C. Delobel
R. G. Casey

Decomposition of a Data Base and the Theory of Boolean Switching Functions

Abstract: The notion of a functional relation among the attributes of a data set can be fruitfully applied in the structuring of an information system. These relations are meaningful both to the user of the system in his semantic understanding of the data, and to the designer in implementing the system. An important equivalence between operations with functional relations and operations with analogous Boolean functions is demonstrated in this paper. The equivalence is computationally helpful in exploring the properties of a given set of functional relations, as well as in the task of partitioning a data set into subfiles for efficient implementation.

1. Introduction

The implementation of information systems has suffered from a severe dichotomy between the needs of the applications programmer, who wishes to concern himself only with the inherent properties of his data, and the viewpoint of the system designer, who sees the data in terms of the physical devices and processes that store and manipulate it. Inevitably, some of the designer's device-dependent notions are thrust on the system user. In effect the system designer says, "Here is what I do with your data; if you desire access to it you must . . . ," and proceeds to burden the user with a list of implementational details.

The entanglement of logical and physical aspects of data contributes more than inconvenience to large-scale information systems. It also imposes stiff economic penalties in additional training costs, in programming delays, and in deficiencies in program reliability and flexibility.

An important attempt to alleviate these difficulties is by the creation of an interface between the user and the system. The data base sublanguages, for example, provide such a facility. The interface allows the applications programmer to deal with a logical representation of data. As exemplified in [1], however, this approach still dilutes properties of the application with physical notions implied by the system.

An alternative schema has been proposed in order to offer a greater degree of independence between the system and the application. In prospect is an abstract model of data, one that the user can employ in order to characterize properties of his data, and which he can then pass on to the designer in order to assist the latter in selecting the organization of the data in a computer. The model seeks to have both parties communicate in the same framework, while at the same time permitting each to concentrate on those aspects of the data that affect him.

Several examples of such models currently exist in various stages of development. Childs [2] has described a data structure incorporating the notions and operations of set theory. Codd [3] and others [4] have proposed relational models of great flexibility, and one such model is currently being implemented at MIT under Project Mac. Delobel [5], Peccoud [6], and Boittieaux [7] have treated relational concepts in a precise mathematical fashion, placing particular emphasis on the role of "functional relations" in describing the properties of data. The implications of these functional relations for data base administration have been treated by several authors: by Codd [8] in defining normal forms for a formatted data collection; by Heath [9] in exhibiting the constraints that functional dependencies impose upon file operations; and by Rissanen and Delobel [10] in studying the decomposition of a file of data into subfiles.

In the present paper we show that much of the algebra of functional relations can be restated in a setting more familiar to computer specialists; namely, combinatorial Boolean algebra, also called "switching logic." (It is also possible to use the mathematical work of A. Bouchet [11] to prove the equivalence between relational algebra and Boolean algebra.) After performing this transformation, several aspects of the data model that are fundamental to both user and designer are found to be classical

Table 1 Tabular representation of data: (a) the original data set; (b) files $\mathscr{PY}$, $\mathscr{PH}$, $\mathscr{PT}$, $\mathscr{PYH}$, and $\mathscr{PYJ}$.

Attributes (a)

Items	*P*	*T*	*Y*	*N*	*H*
e(1):	Harry	Surgery	4	2B	6
e(2):	John	Pathology	2	1A	3
e(3):	John	Pathology	1	3C	5
e(4):	Evan	Pathology	2	3C	10
e(5):	John	Pathology	2	1A	7
e(6):	Evan	Pathology	3	2A	3
e(7):	Harry	Surgery	5	1A	5

Files (b)

