge in E^+ or its left and right sides are in the same label for V^+. In the latter case the functional dependency is covered by F'. Therefore all the given edges are covered, and thus $\underline{F}$(A, F) is covered. □

Lemma 5.15: The above algorithm produces relational descriptions that are in Codd third normal form.

proof: By Lemma 5.13, the labels for V^+ contain all the minimal members for a strongly connected component in the equivalent graph. The lemma follows by observing that the proofs of Lemmas 5.10 and 5.11 do not rely on the fact that G^+ is an acyclic equivalent graph but only on the construction of the strongly connected components, the construction of $\{(A(i), F(i))\}$ from the algorithms, and the fact that G^+ is a minimal reduction. □

Lemma 5.16: A is reconstructible from the relational descriptions produced.

proof: By construction there is a key for all of A in one A(i). Each left-hand side in the minimal covering of F(A, F) is a key for one L* and therefore a key for one of the extracted relations. Thus the collection produced by the algorithm satisfies the conditions of Lemma 3.9. □

Lemma 5.17: The algorithm runs in time $0(|A|^2 |F|^2 + |A|^2 |F| \sum(1,|F|+1)k(i) + |A| |F| \sum(1,|F|+1)k(i)^2 + |A| |F|^2 k')$ where k(i) is the number of keys for (A(i), F(i)) and k' is the largest k(i).

proof: We will analyze the steps separately. Since they are executed sequentially, the total run time will be the sum of the times for each step.

1. Non-full dependencies can be deleted by checking, for each f in F, if the left-hand side of f minus one attribute name implies the right-hand side of f. Thus the membership in the closure algorithm must be called $|A| |F|$ times.

Therefore this step requires $0(|A|^2\ |F|^2)$ operations.

2. Finding $|F|$ maximal expansions takes a total of $0(|A|\ |F|^2)$ operations.

3. At this point, $\underline{C}$ contains at most $|F|$ entries. Therefore deleting duplicates from $\underline{C}$ takes $0(|A|\ |F|^2)$ operations.

4. Adding A to $\underline{C}$, if necessary, requires $0(|A|\ |F|)$ operations and leaves $\underline{C}$ with at most $|F| + 1$ members.

5. Let k(i) be the number of minimal keys found for the ith member of $\underline{C}$. Then, in total, this step takes $0(|A|^2\ |F|\ \sum(1,|F|+1)k(i) + |A|\ |F|\ \sum(1,|F|+1)k(i)^2)$ operations.

6. V^+ will have at most $|A| + |F| + 1$ members. For any members that were already in $\underline{C}$, the labels for these are output by step 5 and thus could be stored at that time in the required format. Setting up vertex labels for the remaining single attribute names involves only $0(|A|)$ steps.

7. There are at most $|F|$ edges to account for. We could keep track, in step 3, of which member of $\underline{C}$ is associated with each left-hand side. The right-hand side of any functional dependency in F must be a single attribute name. Thus, as step 5 is done, if any minimal key is a single attribute name, the member of $\underline{C}$ to which it belongs can be recorded. This saves scanning all minimal keys for all members of $\underline{C}$ in order to record E^+. Thus, if the necessary information is stored during steps 3 and 5, this step takes $|F| + |A|$ operations.

8. There will be, for each minimal key K for each member B

of $\underline{C}$, one functional dependency for each attribute name in any key in B that is not in K. Thus the number of functional dependencies in F' is bounded by $\sum(1,|\underline{C}|)|A|k(i)$ where k(i) is the number of keys for the ith member of $\underline{C}$. Recall that $|\underline{C}| \leq |F| + 1$. Therefore this step requires $0(|A| \sum(1,|F|+1)k(i))$ operations.

9. Each call of Bernstein and Beeri's membership algorithm takes $|A|(|E^+|-1 + |F'|)$ steps. It will be called $|E^+|$ times. Since $|E^+| \leq |F|$, this step takes $0(|A|\,|F|^2 + |A|^2\,|F| \sum(1,|F|+1)k(i))$ operations.

10. The extraction algorithm was not analyzed in Section 5.2. For each vertex in V^+, or in terms of the extraction algorithm, for each source in the graph, step 4 takes $0(|A|k(i))$ operations and step 5 takes $0(|A|k(i))$ operations. Observing that for vertices in V^+ that were added in step 6 of the Codd third normal form algorithm, $k(i) = 1$, these 2 steps, over all times through the loop in the extraction agorithm, take $0(|A| \sum(1,|F|+1)k(i))$ operations. Steps 6 to 8 of the extraction algorithm are done a total of $|E^+|$ times; each time, the k(j) minimal members of $\underline{W}$ are examined. Since (almost) all edges in E^+ could have the same terminus, if k' is the largest k(i), then all executions of steps 6 to 8 take $0(|F|k')$ operations. Steps 15 and 16 of the extraction algorithm are done n times where n is the number of relations left after trivial ones have been discarded. If we note that any vertex added in step 6 of

the Codd third normal form algorithm would be discarded, then steps 15 and 16 of the extraction algorithm are done $|\underline{C}|$ times or 0 ($|F|$) times. Each execution of these steps involves $(|F'| + |E^+|)\ |A(i)|$ operations, or at most $|A(i)|\ (\sum(1, |\underline{C}|)|A|\ k(i) + |F|)$ steps. Thus these steps take at most $0\ (|F|\ |A| \sum(1,|F|+1)k(i) + |F|)$ operations.

Thus, in total, the Codd third normal form algorithm takes $0\ (|A|^2\ |F| + |A|^2\ |F| \sum(1,|F|+1)\ k(i) + |A|\ |F| \sum(1,|F|+1)\ k(i)^2 + |A|\ |F|\ k')$ operations. Note that if each $k(i) = 1$, then the bound becomes $0\ (|A|^2\ |F|^2)$, which is exactly Bernstein's bound of (length of input)^2.

If the $k(i)$'s are not small, then the terms involving $\sum k(i)^2$ and k' dominate. This is to be expected since the number of keys can be so large.

There will be cases where we output one more relation than Bernstein; this occurs when a vertex containing the minimal keys for all of A must be added. Since Bernstein does not require reconstructibility, his algorithm does not do this. Since both algorithms merge equivalent left-hand sides and then produce one relation for each set of equivalent left-hand sides, in all cases we will produce either exactly the same number of relations or one more.

Recall, however, from Section 3.3, that if update units are not essential, then one could simply record a functional dependency like t --> j, in the sjt example, in the set of functional dependencies for {s, j, t}, and not store a

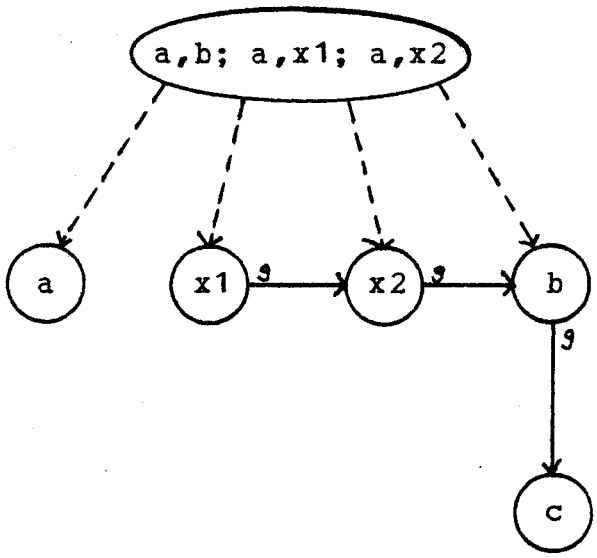
separate relation for t --> j. The relation {s, j, t} is still in Codd third normal form. A slight modification of the extraction algorithm, adding the following three lines, would accomplish this:

17. for all i, j do
18. if A(i) $\subseteq$ A(j)
19. then delete (A(i), F(i))

This modified algorithm has the same worst-case run time as the original algorithm. Each relation is still in Codd third normal form, and U F(i) still covers the functional dependencies.

