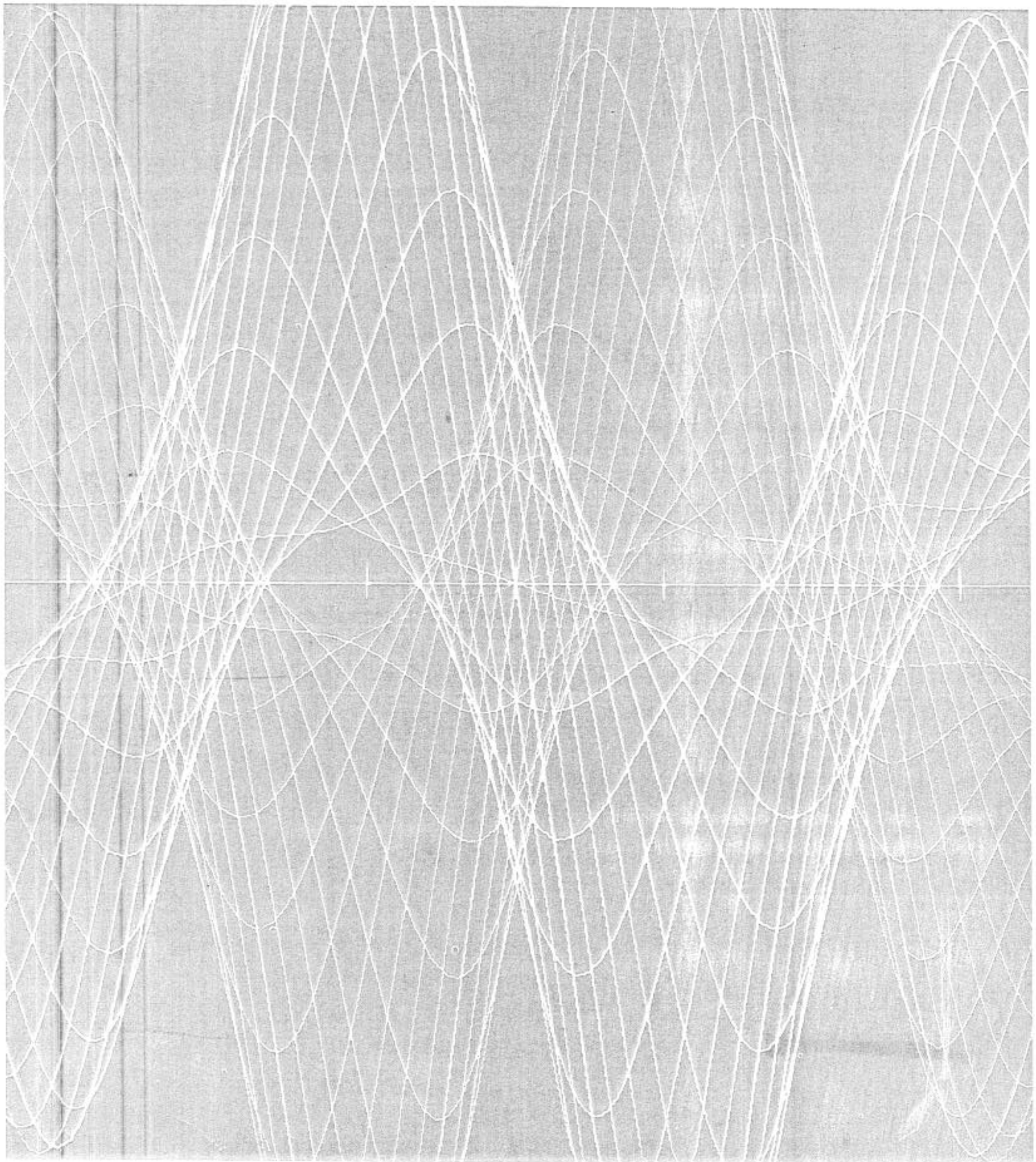

RJ 599

DERIVABILITY, REDUNDANCY AND CONSISTENCY OF RELATIONS STORED IN LARGE DATA BANKS

E. F. Codd

August 19, 1969

IBM RESEARCH

The cover is designed from an example of an interaction between computers and the physical and mathematical sciences. The design depicts ion trajectories in a type of mass spectrometer used for chemical analysis of residual gases in ultra high vacuum systems. These ion trajectories represent solutions of Mathieu's differential equation. They are generated by numerical integration of the equation using a high speed computer, and are plotted automatically from an output tape as part of the Research Center computing service.