$\mathscr{PY}$

Harry	4
John	2
John	1
Evan	2
Evan	3
Harry	5

$\mathscr{PH}$

Harry	6
John	3
John	5
Evan	10
John	7
Evan	3
Harry	5

$\mathscr{PT}$

Harry	Surgery
John	Pathology
Evan	Pathology

$\mathscr{PYH}$

P	*Y*	*H*
Harry	4	6
John	2	3
John	1	5
Evan	2	10
John	2	7
Evan	3	3
Harry	5	5

$\mathscr{PYT}$

P	*Y*	*T*
Harry	4	Surgery
John	2	Pathology
John	1	Pathology
Evan	2	Pathology
Evan	3	Pathology
Harry	5	Surgery

topics. This result suggests that switching theory is a powerful tool for exploring the abstract data model. Moreover, since the manipulation of Boolean functions is well-developed, it is apparent that the computations required in applications of the model can be carried out very efficiently by computer program. This feature is particularly important to the designer, who may wish to generate and evaluate a number of data organizations in order to optimize his system. It also offers an avenue for developing design procedures that optimize data structure by automatic means alone.

2. Preliminary notions

A user ordinarily perceives his data base as consisting of a collection of descriptions of various items. Each item is described by a set of attributes, i.e., designations of pertinent characteristics of the item, together with the corresponding attribute values for that item.

The data base may change with time: new items may be introduced or old ones deleted; values may be changed; the attributes themselves may be augmented or suppressed. In a more general analysis the time-varying behavior is important; here we shall be concerned with the data base only as it exists at a given instant, and shall disregard temporal factors.

Notationally we shall represent attributes by capital letters, e.g., $A, B, C, \ldots$, and attribute values by lower case letters, e.g., a for the value of the attribute A. Notation is simplified if we accept that a single letter may denote a compound attribute consisting of a number of elementary data characteristics (in the same way as, say, DATE consists of YEAR, MONTH, DAY). The associated attribute value is then a vector.

A convenient picture of the data base is as a table of attributes vs items, with values as the table entries. In Table 1, for example, we have five attributes with the following meaning:

Name of Professor $= P = \{$John, Harry, Evan$\}$
Year $= Y = \{1, 2, 3, 4, 5\}$
Room Number $= N = \{1A, 2A, 2B, 3C\}$
Time (hour) $= H = \{1, 2, 3, 4, 5, 6, \ldots, 24\}$
Teaching Course $= T = \{$Pathology, Physiology, Anatomy, Surgery$\}$

We shall have occasion to refer to such a display of the data, but we should also bear in mind that no restriction on the format of the data as actually stored in a memory device is entailed by so doing. The information may be stored hierarchically, or in accordance with list conventions, or as a collection of separate files, etc.

Another notational convention is our use of script capitals to denote subsets of the data base defined by projection over specified attributes (Table 1*b*). Thus $\mathscr{ABC}$ will represent the extraction from the data base of all distinct vectors of the form (a,b,c). In the notation of set theory we can write:

$$\mathscr{ABC} \triangleq \{(a,b,c) | \exists_j \ni a_j = a, b_j = b, c_j = c\}$$

Such a projection, representing all the data available concerning attributes A, B and C, we shall call a *file*. Furthermore, we shall consider any permutation of $\mathscr{ABC}$ to define the same file; for example, $\mathscr{BAC}$ is equivalent to $\mathscr{ABC}$. For our purposes data content is the essential thing, not the order in which values occur.

• *Join operation*

Let us give a label to that subset of a given data collection that is linked with a specified value of an attribute outside the collection. Given a value, a, belonging to file $\mathscr{A}$, we define

$$\mathscr{B}/a \triangleq \{b | (a,b) \in \mathscr{AB}\}.$$

If there is no b such that $(a,b) \in \mathscr{AB}$ then $\mathscr{B}/a \triangleq \{\phi\}$, where ϕ represents a null element. Thus $\mathscr{B}/a$ always has at least one element.

We can now define the *join* [3] (symbolized by an asterisk), an operation that merges a pair of files. The definition has two parts, depending on whether the files

Table 2 The joins $\mathscr{PY}*\mathscr{PJ}$ and $\mathscr{PY}*\mathscr{PH}$. Note that the first join is equal to the file $\mathscr{PYJ}$ (Table 1b), while the second contains (i.e., is larger than) the file $\mathscr{PYH}$.