Consider the following example: A = {a, b, c, x1, x2} and F = {a,b --> x1, x1 --> x2, x2 --> b, b --> c}. The minimal keys for a,b* are a,b, a,x2, and a,x1. The graph (V^+, E^+) is shown in Figure 5.9 with projectivity edges added. With the modified version of the extraction algorithm, only two relational descriptions would be output: A(1) = {a, b, x1, x2} with F(1) = {a,b --> x1, a,b --> x2, a,x1 --> b, a,x1 --> x2, a,x2 --> b, a,x2 --> x1, x1 --> x2, x2 --> b} and A(2) = {b, c} with F(2) = {b --> c}. Even if Bernstein's algorithm discards a relation whose attribute name set is a proper subset of another, it would not be able to do so here since, without generating all keys, there is no way to know that a,x1 and a,x2 are equivalent to a,b. In this example, Bernstein's algorithm would produce four relations. This example can be generalized to a system in which

Figure 5.9



A = {a, b, c, x1, . . . xn} with F = {a,b --> x1, . . . , xi --> xi+1, . . . , xn --> b, b --> c}. In this case our modified algorithm would produce two relations and Bernstein's would produce n + 2.

The collection of relations produced by this modified algorithm no longer satisfies the reconstructibility criteria of Lemma 3.9. Thus we need:

Lemma 5.18: The relation R whose attribute name set is A is reconstructible from the collection {(A(i), F(i))} produced by the modified algorithm.

proof: The only condition of Lemma 3.9 missing is the one requiring that, for every functional dependency f in the covering, there be a relation in the collection for which the left-hand side of f is a key. However, the collection

does embody a covering of the fuctional dependencies. In the reconstruction of A from a key K according to a derivation f(1), . . . , f(m), if the left-hand side of a given f(j) in the derivation is not the key for any (A(i), F(i)), then we do know it is contained in some F(i) by construction. This relation R(i) can be projected onto the attribute names in f(j) and the resulting relation be used as S(j) in the algorithm in Lemma 3.9. □

Note that if the functional dependencies such as f(j) have been maintained, then the join mentioned in the above proof will not generate any "extra facts." We may, however, lose some partial information due to the omission (by choice) of update units such as f(j).

We should note at this point that, with respect to a relation over A(i), the functional dependencies in F(i) which are transitive dependencies of prime attribute names (and therefore have to be maintained) are easily identified because they are all added in step 16 of the extraction algorithm. By construction, before that, all functional dependencies are either those required to reconstruct the strongly connected component or are functional dependencies of nonprime attribute names that appear as edges in E^+. Although some functional dependencies needed to reconstruct the strongly connected component are non-full, it is only the full functional dependency that is added in step 16 that has to be maintained explicitly. Note in the previous

example, a,x1 --> x2 is needed to reconstruct the key node whereas it is x1 --> x2 that has to be maintained. Not all the functional dependencies added in step 16 have to be maintained because some of them could be non-full as well. The point is that we have not worried about minimal representation of the F(i), but only about minimizing the number of relations. The F(i) could be minimized by deleting non-full dependencies, if this were considered necessary.

The remainder of this section is devoted to showing that we have achieved our optimality objectives, that is, that the collection of relations produced by the algorithm has the smallest cardinality possible. Note that we do not require that the size of the covering be the smallest possible nor that the covering contain only given functional dependencies. These lemmas, then, will justify our decision to use only given functional dependencies in the covering and show that no other method could produce fewer relations.

Lemma 5.19: The covering Codd third normal form algorithm produces the smallest possible collection {(A(i), F(i))} such that:

1. each (A(i), F(i)) is in Codd third normal form,
2. U F(i) covers $\underline{F}$(A, F),
3. for every functional dependency f in some minimal covering of $\underline{F}$(A, F), the left-hand side of f is a key for one of the (A(i), F(i)),

4. the original relation A is reconstructible from {(A(i), F(i))}.

proof: The proof is by induction on the number of relations output by the algorithm. Clearly when the algorithm outputs one relation, this collection of size one is optimum. Assume that for all $i \leq k$, the algorithm outputs an optimum collection. To show that this is also true for k+1, assume that the algorithm produces a collection C(1) of size k+1, but there exists a smaller collection C(2) that also satisfies all of these properties. There exists some minimal covering of the functional dependencies such that each relation in C(2) contains either a functional dependency in this minimal covering or the keys for all of A, or such a relation could be deleted from C(2) and all the properties (in particular reconstructibility) would still be satisfied. Consider the equivalent-graph closure G* for the given (A, F) and the minimal covering used in extracting C(1). By construction, there is one relation in C(1) for each strongly connected component in G* with an edge corresponding to a given full functional dependency originating in it. For every strongly connected component incident to an edge in one minimal covering, there is an edge incident to this strongly connected component in every minimal covering. This follows from the fact that every functional dependency in a minimal covering is full, and thus no proper subset of its left-hand side derives its

right-hand side. Therefore, the only way to derive this functional dependency in the closure of another minimal cover is by initially applying additivity or transitivity.

The algorithm produces one relation for each subgraph of G* such that one of the minimal keys for that subgraph is the left-hand side of a functional dependency in the covering. By the above arguments, any other minimal covering would have a functional dependency incident to this strongly connected component. We will now consider all ways in which C(2) can differ from C(1).

Case 1: a relation (A(j), F(j)) in C(2) properly contains a relation (A(i), F(i)) in C(1). If it contains more than one (A(i), F(i)), consider a maximal such A(i). Let B = A(j) - A(i). There are two subcases: A(i) --> B and A(i) -/-> B. If A(i) --> B, then A(i) contains all the keys for A(j) and B is nonprime with respect to (A(j), F(j)). Either this introduces a transitive dependency in (A(j), F(j)), and thus (A(j), F(j)) is not in third normal form, or, for, each attribute name b in B, A(i) --> b nontransitively in the closure of (A(j), F(j)). If A(i) --> b exists and is not a transitive dependency, then it cannot be derived by transitivity in the closure of (A(j), F(j)), and since it is not due to projectivity by the definition of B, then it must be a given functional dependency. In this case the algorithm would have included each of these b's in A(i).

If A(i) -/-> B, then A(i) U B corresponds to a bigger

subgraph of G* with keys in a strongly connected component other than A(i)^0. If A(i) is not contained in any other A(j), then the left-hand side of the functional dependency in the minimal covering incident to A(i)^0 is no longer represented by a relation for which it is a key, contradicting property 3. If A(i) is in another A(j), then by deleting the functional dependencies in A(i)^0 from the given F, the algorithm would produce an optimum collection of size less than or equal to k by the induction hypothesis, and thus C(2) cannot possibly have fewer than k+1 relations.

Case 2: There is a relation (A(j), F(j)) in C(2) that does not contain all of an A(i), for any (A(i), F(i)) in C(1). There are two subcases to consider: the minimal keys for A(j) are keys for some A(i) in C(1), or they are not. If they are, then either A(j) contains some attribute names B not in the A(i) which has the same keys, or it is a proper subset of this A(i). If it does contain extra attribute names B, then by arguments similar to ones used above, if A(j)'s keys --> B nontransitively, then the algorithm would have put B into A(i). If A(j) is a proper subset of A(i), then some other relation in C(2) must cover part of F(i). By removing from the given F the functional dependencies in the minimal covering incident to A(i)^0, the algorithm would produce a collection with less than or equal to k relations, and by the induction hypothesis that collection is optimum. Thus C(2) cannot have fewer than k+1 relations.