Printed in the U.S.A.

DERIVABILITY, REDUNDANCY AND CONSISTENCY OF RELATIONS STORED IN LARGE DATA BANKS

E. F. Codd
Research Division
San Jose, California

ABSTRACT: The large, integrated data banks of the future will contain many relations of various degrees in stored form. It will not be unusual for this set of stored relations to be redundant. Two types of redundancy are defined and discussed. One type may be employed to improve accessibility of certain kinds of information which happen to be in great demand. When either type of redundancy exists, those responsible for control of the data bank should know about it and have some means of detecting any "logical" inconsistencies in the total set of stored relations. Consistency checking might be helpful in tracking down unauthorized (and possibly fraudulent) changes in the data bank contents.

RJ 599(# 12343)
August 19, 1969

CONTENTS

INTRODUCTION

The first part of this paper is concerned with an explanation of a relational view of data. This view (or model) of data appears to be superior in several respects to the graph or network model [1, 2] presently in vogue. It provides a means of describing data with its natural structure only: that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level retrieval language which will yield maximal independence between programs on the one hand, and machine representation and organization of data on the other. A further advantage of the relational view is that it forms a sound basis for treating derivability, redundancy, and consistency of relations -- these are discussed in the second part of this paper. The network model, on the other hand, has spawned a number of confusions, not the least of which is mistaking the derivation of connections for the derivation of relations. Finally, the relational view permits a clearer evaluation of the scope and logical limitations of present management information systems, and also the relative merits (from a logical standpoint) of competing representations of data within a single system.

1. A Relational View of Data

The term relation is used here in its accepted mathematical sense. Given sets $S_1, S_2, \ldots, S_n$ (not necessarily distinct), R is a relation on these n sets if it is a set of n-tuples, each of which has its first element from S_1, its second element from S_2, and so on. We shall refer to S_j as the j^{th} domain of R. As defined above, R is said to have degree n. Relations of degree 1 are often called unary, degree 2 binary, degree 3 ternary, and degree n n-ary.

For expository reasons, we shall frequently make use of an array representation of relations, but it must be remembered that this particular representation is not an essential part of the relational view being expounded. An array which represents an n-ary relation R has the following properties:

(1) Each row represents an n-tuple of R;

(2) The ordering of rows is immaterial;

(3) All rows are distinct;

(4) The ordering of columns is significant - it corresponds to the ordering $S_1, S_2, \ldots, S_n$ of the domains on which R is defined;

(5) The significance of each column is partially conveyed by labeling it with the name of the corresponding domain.

The example in Figure 1 illustrates a relation of degree 4 called ship which reflects the shipments-in-progress of parts from specified suppliers to specified projects in specified quantities.

ship (supplier	part	project	quantity)
1	2	5	17
1	3	5	23
2	3	7	9
2	7	5	4
4	1	1	12

FIGURE 1: A Relation of Degree 4

One might ask: If the columns are labeled by the name of the corresponding domains, why should the ordering of columns matter? As the example in Figure 2 shows, two columns may have identical headings (indicating identical domains), but possess distinct meanings with respect to the relation. The relation depicted is called component. It is a binary relation, each of whose two domains is called part. The meaning of component (x, y) is that part x is an immediate component (or subassembly) of part y.

component (part	part)
1	5
2	5
3	5
2	6
3	6
4	7
6	7

Figure 2: A Relation with Two Identical Domains

We now assert that a data bank is a collection of time-varying relations. These relations are of assorted degrees. As time progresses, each n-ary relation may be subject to insertion of additional n-tuples, deletion of existing ones, and alteration of components of any of its existing n-tuples.