$\mathscr{PY}*\mathscr{PT}$

P	Y	T
Harry	4	Surgery
John	2	Pathology
John	1	Pathology
Evan	2	Pathology
Evan	3	Pathology
Harry	5	Surgery

$\mathscr{PY}*\mathscr{PH}$

P	Y	H
Harry	4	6
Harry	4	5
Harry	5	6
Harry	5	5
John	2	3
John	2	5
John	2	7
John	1	3
John	1	5
John	1	7
Evan	2	10
Evan	2	3
Evan	3	10
Evan	3	3

do or do not contain common attributes. In the latter case there are distinct attributes A and B, and the join is defined as the Cartesian product:

$$\mathscr{A}*\mathscr{B} \triangleq \mathscr{A} \times \mathscr{B} = \{(a,b) | a \in \mathscr{A} \text{ and } b \in \mathscr{B}\}.$$

In the overlapping case let A be the attribute(s) common to the two collections. We then define

$$\mathscr{AB}*\mathscr{AC} = \bigcup_{a \in A} \{a\} \times (\mathscr{B}|a) \times (\mathscr{C}|a).$$

Thus the join combines all pairs of vectors, one from $\mathscr{AB}$ and one from $\mathscr{AC}$, that have a common a value. It is important to realize that this operation preserves the information contained in both $\mathscr{AB}$ and $\mathscr{AC}$. In fact, it may contain superfluous data, i.e., triples (a,b,c) that are not part of any item in the original data set. Table 2 illustrates this by means of the joins $\mathscr{PY}*\mathscr{PT}$ and $\mathscr{PY}*\mathscr{PH}$, using the files of Table 1*b*. In certain basic cases, as will be discussed, the join of $\mathscr{AB}$ and $\mathscr{AC}$ yields exactly the file $\mathscr{ABC}$. This is the case in Table 2 where $\mathscr{PY}*\mathscr{PT}$ is equal to $\mathscr{PYT}$.

The join can be shown to be commutative and associative. In addition it possesses an absorption property:

$$\mathscr{A}*\mathscr{AB} = \mathscr{AB}.$$

The join can occur in several different ways during an analysis of data. Otherwise unrelated attributes may be linked to a common attribute. For example, consider the two files (EMPLOYEE,SKILL) *and* (EMPLOYEE,CHILD). There is an association between SKILL = plumber and CHILD = Timmy if and only if there is an EMPLOYEE having both plumbing skill and a child named Timmy. Thus, it is logical to form the composite file (EMPLOYEE, CHILD,SKILL) by means of the join operator:

(EMPLOYEE,CHILD,SKILL) = (EMPLOYEE,SKILL) *(EMPLOYEE,CHILD).

The larger file would not ordinarily be stored in the tabular form that the above expression implies, but rather in a hierarchical structure as shown in a conventional way in Fig. 1. However, it is to the point to observe that the data relationships implied in such a structure can be represented algebraically using the join.

• *Functional relations*

Another way in which the join arises very naturally is in the case of a *functional relation* (FR). We shall say that there exists a functional relation from attribute A to attribute B, denoted $A \rightarrow B$, if, for each element (a,b) of $\mathscr{AB}$ there is no other element (a,b') in $\mathscr{AB}$ such that $b \neq b'$ (i.e., the set of ordered pairs (a,b) is a function). In application it is assumed that the property holds over a significant period of time when data are changing by addition, updating and deletion. An example is a personnel file in which an EMPLOYEE is assigned to exactly one DEPT. In this case

EMPLOYEE → DEPT.

In a revamping of the record-keeping system it might be decided to allow personnel to have several departmental assignments, thus destroying the functional relation. However, during the period in which the FR holds it can be useful in file structuring.