The remaining subcase occurs if A(j)'s key strongly connected component is not the key strongly connected component for any A(i) in C(1). That is, the inclusion of B with possibly part of some A(i) to make A(j), requires another strongly connected component to hold the keys for A(j). This implies that A(j)'s strongly connected component is not the strongly connected component containing the keys for all of (A, F). Thus whatever functional dependency in the minimal covering A(j) is covering is a non-full dependency in A(j), thus violating property 3. □

Lemma 5.20: The modified algorithm produces the smallest possible collection {(A(i), F(i))} for a given (A, F) with the following properties:

1. each relation is in Codd third normal form,
2. ∪ F(i) covers $\underline{F}$(A, F),
3. The original relation is reconstructible from {(A(i), F(i))}.

proof: The modified algorithm discards an (A(i), F(i)) if A(i) is properly contained in some A(j). Consider the key strongly connected component for A(i), A(i)^0. After the algorithm is finished, all of A(i)^0 is contained in A(j). Thus either A(i)^0 $\subseteq$ A(j)^0 or A(i)^0 contains only one minimal member, or (A(j), F(j)) would not be in Codd third normal form. Thus there is no choice of representative for A(i)^0 when forming A(j).

By the arguments in the previous lemma, any optimum,

covering, reconstructible, Codd third normal form has to contain a relation covering functional dependencies incident to the key strongly connected component of A(i) and the key strongly connected component of A(j). The only way two collections satisfying the set of properties in the previous lemma could not both accomodate deleting (A(i), F(i)) would be if there is a choice of which minimal member of $A(i)^0$ to include in A(j), which has just been ruled out.

The fact that no other merging of relations in the collection can be done follows from arguments in the previous lemma. □

5.4 Choice of primary and foreign key

As well as allowing us to produce a collection of third normal form relations, the graph input to the extraction algorithm contains other useful information for a data base designer. We will show in this section how this information can be used in specifying primary and foreign keys. In the next section, we will show how to use the information in the graph to answer queries.

A foreign key in a relation described by (A(i), F(i)) is a subset of A(i) which is not a key for (A(i), F(i)) but is the primary key for some other relation (A(j), F(j)) in the collection [Codd 70]. Let us consider the following example: A = {a, b, c}, F = {a --> b, b --> c, c --> b}. The algorithm extracts the following relations:

A(1) = {a, b}, F(1) = {a --> b},

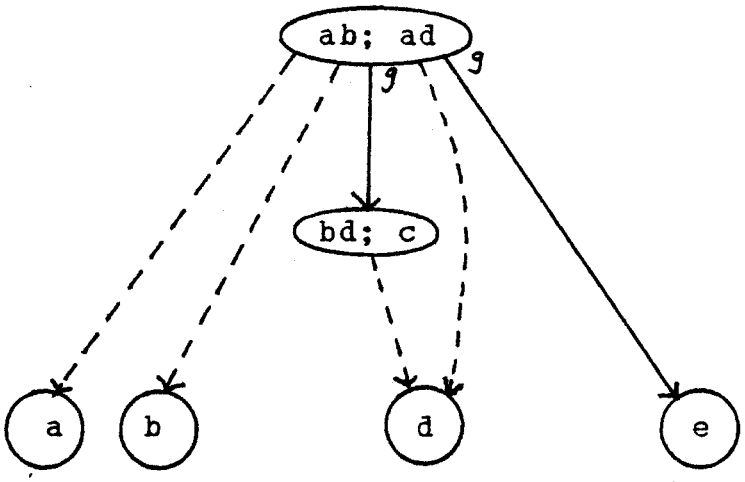
A(2) = {b, c), F(2) = {b --> c, c --> b}.

Attribute name c could replace b in A(1) and F(1). Whichever of b and c is chosen as the "link" to R(2), that is, as the foreign key for (A(2), F(2)) to be included in (A(1), F(1)), should be the primary key for (A(2), F(2)). Recall that it is the primary key values that are not allowed to be unknown, and that joins are performed over the intersection of the two attribute name sets. If this intersection is the primary key for relation R, and R's foreign key in relation S, then the join R * S is performed over a key for one of the two relations involved and

furthermore, there are no unknown values to worry about. On the other hand, the intersection could properly contain the primary key for R, in which case there could be unknown non-key values for attribute names in the intersection. In this case the standard join algorithm should be modified to make use of the primary key's properties and the fact that functional dependencies are being maintained.

The extraction algorithm tells us the choices we have for foreign keys which are also to be the primary keys for their relations. The conditions under which we have a choice are given in line 7, and the choice itself is in line 8. There is no choice when one of the minimal members of $\underline{W}$ is a subset of A(i) as in the following example: let A = {a, b, c, d, e} and F = {ab --> d, ad --> b, ab --> c, ab --> e, bd --> c, c --> b, c --> d}. The transitive reduction of the acyclic equivalent graph is given in Figure 5.10. If c is chosen as the primary key for {b, c, d}, then it must also be the foreign key in A(1). This, however, would introduce a transitive dependency in (A(1), F(1)). Therefore, A(1) must be {a, b, d}, not {a, b, c, d}. Thus, as stated in the algorithm, only if no minimal member of the strongly connected component at the terminus of a given edge is a subset of what is already in A(i) do we have a choice. Note that although edge {ab; ad} --> {bd; c} is shown as a given edge (in fact ab --> c is the given functional dependency that lead to this), this edge is covered in

Figure 5.10



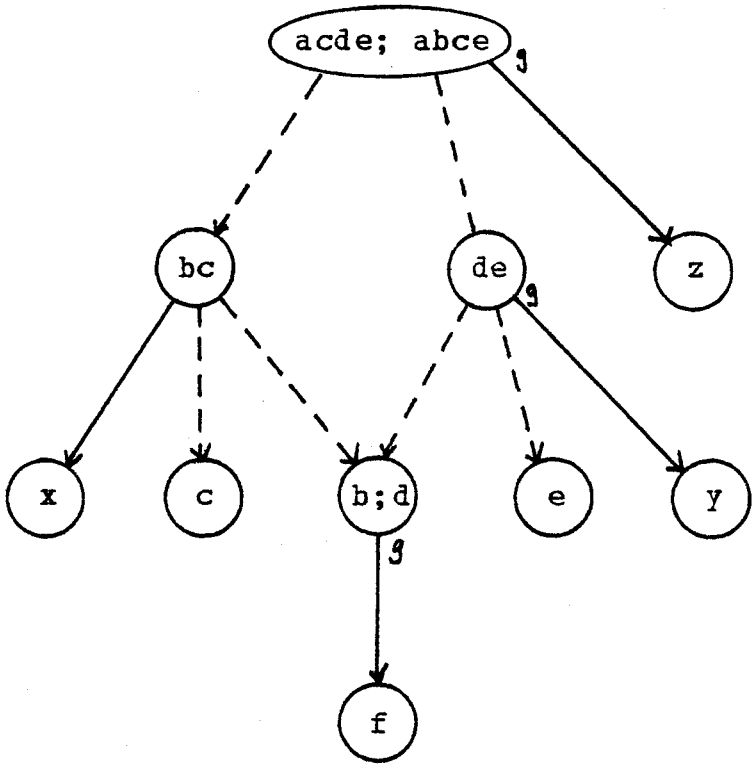
U F(i) by ab --> d; thus ab --> bd by projectivity and additivity.

In those cases in which there is a choice of primary key, the choice could be made according to other criteria, e.g. the key less likely to be unknown in the real world situation being modeled, or the key that requires the fewest bits for storage because these key values are stored two or more times (if they are the foreign key in more than one relation).