Consider, for example, a data bank which contains information about parts, projects, and suppliers. The individual description for an individual object (such as a particular part) is called an entity [3]. The prototype description for a class of objects is called an entity type. The set of entities of a given entity type can be viewed as a relation, and we shall call such a relation an entity type relation. In the example under consideration, there might be an entity type relation called part defined on the following domains:

(1) part number

(2) part name

(3) part color

(4) part weight

(5) quantity on hand

(6) quantity on order

and possibly other domains as well. Each of these domains is, in effect, a pool of values, some or all of which may be represented in the data bank at any instant. While it is conceivable that, at some instant, all part colors are present, it is unlikely that all possible part weights, part names, and part numbers are. The domains listed above correspond to what are commonly called the <u>attributes</u> of the entity type part.

Normally, one attribute (or combination of attributes) of a given entity type has values which uniquely identify each entity. Such an attribute (or combination) is called a <u>key</u>. In the example above, part number would be a key, while part color would not be. A key is <u>non-redundant</u> if it is either a simple attribute (not a combination) or a combination such that none of the participating attributes is superfluous in uniquely identifying each entity. An entity type may possess more than one non-redundant key. This would be the case in the example, if different parts were always given distinct names.

The remaining relations in a data bank are between entity types, and are, therefore, called <u>inter-entity relations</u>. An essential property of every inter-entity relation is that its domains include at least two keys which either refer to distinct entity types or refer to a common entity type serving distinct roles.

The examples in Figures 1 and 2 will help clarify this. The relation exhibited in Figure 1 involves three keys, one for each of the entity types <u>supplier</u>, <u>part</u>, <u>project</u>. The relation exhibited in Figure 2 involves two keys referring to the common entity type part, the first key serving to identify a component, the second to identify an assembly containing that component. Both of these relations are inter-entity relations.

So far, we have discussed examples of relations which are defined on simple domains - domains whose elements are atomic (non-decomposable) values. Non-atomic values can be discussed within the relational framework. Thus, some domains may have relations as elements. These relations may, in turn, be defined on non-simple domains, and so on. For example, one of the domains on which the entity type relation <u>employee</u> is defined might be <u>salary history</u>. An element of the salary history domain is a binary relation defined on the domain <u>date</u> and the domain <u>salary</u>. The salary history domain is the set of all such binary relations.

2. Some Linguistic Aspects

The adoption of a relational view of data, as described above, permits the development of a universal retrieval sublanguage based on the second-order predicate calculus.* Such a language would provide a yardstick of linguistic power for all other proposed retrieval languages, and would itself be a strong candidate for embedding (with appropriate syntactic modification) in a variety of host languages (programming, command or problem oriented). While it is not the purpose of this paper to describe such a language in detail, its salient features would be as follows.

Let us denote the retrieval sublanguage by R and the host language by H. R permits the declaration of domains, together with relations of various degrees on those domains. H permits supporting declarations which indicate, perhaps less permanently, how these relations are represented in storage. R permits the specification for retrieval of any subset of data from the data bank. Action on such a retrieval request is subject to security constraints.

The class of qualification expressions which can be used in a set specification is in a precisely specified one-to-one correspondence with the class of well-formed formulas of the predicate calculus in prenex normal form [4]. Any arithmetic functions needed can be defined in H and invoked

*The second-order predicate calculus (rather than first-order) is needed because the domains on which relations are defined may themselves have relations as elements (see section 1).

in R. A set so specified may be fetched for query purposes only or it may be held for possible changes. Insertions take the form of adding new elements to declared relations without regard to any ordering that may be present in their machine representation. Deletions which are effective for the community (as opposed to the individual user or sub-communities) take the form of removing elements from declared relations. Some deletions may be triggered by others, if deletion dependencies between specified relations are declared in R.

One important effect that the view adopted toward data has on the language used to retrieve it is in the naming of data elements and sets. With the usual network view, users will often be burdened with coining and using more relation names than are absolutely necessary, since names are associated with directed paths rather than with relations. Two such paths are needed to support symmetric exploitation* of a single binary relation. For a relation of degree n, the number of paths to be named and controlled is factorial n.