The definition given above can be extended to compound-attributes. If E and F are two distinct compound-attributes, where $E = \{E_1, E_2, \cdots, E_n\}$ and $F = \{F_1, F_2, \cdots, F_p\}$, we shall write either

$$E \rightarrow F$$

or else the equivalent notation

$$\{E_1, E_2, \cdots, E_n\} \rightarrow \{F_1, F_2, \cdots, F_p\}.$$

If for given E and F the above relation holds, and if in addition for every proper subset E' in E it is false that $E' \rightarrow F$, then we shall say that $E \rightarrow F$ is an *elementary functional relation* (or *EFR*).

The concept of a functional relation is similar in nature to that of a key attribute. If there exists a table of data $\mathscr{EF}$ and if $E \rightarrow F$ as above, then a given value of attribute E *uniquely* determines an entry in the table. An elementary functional relation defines an extremal attribute set having this property.

If two attributes are functionally related, then a file of which they are part may be resolved into two subcollections with no loss of information. Formally, if $B \rightarrow C$ then

$$\mathscr{ABC} = \mathscr{AB} * \mathscr{BC}.$$

Thus functionally dependent data (such as attribute C) can be projected out of a given file and the complete file can always be recovered by means of the join operation. This resolution is important to the system designer, for it allows him design freedom. He may or may not decide to break down a file into components, depending on possible storage savings, on the types of transactions made to the file, and other considerations. It is important for him to be aware of the logical possibilities that FR's offer. A discussion of this topic is presented in Section 4.

The set of elementary functional relations can be considered to be an inherent property of the information in a given data collection. The system designer normally has no control over this structure; the most that he can do is to deduce the relations. For example, in our schedule for medical students (Table 1), the following elementary functional relations might be defined:

ℓ_1: $P \rightarrow T$
ℓ_2: $P,H \rightarrow Y$
ℓ_3: $P,H \rightarrow N$
ℓ_4: $H,N \rightarrow P$
ℓ_5: $H,N \rightarrow Y$
ℓ_6: $H,Y \rightarrow P$
ℓ_7: $H,Y \rightarrow N$

Typically these relations would be supplied by someone familiar with the application. They express this person's semantic understanding of properties of the data. Thus, the first relation above means that a professor teaches only one course, while the third means that a professor can only be present in one room at a time. A second observer, asked to supply such a list, might (as we shall see) express the same semantic understanding with a different set of relations. Relations that exist in the data, but do not have semantic, i.e., enduring, relevance (for example, $P,Y \rightarrow N$ in Table 1) would not be declared at this logical level.

• *Properties of Functional Relations*

The following properties are easily proved:

1. Transitivity: if $E \rightarrow F$ and $F \rightarrow G$ then $E \rightarrow G$;
2. Reflexivity: $E \rightarrow E$;
3. Projectivity: if $E \subset F$ then $F \rightarrow E$;
   (in this case $F \rightarrow E$ cannot be an EFR);
4. Additivity: Let F,G represent a compound attribute, the union of components F and G. Then $E \rightarrow F$ and $E \rightarrow G \Rightarrow E \rightarrow F,G$;