There are some cases in which it is impossible to have only one primary key. Consider Figure 5.11. As far as relation bcx is concerned, b should be the foreign key for bdf. But relation dey requires d as the foreign key. Thus there can be a conflict within a relation such as bdf over which key should be the primary key. We cannot add d to bcx nor add b to dey without introducing a transitive

Figure 5.11

A = {a, b, c, d, e, w, x, y, z}
F = {abcde --> z, bc --> x, de --> y, b --> d,
d --> b, b --> w}



dependency. Thus it may not be possible to have a unique primary key in all cases. The best solution, to avoid insertion anomalies, would be to have two primary keys, i.e. to insist that neither key can have an unknown value.

5.5 Retrieval from the third normal form collection

In this section we will suggest how the relational descriptions output by the third normal form algorithm can be used to answer a query involving an arbitrary subset of A. It was observed by Lochovsky and Tsichritzis that the most frequent logic error made by users of a relational data base concerned specifying the correct joins in a query [Lochovsky 77]. While the suggestions to be given below do not constitute an efficient algorithmic solution to this problem, they do provide a first step toward removing this responsibility from a user.

This solution is based on three assumptions. The first is that the query contains only restrictions (or Boolean expressions involving attribute names) and no projection or join operators. In other words, we are proposing that the user consider only the original table, and not be aware of the third normal form used to maintain the data base, in formulating queries.

The second assumption is that the method for answering a query is to join together everything mentioned in the query and, as a final step, project the resulting relation onto those attribute names mentioned in the conditions in the query. Obviously there will be many cases in which it is more efficient to do projection operations in the middle of the query-answering process and join together resulting relations each of which contains a smaller set of attribute

names directly related to the query. Since the algorithms to be given below will give all possible join sequences, this output could be used to see which of the possible sequences allow projection and extraction of tuples concerning parts of the query along the way. That is, this work should be thought of as a way of using the knowledge available from the fact that the relations are in covering, reconstructible third normal form to say where all the valid answers to a query can be found, and not as a complete query-answering method. The algorithms given are not efficient, but they might be considered acceptable because they deal only with the relational descriptions, whereas the actual retrieval involves manipulating the data base itself, and is thus typically far more costly.

Our third assumption is the partial information assumption which has motivated all the work on third normal forms; that is, not all information is available in all the relations in the collection. Recall the example of Figure 3.2 in which A = {p, j, d, l} and F = {p --> j, j --> l, p --> d, d --> l}. There may be some (p, l) pairs available by joining {p, j} * {j, l} and projecting onto {p, l} that are not available by doing {p, d} * {d, l} and projecting onto {p, l}, and vice versa. Thus to obtain all possible tuples satisfying a query, it may be necessary to process the query according to <u>all</u> possible join sequences and take the union of the results.

As a diagramatic aid we will use a modified graph G". This graph has one vertex for each A(i) extracted by the third normal form algorithm, with A(i) as its label and the primary key underlined. G" has two types of edges:

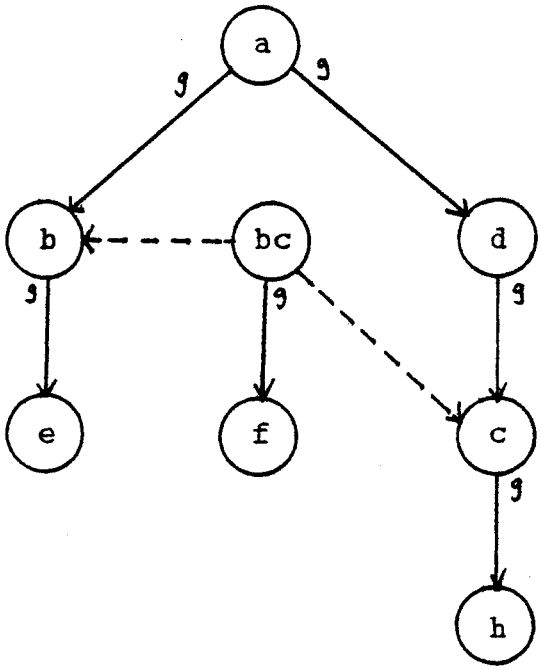
1. (B, C) is a type 1 edge in G" if the primary key for C is completely contained in B;
2. (B, C) is a type 2 edge if a proper subset of the primary key for C is contained in B.

We will call this the relation graph. An example of a relation graph for a system is given in Figure 5.12.

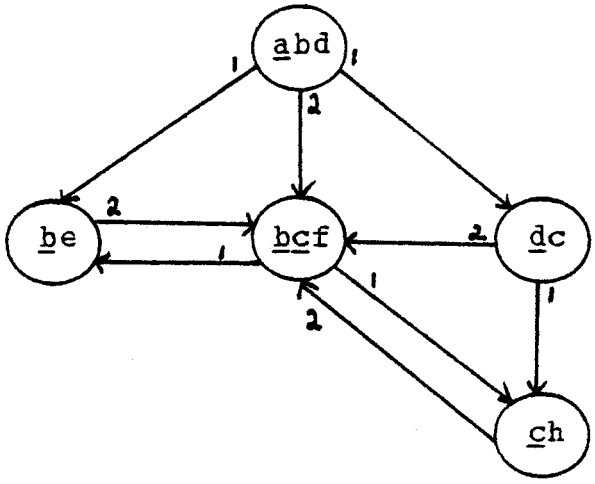
The theoretical aspects of joins have been examined by others in some detail [Rissanen 73, 77, Aho 77]. In particular, the join operation is associative and commutative. This does not mean, however, that we want to do the joins in an arbitrary order. It was observed earlier that if two relations with disjoint attribute name sets are joined together, then the result is the cartesian product of the two relations. Such a situation is called a lossy join [Aho 77] because, by creating "extra facts" or tuples, we have lost some information. The same is true if the intersection is non-empty, but is not a key for one of the relations involved. Thus the following result from [Rissanen 73,77, Aho 77] is the theoretical basis for this section. Let R and S be two relations with attribute names A and B respectively. Then R * S is non-loss if and only if A B --> A or A B --> B. As a corollary to this, if

Figure 5.12

A = {a, b, c, d, e, f, h}
F = {a --> b, a --> d, b --> e, bc --> f, d --> c, c --> h}



G+ for this (A, F)



relation graph G" for this (A, F)

there exists a sequence of joins such that the intersection of the attribute names always contains a key for the new relation being joined in, then a key of the first relation of the sequence is a key for any of the resulting relations, and the result of the whole sequence of joins is non-loss.

The work of Aho, Beeri and Ullman is related to this but not directly applicable. They give an algorithm to test if a set of relations has a non-loss join. Since the relations in our third normal form collection are reconstructible, they clearly have a non-loss join, given by the algorithm in Lemma 3.9. Their work also includes a test of whether a set of relations contains relations with a lossy join. Whereas this can happen with relations output by our algorithm (for example {a b d} * {b c f} in Figure 5.12), the purpose of the algorithms to be suggested is to make sure such lossy joins are never performed.

Considering the relations in Figure 5.12, suppose we wanted to answer a query relating attribute names a and f. By the above discusion, we cannot join {a, b, d} * {b, c, f} since b is not a key for either relation. However, if we do {a, b, d} * {d, c} * {b, c, f} in left to right order, then every join is over the key of one of the two relations involved. Since it is necessary to join over a key, we must always make sure, for a relation like {b, c, f} which has only incoming type 2 edges in the relation graph, that we have joined enough relations to have b and c among the

attribute names in the result. For {b, c, f}, this could be done in any of the following ways:

{a, b, d} * {d, c} * {b, c, f},

{a, b, d} * {b, e} * {d, c} * {b, c, f},

{a, b, d} * {d, c} * {c, h} * {b, c, f}.