Again, if a relational view is adopted in which every n-ary relation ($n > 2$) has to be expressed by the user as a nested

*Once a user is aware that a certain relation is stored, he will expect to be able to exploit it using any combination of its arguments as "knowns" and the remaining arguments as "unknowns," because the information (like Everest) is there. This is a system feature (missing from many current information systems), which we shall call (logically) <u>symmetric exploitation</u> of relations.

expression involving only binary relations, then $2n-1$ names have to be coined instead of only $n+1$ with direct n-ary notation as described in Section 1. For example, the 4-ary relation ship of Figure 1, which entails 5 names in n-ary notation, would be represented in the form

P (supplier, Q (part, R (project, quantity)))

in nested binary notation and, thus, employ 7 names.

3. Operations on Relations

Since relations are sets, all of the usual set operations are applicable to them. Nevertheless, the result may not be a relation; for example, the union of a binary relation and a ternary relation is not a relation.

The operations discussed below are specifically for relations. These operations are introduced because of their key role in deriving relations from other relations. Most users would not be directly concerned with these operations. Information systems designers and people concerned with data bank control should, however, be thoroughly familiar with these operations.

3.1 Permutation

A binary relation has an array representation with two columns. Interchanging these two columns yields the converse relation. More generally, if a permutation is applied to the columns of an n-ary relation, the resulting relation is said to be a permutation of the given relation. There are, for example, $4! = 24$ permutations of the relation ship in Figure 1, if we include the identity permutation which leaves the ordering of columns unchanged.

In a system which provides symmetric exploitation of relations, the set of queries answerable by a stored relation is identical to the set answerable by any permutation of that relation. Although it is logically unnecessary to store both a relation and some permutation of it, performance considerations could make it advisable.

3.2 Projection

Suppose now we select certain columns of a relation (striking out the others) and then remove from the resulting array any duplication in the rows. The final array represents a relation which is said to be a projection of the given relation.

A selection operator Π is used to obtain any desired permutation, projection, or combination of the two operations. Thus, if L is a list of k indices

$$L = i_1, i_2, \ldots, i_k$$

and R is an n-ary relation $(n \geq k)$, then $\Pi_L(R)$ is the k-ary relation whose j^{th} column is column i_j of R $(j = 1, 2, \ldots, k)$ except that duplication in resulting rows is removed. Consider the relation ship of Figure 1. A projection of this relation is exhibited in Figure 3.

Π_{31} (ship)	(project	supplier)
	5	1
	5	2
	1	4
	7	2

Figure 3: A Permuted Projection of the Relation in Figure 1

Note that, in this particular case, the projection has fewer n-tuples than the relation from which it is derived.

3.3 Join

Suppose we are given two binary relations, which have some domain in common. Under what circumstances can we combine these relations to form a ternary relation which preserves all of the information in the given relations?

The example in Figure 4 shows two relations R, S, which are joinable without loss of information, while Figure 5 shows a join of R with S. A binary relation R is joinable with a binary relation S if there exists a ternary relation U such that $\Pi_{12}(U) = R$ and $\Pi_{23}(U) = S$. Any such ternary relation is called a join of R with S. If R, S are binary relations such that $\Pi_2(R) = \Pi_1(S)$, then R is joinable with S. One join that always exists in such a case is the natural join of R with S defined by

$$R*S = \{(a, b, c):R(a, b) \wedge S(b, c)\}$$

where R(a, b) has the value true if (a, b) is a member of R and similarly for S(b, c). It is immediate that

$$\Pi_{12}(R*S) = R$$

and

$$\Pi_{23}(R*S) = S$$

Note that the join shown in Figure 5 is the natural join of R with S from Figure 4. However, this join is not the only one of R with S. Figure 6 shows another possible join of the relations in Figure 4.