We cannot, however, use any (b, c, f) tuples in the final join if we do not have both b and c known. For example, consider the following relations:

a	b	d
1	1	2
2	2	1
3	1	4

d	c
1	4
2	3
3	6

b	c	f
1	1	6
1	2	7
3	1	8

The join {a, b, d} * {d, c} gives the following:

a	b	c	d
1	1	3	2
2	2	4	1
3	1	?	4

There is no way we can associate the correct f value with a = 3 without knowing the correct c value. Joining in {c, h} before {b, c, f} does not help because we cannot validly associate the unknown value ? when it occurs in the primary key over which the join is being done. That is, the special value ? does not behave the way other values do under the join operator. Since we do not allow tuples for which primary key values are unknown, if they occur in the other relation, the tuple is discarded in forming the join.

The above discussion also demonstrates that joining in either {b, e} or {c, h} before {b, c, f} does not add any information about unknown b and c values. If possible, we want to avoid such "extra" joins.

In the collection of Codd third normal form relations, a relation X is an ancestor of relation Y if there is a non-loss join sequence starting with X and containing Y. Referring to the relation graph, this can happen in two ways: either the join sequence corresponds to a directed path consisting only of type 1 edges, or at some point in the join sequence a relation like {b, c, f} occurs which means that enough of the origins of the type 2 edges into {b, c, f} must precede it in the join sequence to cover the attribute names in its primary key.

A set of relations {R(i)} has a common ancestor N if N is an ancestor of each R(i). Note that in the collection output by our algorithm, all sets of relations have at least one common ancestor, namely the relation containing all the keys for A.

To summarize, the following characteristics are desired in a join sequence:

1. The join sequence must start with a common ancestor of all the relations taking part.
2. The ordering of a join sequence is governed by the fact that the primary key of the relation being added to the sequence must be contained in the attribute names collected

so far.

The following two additional rules concerning the treatment of unkncwn values when performing the joins should also be noted:

3. Unknown values, ?, may be kept in the relation resulting from the jcins unless they appear in the primary key over which a join is being done. When this happens, the whole tuple must be discarded.

4. When the intersection over which a join is being done properly contains the primary key of one relation, then any unknown values appearing in one of the relations in non-primary key columns in the intersection may be resolved if the value appears in the other relation.

Of course we are assuming that all data is correct so that if there is, say, more than one way to derive a tuple, they all agree.

The algorithms to be suggested include some preprocessing to be carried out before any queries are asked and some other processing to be done for each query. They are not meant to be rigorous but only to indicate how this might be done. Some open problems will be outlined.

The result of the preprocessing will be an ancestor matrix, which tells us whether or not relation i is an ancestor of relation j. As well, for each pair of relations, we will store a list of all valid join sequences from one to the other. For the time being we will assume we must

generate all permutations of a given set of relations each of which obeys rule 2 above.

The preprocessing uses the following data structures (r is the number of relations in the collection):

Ancestor(1::r, 1::r)	the ancestor matrix which at termination contains Ancestor(i, j) = 1 if relation i is an ancestor of relation j, and 0 otherwise.
Join Seqs(1::r, 1::r)	points to a list of sequences from relation i to relation j, for each i and j.
A(1::r)	attribute names in relation i, for all i = 1, r.
B	set of attribute names covered by the current join sequence.
Key(1::r)	primary keys for the relations
P	current join sequence

The notation Pj is used for the concatenation of P and j to form a new sequence.

```
Algorithm Join Sequences ({A(i)}, {Key(i)});
 procedure Search (B, P);
    for each unmarked relation j do
       if Key(j) ⊆ B then
          mark j;
          Ancestor(i,j) <-- 1;
          Join Seqs(i,j) <-- Join Seqs(i,j) U Pj;
```

```
            Search (B U A(j), Pj);
            unmark j
    for i <-- 1 step 1 until r do
        mark i;
        Search (A(i), i).
```

To answer a query, given the ancestor matrix and all the join sequences, the following algorithm could be used. Q contains the subset of A in the query.

```
Algorithm Answers (Q);
 C <-- all subsets of {A(i)} that cover Q;
 for each S in C do
    comment find all common ancestors of S;
    D <-- AND of the rows for S in Ancestor;
    for each common ancestor i in D do
       merge Join Seqs(i, k) for this i and all k in S
          in all possible ways obeying rule 2 above;
       output a set of join sequences for this common
          ancestor i and this covering set S.
```

Both the preprocessing and the query answering algorithms are exponential. After preprocessing, Join Seqs could contain almost all permutations of subsets of {A(i)} of any size, where finding one join sequence takes time polynomial in the number of relations. The set C could also be exponential in the number of relations. We could reduce the number of possible join sequences to being all possible combinations of relations if we could strengthen

rule 2 by saying that for each set of relations there is a unique ordering for the join sequence that guarantees we will get all the tuples that any other join sequence over the same relations would produce. This is certainly true if the relation graph contains only type 1 edges. A relation graph with only type 1 edges must be acyclic since, if it contained a cycle, then all of the primary keys on the cycle would functionally determine each other and thus would have been put in the same relation by the third normal form algorithm. If such a unique ordering can be found, then merging join sequences in the query answering algorithm is well defined and yields exactly one join sequence.

Even if we can resolve this problem for graphs with type 2 edges, we still need a good way of avoiding "extra" relations in a sequence. If one sequence for going from an ancestor to a given relation properly contains another sequence for the same pair, then are the extra relations always unnecessary? For the example of Figure 5.12, in the join sequence {a, b, d} * {d, c} * {c, h} * {b, c, f}, relation {c, h} is unnecessary, but is this always the case? Even if this can be resolved, however, how do we prevent the first step in the query answering algorithm from choosing {b, e} and {c, h} as a member of $\underline{C}$ for the query Q = {b, c}? Although we know for this example that joining in {c, h} can never add any c values, is this

necessarily the case when one join sequence properly contains another?

If we could solve all of these problems, then the join sequences could be used in the following ways:

- If only one join sequence is wanted, or the incomplete information assumption does not hold, then choose a single, most efficient jcin sequence according to some other criterion; e.g. either the ease with which projection can be used to reduce the work involved, or, perhaps, because of the range of values present in the data, one join sequence might result in a smaller number of tuples in all the intermediate relations, etc.
- If the incomplete information assumption is valid, and all possible tuples to answer a query are wanted, then, for a given ancestor, merge (uniquely) all sequences from each ancestor, even if they are for different members of $\underline{C}$. This typically will result in very few join sequences (one for each ancestor) and will give all valid ways of associating the attribute names in a query.

Chapter 6

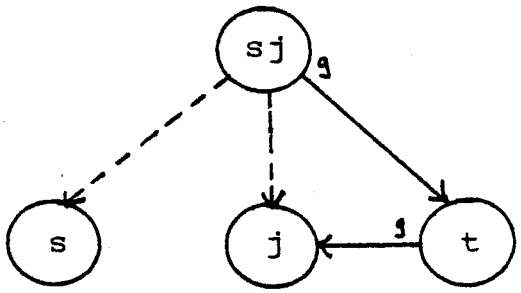
Covering Kent third normal forms

6.1 Examples

We begin this chapter with a number of examples to illustrate the difficulty of finding covering Kent third normal forms or even determining when they exist.

Consider again the standard example: A = {s, j, t} and F = {sj --> t, t --> j}. Figure 6.1 shows the subset of the equivalent-graph closure on vertices whose labels are

Figure 6.1



left-hand sides or elements of A. Recall that st is also a key. Since we no longer have the luxury of ignoring the behaviour of prime attribute names such as j, we must consider the projectivity edges as well as the given edges. If transitive reduction were performed on the projectivity and given edges, the projectivity edge from sj to j would be deleted. This captures the dilemna of a covering Kent third normal form not existing. In order for sj --> t to be covered, it must be embodied in a relation; but whenever

s, j and t are put together, t --> j is also present and this triple can never be in Kent third normal form with respect to these dependencies.

The next example, originally suggested by Bernstein and Beeri [Bernstein 76c], shows that sometimes a covering Kent third normal form can be found by choosing one covering over another. Let A = {a, b, c, d, e} and F = {a --> b, a --> c, bc --> a, ad --> e, bcd -->e, e --> c}. Figure 6.2 shows the subset of the equivalent-graph closure on vertices whose labels are left-hand sides or elements of A. Graphs such as these are shown in this section for the purpose of illustration only. It will be shown below that we cannot consider the acyclic equivalent graph for Kent third normal forms, as merging any equivalent keys is too likely to cause Kent third normal form violations.

The following relational descriptions constitute a covering third normal form collection for this example:

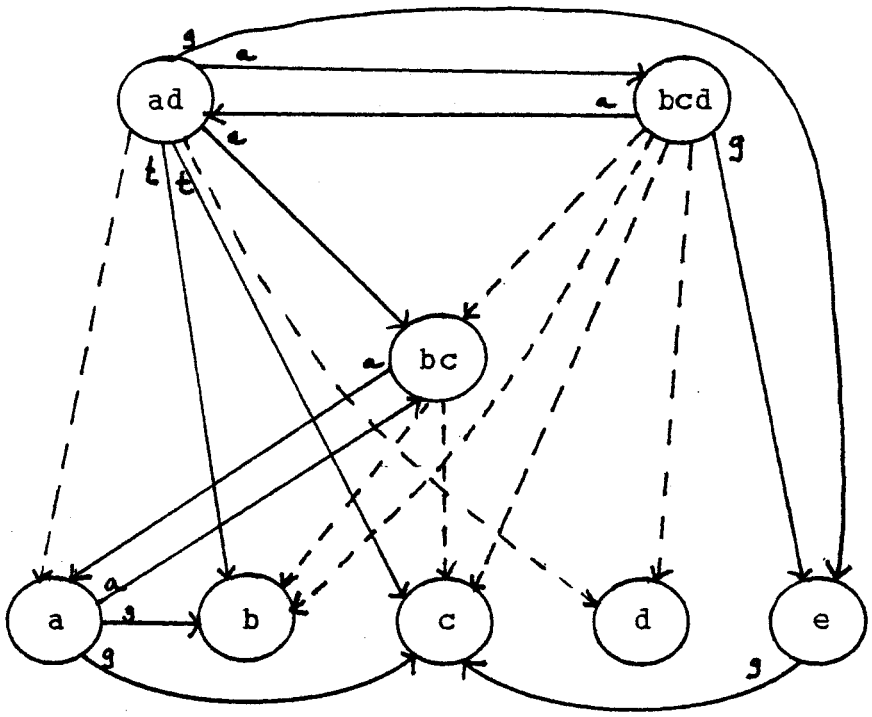
A(1) = {a, d, e} F(1) = {ad --> e}

A(2) = {a, b, c} F(2) = {a --> b, a --> c, bc --> a}

A(3) = {c, e} F(3) = {e --> c}.

If we replace (A(1), F(1)) by A(1)' = {b, c, d, e}, F(1)' = {bcd --> e, e --> c} then we still have a minimal covering of the functional dependencies but (A(1)', F(1)') is not in Kent third normal form. Under Codd third normal form, a relaticn embodying a single functional dependency is always in third normal form, and thus any minimal

Figure 6.2



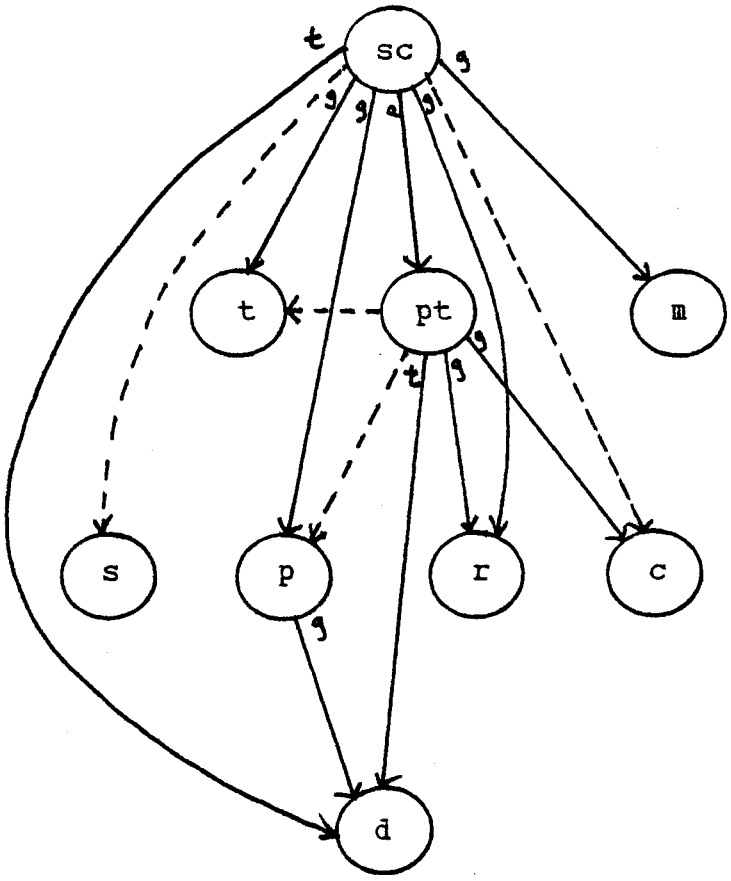
covering leads to a third normal form collection, alghough not necessarily an optimum one.

There are other disturbing points illustrated by this example. Under Codd third normal form, equivalent subsets of A could always coexist in a single relation. For this example, ad and bcd cannot be in the same relation because, as only one of several Kent third normal form violations, attribute name a is functionally dependent on bc which is not a minimal key. A designer might find it unacceptable that such equivalent keys cannot coexist; it should always be possible to store a one-to-one relationship in one relation.

The other disturbing thing shown by this example is that, whereas with Codd third normal form we were guaranteed to find a covering third normal form (although not necessarily optimum) without finding all the keys (by using Bernstein's algorithm), for Kent third normal form this is not the case. If ad --> e is deleted from F, the closure of the functional dependencies is the same. As we saw above, this would lead to the collection with (A(1)', F(1)') in place of (A(1), F(1)). A covering Kent third normal form exists for this example but to find it we would have to generate all keys (knowing they might not end up together in a relation as with the algorithm in Section 5.3) and possibly test all combinations of different keys for different relations. Thus it seems that finding Kent third normal forms may require examining all minimal coverings.

Our final example shows that even the simplest type of optimization for the covering approach, merging equal left-hand sides, cannot always be done. Let A = {s, c, m, r, p, t, d} and F = {sc --> m, sc --> p, sc --> r, sc --> t, pt --> r, pt --> c, p --> d}. Figure 6.3 shows the subset of the equivalent graph closure on vertices whose labels are left-hand sides or elements of A. The subset with attribute names {s, c, p, r, m} is in Kent third normal form but adding t introduces pt --> c and pt --> r, and pt is thus a left-hand side that is not a key. Similarly {s,

Figure 6.3



c, t, r, m} can exist together but p cannot be added to this set. Thus even when a covering Kent third normal form exists, finding an optimum collection may require examining not only all coverings, but all possible ways to merge equal left-hand sides, as well as possible ways to merge equivalent left-hand sides.

6.2 Testing for existence of a covering Kent third normal form

In this section we give an algorithm for determining, for a given (A, F), whether or not there exists a covering Kent third normal form. The test is based on the following two lemmas.