R (supplier	part)		S (part	project)
1	1		1	1
2	1		1	2
2	2		2	1

Figure 4: Two Joinable Relations

R*S (supplier part project)

supplier	part	project
1	1	1
1	1	2
2	1	1
2	1	2
2	2	1

Figure 5: The Natural Join of R with S (from Figure 4)

U (supplier part project)

supplier	part	project
1	1	2
2	1	1
2	2	1

Figure 6: Another Join of R with S (from Figure 4)

Inspection of these relations reveals an element (element 1) of the domain part (the domain on which the join is to be made) with the property that it possesses more than one relative under R and also under S. It is this element which gives rise to the plurality of joins. Such an element in the joining domain is called a point of ambiguity with respect to the joining of R with S.

If either $\Pi_{21}(R)$ or S is a function*, no point of ambiguity can occur in joining R with S. In such a case, the natural join of R with S is the only join of R with S. Note that the reiterated qualification "of R with S" is necessary, because S might be joinable with R (as well as R with S), and this join would be an entirely separate consideration. In Figure 4, none of the relations R, $\Pi_{21}(R)$, S, $\Pi_{21}(S)$ is a function.

Ambiguity in the joining of R with S can sometimes be resolved by means of other relations. Suppose we are given, or can derive from sources independent of R and S, a relation T on the domains project and supplier with the following properties:

(1) $\Pi_1(T) = \Pi_2(S)$

(2) $\Pi_2(T) = \Pi_1(R)$

(3) $T(j, s) \rightarrow \exists p(R(s, p) \wedge S(p, j))$

(4) $R(s, p) \rightarrow \exists j(S(p, j) \wedge T(j, s))$

(5) $S(p, j) \rightarrow \exists s(T(j, s) \wedge R(s, p))$,

then we may form a three-way join of R, S, T; that is, a ternary relation such that

$$\Pi_{12}(U) = R$$

$$\Pi_{23}(U) = S$$

$$\Pi_{31}(U) = T.$$

*A function is a many-one binary relation.

Such a join will be called a cyclic 3-join to distinguish it from a linear 3-join which would be a quaternary relation V such that

$$\Pi_{12}(V) = R$$

$$\Pi_{23}(V) = S$$

$$\Pi_{34}(V) = T.$$

While it is possible for more than one cyclic 3-join to exist (see Figures 7, 8 for an example), the circumstances under which this can occur entail much more severe constraints than those for a plurality of 2-joins. To be specific, the relations R, S, T must possess points of ambiguity with respect to joining R with S (say point x), S with T (say y), and T with R (say z), and, furthermore, y must be a relative of x under S, z a relative of y under T, and x a relative of z under R. Note that in Figure 7 the points

$$x = a; \quad y = d; \quad z = 2$$

have this property.

R:

s	p
1	a
2	a
2	b

S:

p	j
a	d
a	e
b	d
b	e

T:

j	s
d	1
d	2
e	2

Figure 7: Binary Relations with a Plurality of Cyclic 3-Joins

U:

s	p	j
1	a	d
2	a	e
2	b	d
2	b	e

U':

s	p	j
1	a	d
2	a	d
2	a	e
2	b	d
2	b	e

Figure 8: Two Cyclic 3-Joins of the Relations in Figure 7

The natural linear 3-join of three binary relations R, S, T is given by

$$R*S*T = \{(a, b, c, d) : R(a, b) \wedge S(b, c) \wedge T(c, d)\}$$

where parentheses are not needed on the left hand side because the natural 2-join (*) is associative. To obtain the cyclic counterpart, we introduce the operator γ which produces a relation of degree $n-1$ from a relation of degree n by tying its ends together. Thus, if R is an n-ary relation

$$\gamma(R) = \{(a_1, a_2, \ldots, a_{n-1}) : R(a_1, a_2, \ldots, a_{n-1}, a_n) \wedge a_1 = a_n\}.$$

We may now represent the natural cyclic 3-join of R, S, T by the expression

$$\gamma(R*S*T).$$

Extension of the notions of linear and cyclic 3-join and their natural counterparts to the joining of n binary relations (where $n \geq 3$) is obvious. A few words may be appropriate, however, regarding the joining of relations which are not necessarily binary. Consider the case of two relations R (degree r), S (degree s) which are to be joined on p of their domains ($p < r$, $p < s$). For simplicity, suppose these p domains are the last p of the r domains of R, and the first p of the s domains of S. If this were not so, we could always apply appropriate permutations to make it so. Now, take the cartesian product of the first r-p domains of R, and call this new domain A. Take the cartesian product of the last p domains of R, and call this B. Take the cartesian product of the last s-p domains of S and call this C.