Lemma 6.1: A system (A, F) is in Kent third normal form iff every left-hand side in F is a key (not necessarily minimal).

proof: It follows from the definition (Boyce-Codd wording) that, for a relation to be in Kent third normal form, every left-hand side must be a key. However, the definition is stated in terms of the closure $\underline{F}(A, F)$ and thus the converse is not immediate. Assume that every left-hand side in F is a key but that (A, F) is not in Kent third normal form. Thus there is a transitive dependency in the closure of the form Key --> X --> y, y $\notin$ Key, X $\not\rightarrow$ Key, y $\in$ X. Thus X is $\underline{F}$-expansible and therefore, by Lemma 4.7, X or a proper subset of X is a left-hand side in F. But X is not a key since it cannot derive Key. This contradicts the assumptions. □

The above lemma provides a test for Kent third normal form providing we know F and thus the basis of the closure over the attribute names in question. The problem is that in a larger system A, when we want to know if a subset A(i) and some functional dependencies F(i) constitute a third

normal form relation, we do not know all the potential left-hand sides which might be expansible, i.e. F(i) may not generate the same closure over A(i) as F does. For example, let A = {a, b, w, y, z} and F = {a --> y, x --> z, ax --> b}. Suppose we are considering {a, x, b}. Then the fact that a and x are left-hand sides in F does not mean they are expansible with respect to {a, x, b}. If we add z --> b to F, then x is a left-hand side that is expansible with respect to {a, x, b} but the functional dependencies used to derive this are not contained in {a, x, b}. Thus Bernstein and Beeri found that the question "does a subset of A constitute the attribute names for a Kent third normal form relation?" is NP-complete in |A| and |F| [Bernstein 76c].

Lemma 6.2: There exists a covering Kent third normal form for a given relational system (A, F) iff there exists a covering Kent third normal form consisting of canonical full functional dependencies. (By this we mean that if L --> r is a canonical full functional dependency, then one relational description in the collection is given by: A(i) = {L} U {r} and F(i) consists of any functional dependencies in the closure whose left- and right-hand sides are contained in {L} U {r}).

proof: Let a covering Kent third normal form contain a relational description (A(i), F(i)) where F(i) contains L --> r(1), L --> r(2) for some L. Then our claim is that if

(A(i), F(i)) contains no Kent third normal form violations, neither will ({L} U {r(i)}, L --> r(i) U any other functional dependencies over {L} U {r(i)}), for any such L and r(i). Since (A(i), F(i)) is in Kent third normal form, then every left-hand side J in F(i) has the property J --> A(i), i.e. it is a key for A(i). Clearly if J $\subseteq$ {L} U {r(i)} then J --> {L} U {r(i)}. The proof of the converse is obvious. □

Corollary: The algorithm for testing the existence of a covering Kent third normal form is:

Algorithm Kent third normal form existence (A, F);

```
1.  Comment generate the closure F of (A, F);
    for all X, Y in 2**A do
       if Membership (F, X --> Y )
          then F <-- F U {X --> Y};
2.  for each canonical functional dependency L --> r
     in the closure do
       if L --> r is a full functional dependency
          then A(i) <-- {L} U {r};
               F(i) <-- all functional dependencies in
                 the closure over A(i);
               Kent <-- "true";
               for all functional dependencies L(j) --> R(j)
                 in F(i) do
                  if ¬ Key (A(i), F, L(j))
                     then Kent <-- "false";
```

```
            if Kent then mark L --> r;
3.  Comment find the closure F' of the marked full functional
      dependencies;
    F' <-- {marked functional dependencies};
    for all X, Y in 2**A do
       if Membership (F', X --> Y)
          then F' <-- F' U {X --> Y};
4.  if F = F'
       then a covering Kent third normal form exists.
```

Corollary: If one of the marked canonical full functional dependencies has a left-hand side that is a key for all of A, then there exists a reconstructible covering Kent third normal form collection for (A, F).

Lemma 6.3: The Kent third normal form existence algorithm runs in time 0 ($|A|^2 |F| (2^{**}|A|)^3$).

proof: Step 1 using Bernstein and Beeri's membership algorithm takes $0((2^{**}|A|)^2 |A| |F|)$ operations. There are at most $|A|$ $2^{**}|A|$ canonical functional dependencies. To check if each one is a full functional dependency takes at most $|A|$ calls of membership. For each such functional dependency, determining F(i) requires looking at all the $0((2^{**}|A|)^2)$ functional dependencies in the closure. The expression $0((2^{**}|A|)^2)$ is also the bound on $|F(i)|$. Determining if a left-hand side is a key takes $|A(i)| |F| \leq |A| |F|$ operations. Thus, in total, step 2 takes $0(|A| 2^{**}|A| (|A|^2 |F| + ((2^{**}|A|)^2 + |A| |F| (2^{**}|A|)^2))$ or

$O(|A|^2 \ |F| \ (2^{**}|A|)^3)$ operations. Step 3 involves $O((2^{**}|A|)^2)$ calls of the membership algorithm with a set of functional dependencies of size $O(2^{**}|A|)$. Thus step 3 requires $O(|A| \ (2^{**}|A|)^3)$ operations. Step 4 takes $O((2^{**}|A|)^2)$ operations. In total, then, the algorithm requires at most $O(|A|^2 \ |F| \ (2^{**}|A|)^3)$ operations. □

An optimum covering Kent third normal form collection will clearly be one in which a minimal covering for the marked canonical full functional dependencies has been found. It is certainly possible to find such an optimum collection by an exhaustive enumeration of all such minimal coverings, merging equil and equivalent left-hand sides in all possible combinations. However, such an algorithm does not have a time bound which we consider to be acceptable. As we saw in the examples in Section 6.1, none of the known optimization techniques used to find optimum covering Codd third normal form collections are guaranteed to work here. Thus the task of efficiently finding optimum covering Kent third normal forms appears to be intractible.

7. Conclusions

This thesis examines several ways of solving the integrity problems identified in a relational data base by the definition of functional dependencies on the data base. After comparing covering, decomposition and third normal form approaches, it is shown that the most desirable form for a relational data base is a covering, reconstructible, Kent third normal form. However, since such a form does not always exist, covering, reconstructible Codd third normal forms are also shown to be acceptable.

Finding all the minimal keys of a relation is necessary for the Codd third normal form algorithm. A number of problems related to key finding are examined, including establishing upper bounds on the number of keys for a relational description both in terms of the number of functional dependencies given and in terms of the number of attribute names in the relaion. As well, it is shown that two related problems are NP-complete: determining whether or not an attribute name is prime, and deciding whether or not there is a key with fewer than m attribute names, for a given parameter m. An algorithm is then given which finds all minimal keys for a given subset of the attribute names in time polynomial in $|A|$, $|F|$ and the number of keys found. Thus when there are only a few keys, this algorithm gives an efficient method for finding them.

The key finding algorithm is then incorporated into an

algorithm which finds a covering, reconstructible, Codd third normal form collection for a given (A, F). This collection differs from that output by other algorithms because every relation in the collection contains all its minimal keys. We then show how this collection might be used automatically to generate join sequences to answer an arbitrary query such that all the joins are non-loss and thus valid as far as the functional dependencies are concerned. We also discuss possible choices for primary keys.

Finally, a test is given to determine, for a given (A, F), whether or not there exists a covering Kent third normal form. Some of the problems of finding an optimum Kent third normal form when one exists are also discussed.

In addition to the specific results mentioned above, several important ways in which our approach differs from others are in emphasising reconstructibility and in stressing the incomplete information assumption. The former allows us to say more about answering queries, because we know there is always at least one common ancestor from which to start, namely the relation containing the keys for all of A. The latter tells us that if there is more than one join sequence for a query, and if we want all possible tuples which answer the query, we must take the union of all these join sequences.

Some open problems are introduced in Section 5.5. In

generating join sequences for a query, is there a unique ordering for a given sequence such that the resulting joins produce all the tuples any other sequence over the same relations would produce? It would seem that there would be one such sequence for each ancestor, since, in Figure 5.12, if {a, b, d} is the ancestor, then {c, h} should precede {b, c, f} in any sequence they share, but if {b, c, f} is the ancestor, then clearly it must precede {c, h}.