We can treat R as if it were a binary relation on the domains A, B. Similarly, we can treat S as if it were a binary relation on the domains B, C. The notions of linear and cyclic 3-join are now directly applicable. A similar approach can be taken with the linear and cyclic n-joins of n relations of assorted degrees.

3.4 Composition

The reader is probably familiar with the notion of composition applied to functions. We shall discuss a generalization of that concept and apply it first to binary relations. Our definitions of composition and composability are based very directly on the definitions of join and joinability given above.

Suppose we are given two relations R, S. T is a composition of R with S if there exists a join U of R with S such that $T = \Pi_{13}(U)$. Thus, two relations are composable if and only if they are joinable. However, the existence of more than one join of R with S does not imply the existence of more than one composition of R with S.

Corresponding to the natural join of R with S is the natural composition of R with S defined by

$$R \cdot S = \Pi_{13}(R * S).$$

Taking the relations R, S from Figure 4, their natural composition is exhibited in Figure 9 and another composition is exhibited in Figure 10 (derived from the join exhibited in Figure 6).

R·S	(project	supplier)
	1	1
	1	2
	2	1
	2	2

Figure 9: The Natural Composition of R with S (from Figure 4)

T (project	supplier)
1	2
2	1

Figure 10: Another Composition of R with S (from Figure 4)

When two or more joins exist, the number of distinct compositions may be as few as one or as many as the number of distinct joins. Figure 11 shows an example of two relations which have several joins but only one composition. Note that the ambiguity of point c is lost in composing R with S, because of unambiguous associations made via the points a, b, d, e.

R (supplier	part)
1	a
1	b
1	c
2	c
2	d
2	e

S (part	project)
a	g
b	f
c	f
c	g
d	g
e	f

Figure 11: Many Joins, Only One Composition

Extension of composition to pairs of relations which are not necessarily binary (and which may be of different degrees) follows the same pattern as extension of pairwise joining to such relations. We now proceed to make use of these operations on relations in considering what relations need to be actually stored.

4. Expressible, Named and Stored Relations

Associated with a data bank are three collections of relations:

(1) the expressible set

(2) the named set

(3) the stored set

The expressible set is the collection of relations which can be designated by expressions in the retrieval language for the purpose of defining sets of data to be retrieved. Such expressions are constructed from simple names of relations, relational operators such as =, logical connectives and the quantifiers of the predicate calculus.

The named set is the collection of all relations in the data bank which the user can identify by means of simple public names. This set is a subset of the expressible set - usually a very small subset.

The stored set is the collection of all relations whose values are actually stored in the data bank. This set would normally be a subset of the named set, and we assume that it is. If the traffic on some unnamed but expressible relation grows to such proportions that such a relation should be included in the stored set, then it should be given a public name and thereby included in the named set.

Those relations which are in the named set and not in the stored set are defined by expressions (independent of time) involving names of relations in the stored set, together with the permutation-projection, natural composition, natural join and tie operators (Π, $\cdot$, $*$, γ). Such definitions by expressions must be within the scope of the retrieval language R.

Decisions regarding which relations belong in the named set are based mainly on the logical needs of the community of users, and particularly on the ever-increasing investment in programs using relations by name as a result of past membership of these relations in the named set. On the other hand, decisions regarding which relations belong in the stored set are based mainly on the transaction and interaction loads, the performance requirements of the users, and changes that take place in these factors.

5. Derivability, Redundancy and Consistency

A relation R is derivable from a set S of relations if there exists a sequence* of permutations, projections, natural joins, and ties which yields R from members of S. This sequence of operations yields a correct value for R at virtually any time (for the stored set, we must exclude times at which changes are actually being made to the values of R and S). Note that, because natural join is specified, there is no question as to which join to take.

A set of relations is strongly redundant if it contains at least one relation which is derivable from the rest of the members. While the named set of relations is likely to be redundant in this sense for user convenience, the stored set will often be non-strongly-redundant in order to save storage space as well as time to perform updates, insertions, and deletions. Only in an environment with a heavy load of queries relative to the other kinds of interaction with the data bank would strong redundancy be justified in the stored set of relations.

A set of relations is weakly redundant if it contains at least one relation which is not derivable from other members of the set, but is at all times a projection of some join of other members of the set. The join in question might be the natural one at some time and an unnatural one at some other time.

Generally speaking, weak redundancies are inherent in the logical needs of the community of users. They are not removable by the system or data base administrator. If they appear at all, they appear in both the named and stored sets. Strong redundancies, on the other hand, are removable from the stored set, providing the resulting performance changes are acceptable.

*We can omit natural composition in the list of operations, because it is a combination of a join and a projection.

As an example of a weak redundancy, consider the case cited previously in which there are binary relations R, S, T with meanings as follows:

- R(s, p) supplier s supplies part p to at least one project
- S(p, j) part p is supplied by at least one supplier to project j
- T(j, s) project j is supplied at least one kind of part by supplier s

All three relations are complex* relations with the possibility of points of ambiguity occurring from time to time in the potential joining of any two. Hence, none of them is derivable from the other two. However, constraints do exist between them, since each is a projection of some cyclic join of the three of them. Thus, this set of relations possesses a weak redundancy.

Whenever a set of relations is redundant in either sense, we shall associate with that set a collection of statements which define all of the redundancies which hold independent of time between the member relations. If the information system lacks - and it most probably will - detailed semantic information about each named relation, it cannot deduce the redundancies applicable to the named set. It might, over a period of time, make attempts to induce the redundancies, but such attempts would be fallible.

Given a collection C of relations and an associated set of constraint statements, we shall call C consistent or inconsistent according as it does or does not comply with the stated redundancies. For example, given stored relations R, S, T together with the constraint statement

"$\Pi_{12}(T)$ is a composition of $\Pi_{12}(R)$ with $\Pi_{12}(S)$",

we may check from time to time that the values stored for R, S, T satisfy this constraint. An algorithm for making this check would examine the first two columns of each of R, S, T (in whatever way they are represented in the system) and determine whether

(1) $\Pi_1(T) = \Pi_1(R)$

(2) $\Pi_2(T) = \Pi_2(S)$

(3) for every element pair (a, c) in the relation $\Pi_{12}(T)$ there is an element b such that (a, b) is in $\Pi_{12}(R)$ and (b, c) is in $\Pi_{12}(S)$.

There are practical problems (which we shall not discuss here) in taking an instantaneous snapshot of a collection of relations, some of which may be very large and highly variable.

*A binary relation is complex if neither it nor its converse is a function.

6. Data Bank Control

Inconsistencies in a collection of relations may arise due to inadequate or faulty input. An example of inadequate input is the insertion of a new element, say (2, 5) in the relation S (part, project) when part 2 has no supplier who supplies project 5 (see previous section). The generation of an inconsistency of this kind could be logged internally, so that if it were not remedied within some reasonable time interval by appropriate insertions in the relations R, T the system could notify the security officer. Alternatively, the system could assist the user in making insertions and deletions by informing him that such and such relations now need to be changed to restore consistency in the collection. Ideally, it should be possible to make different selections of system reaction to inconsistency for different subcollections of relations in an individual data bank.

ACKNOWLEDGMENT

The author wishes to thank Dr. F. P. Palermo and Dr. E. B. Altman of the San Jose Research Laboratory for helpful discussions.

REFERENCES

1. C. W. Bachman, "Software for Random Access Processing," Datamation, April 1965.
2. W. C. McGee, "Generalized File Processing," Annual Review in Automatic Programming 5, 13, pp. 77-149, Pergamon Press, 1969.
3. G. H. Mealy, "Another Look at Data," Proceedings Fall Joint Computer Conference, 1967.
4. A. Church, "An Introduction to Mathematical Logic I," Princeton, 1956.