Another problem introduced in Section 5.5 is how to recognize unnecessary relations in a join sequence. An answer to this question could also be used to trim user-generated join sequences.

Of more theoretical interest is the following problem. In finding optimum Codd third normal forms, no matter what the cardinality of the minimal covering used, the cardinality of the resulting collection of covering Codd third normal form relations is the same. Is this also true for optimal covering Kent third normal forms?

A more general extension of the whole thesis would be to examine the feasibility of efficient algorithms for Date third normal forms or Rissanen's atomic relations which are irreducible, reconstructible and cover the functional dependencies. The basic objectives of this work could be altered to make the retention of update units more important than covering or third normal form, and algorithms sought under these constraints. This extends to

the problem of finding a covering third normal form for a system (AUB, F) where attrubute names from A and B cannot be mixed. A solution to this problem might have relevance for a data base in a computer network environment, where, for example, information corresponding to attribute names in A is the property of one company or department on the network and information for B is the property of another, but answering queries relating to A U B is desired.

References

Adiba 76
Adiba, M., Leonard, M., and Delobel, C. A unified approach for modelling data in logical data base design, IFIP Working Conference, Modelling in Data Base Management Systems, G.M. Nijssen (Ed.), North Holland, (1976), 311-338.

Aho 72
Aho, A.V, Garey, M.R. and Ullman, J.D. The transitive reduction of a directed graph, SIAM J. Comput. 1, 2 (June 1972), 131-137.

Aho 77
Aho, A.V., Beeri, C., and Ullman, J.D. The theory of joins in relational data bases, to appear, FOCS, 1977.

ANSI/SPARC
ANSI/X3/SPARC, Study Group on Data Base Management Systems, Interim Report, 75-02-08, ACM SIGMOD newsletter, 7, 2 (1975).

Amstrong 74
Armstrong, W.W. Dependency structures of data base relationships, IFIP 74, North-Holland, (1974), 580-583.

Beeri 77
Beeri, C., Fagin, R., and Howard, J.L., A Complete Axiomatization for Functional and Multivalued Dependencies in Database Relations, IBM Res. Rep. RJ1977, (Apr. 1977).

Bernstein 75a
Bernstein, P.A. Normalization and Functional Dependencies in the Relational Data Base Model, Ph. D. dissertation, University of Toronto, (1975).

Bernstein 75b
Bernstein, P.A., Swenson, J.R., and Tsichritzis, D.C. A unified approach to functional dependencies and relations, ACM SIGMOD International Conference on Management of Data, (May 1975), 237-245.

Bernstein 76a
Bernstein, P.A., Comment on "Segment synthesis in logical data base design", IBM J. of Res. & Dev. 20, 4 (July 1976), 412.

Bernstein 76b
Bernstein, P.A. Synthesizing third normal form relations from functional dependencies, ACM Transactions on Database Systems 1, 4 (Dec. 1976), 277-298.

Bernstein 76c
Bernstein, P.A., and Beeri, C., An Algorithmic Approach to Normalization of Relational Data Base Systems, CSRG 73, University of Toronto, (Sept. 1976).

Boyce 73
Boyce, R.F. Fourth Normal Form and its Associated Decomposition Algorithm, IBM Technical Disclosure Bulletin 16, (1973), 30.

Codd 70
Codd, E.F. A relational model for large shared data banks, Comm ACM 13, 6 (June 1970), 377-387.

Codd 71a
Codd, E.F. Further normalization of the relational data base model, Courant Computer Science Symposium 6, Data Base Systems, R. Rustin (Ed.), Prentice-Hall, (1971), 33-64.

Codd 71b
Codd, E.F. Normalized data base structure: a brief tutorial, ACM SIGFIDET Workshop on Data Description, Access and Control, (Nov. 1971), 1-17.

Codd 74
Codd, E.F. Recent investigations in relational data base systems, IFIP 74, North-Holland, 1017-1021.

Cook 71
Cook, S.A., The complexity of theorem proving procedures, Proc. 3rd Annual ACM Symposium on Theory of Computing, (1971), 151-158.

Date 75
Date, C.J. An Introduction to Database Systems, Addison-Wesley Publishing Co., Reading Mass., (1975).

Date 77
Date, C.J. An Introduction to Database Systems, second edition, Addison-Wesley Publishing Co., Reading Mass., (1977).

Delobel 72
Delobel, C. A theory about data in an information system, IBM Res. Rep. RJ964, (Jan. 1972).

Delobel 73a
Delobel, C. Contributions theoriques a la conception et l'evaluation d'un systeme d'informations applique a la gestion, Ph. D. dissertation, L'Universite Scientifique et Medicale de Grenoble, (1973).

Delobel 73b

Delobel, C. and Casey, R.G. Decomposition of a data base and the theory of Boolean swiching functions, IBM J. of Res. & Dev. 17, 5 (Sept. 1973), 374-386.

Delobel 74
Delobel, C., and Leonard, M. The decomposition process in a relational model, University of Grenoble, Laboratory Informatique, (Sept. 1974).

Even 73
Even, S. Algorithmic Combinatorics, the Macmillan Company, New York, (1973).

Fadous 75
Fadous, R., and Forsyth, J., Finding candidate keys for relational data bases, ACM SIGMOD International Conference on Management of Data, (May 1975), 203-210.

Fagin 76a
Fagin, R. Dependency in a Relational Database and Propositional Logic, IBM Res. Rep. RJ1776, (Apr. 1976).

Fagin 76b
Fagin, R. Multivalued Dependencies and a New Normal Form for Relational Databases, IBM Res. Rep. RJ1812, (1976).

Fagin 77
Fagin R., The Decomposition Versus the Synthetic Approach to Relational Database Design, IBM Res. Rep. RJ1976, (Apr. 1977).

Heath 71
Heath, I.J. Unacceptable File Operations in a Relational Data Base, ACM SIGFIDET Workshop on Data Description, Access and Control, (Nov. 1971), 19-33.

Karp 72
Karp, R.M., Reducibility among combinatorial problems, Complexity of Computer Computations, R.E. Miller and J.W. Thatcher (Ed.), Plenum Press, New York, (1972), 85-103.

Kent 73
Kent, W. A Primer of Normal Forms, IBM Systems Development Division, TR02.600, (Dec. 1973).

Lochovsky 77
Lochovsky, F.H., and Tsichritzis, D.C. User performance considerations in DBMS selection, ACM SIGMOD International Conference on Management of Data, (Aug. 1977), 128-134.

Lucchesi 77

Lucchesi, C.L., and Osborn, S.L., Candidate keys for relations, to appear, JCSS.

Rissanen 73
Rissanen, J. and Delobel, C. Decomposition of Files, a Basis for Data Storage and Retrieval, IBM Res. Rep. RJ1220, (May, 1973).

Rissanen 77
Rissanen, J. Independent components of relations, IBM Res. Rep. RJ1899, (Jan. 1977).

Schenk 77
Schenk, K.L., and Pinkert, J.L. An algorithm for servicing multi-relational queries, ACM SIGMOD International Conference on Management of Data, (Aug. 1977), 10-20.

Wang 75
Wang, C.P. and Wedekind, H. Segment synthesis in logical data base design, IBM J. of Res. & Dev. 19, 1 (Jan. 1975), 71-77.

Wiederhold 76
Wiederhold, G. Comment on "Segment Synthesis in logical data base design", IBM J. of Res. & Dev. 20, 3 (May 1976), 290.

Yu 76
Yu, C.T., and Johnson, D.T., On the complexity of finding the set of candidate keys for a given set of functional dependencies, Info. Proc. Letters, 5, 4 (1976), 100-101.

Zaniolo 76
Zaniolo, C. Analysis and Design of Relational Schemata for Database Systems, Ph. D. dissertation, UCLA, UCLA-ENG-7669, (July 1976).