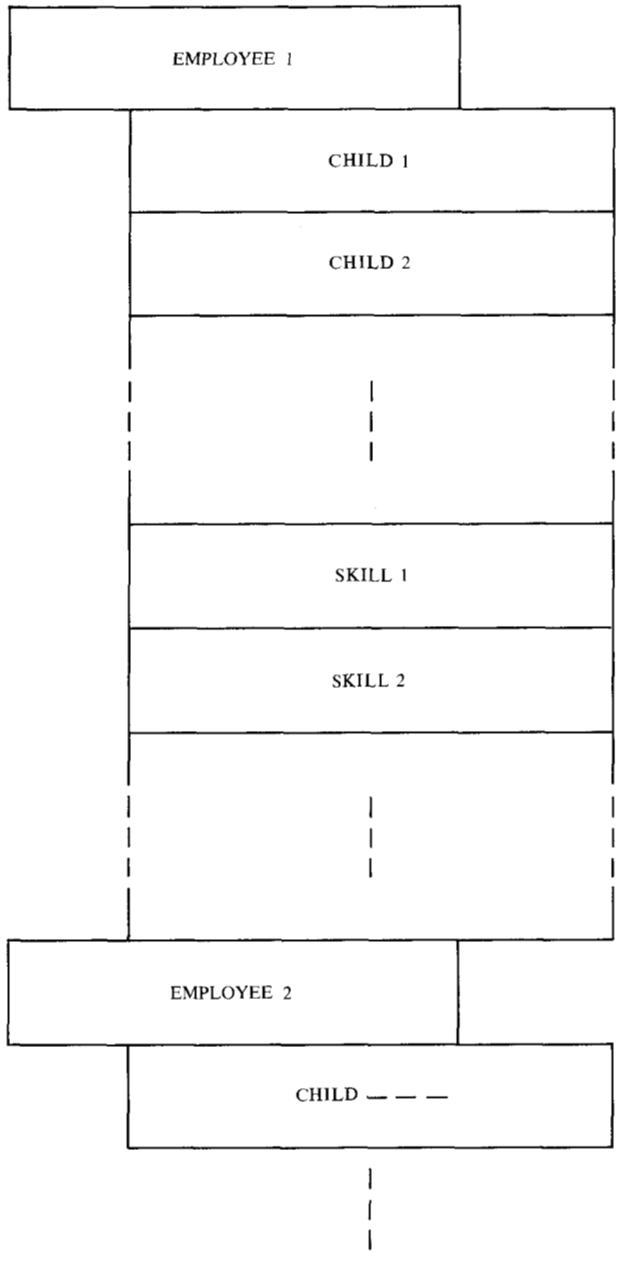

Figure 1 Hierarchical representation of data. Attributes that are not particularly associated with one another are listed in association with the attribute to which they relate.

5. Pseudotransitivity: if $E \rightarrow F$ and $F,G \rightarrow H$, then $E,G \rightarrow H$;
6. Augmentation: if $E \rightarrow G$ then $E,F \rightarrow G$, when F is any other attribute.

One application of the properties is to determine additional functional relations from a given list. In our example, we may use the transitivity rule together with relations ℓ_1 and ℓ_6 to derive

$H,Y \rightarrow T$,

which is not in the original set. By pseudotransitivity (which is actually a more general statement of the transitivity law) we can induce from ℓ_2 and ℓ_7 the relation

$$P,H \rightarrow N,$$

which is in the list as ℓ_3. Therefore, the given list is in a sense redundant.

Let us attempt to define standardized expressions of the information contained in a set of functional relations. In doing this it is sufficient to restrict ourselves to EFR's since these are the least redundant statements of FR's. There are two different descriptions of a relational structure that appear to be useful:

1. The set of all EFR's derivable from a given list. This maximal set is called the *closure*, and is unique.
2. A *nonredundant* set of EFR's from which all other EFR's (i.e., the closure) can be derived. By nonredundant we mean that if any relation is struck from this set the closure can no longer be obtained. This minimal set is called a *minimum cover* and is not unique in general.

To illustrate, the closure for our example consists of $\ell_1 - \ell_7$ together with the relations:

$$H,N \rightarrow T,$$
$$H,Y \rightarrow T.$$

We can derive two minimum covers for this problem; namely,

1. $\{P \rightarrow T, HN \rightarrow P, HY \rightarrow N, PH \rightarrow Y\}$,
2. $\{P \rightarrow T, HY \rightarrow P, HN \rightarrow Y, PH \rightarrow N\}$.

Here, for clarity, the comma between attributes to the left of the $\rightarrow$ has been deleted.

Either of the above sets of EFR's is sufficient to derive the closure by application of rules 1)–6) above.

An FR is a very special type of relation. Among the other classes of relation that occur in practice are:

1. Attributes whose values are calculated from many values of another attribute (or attributes), as YEARLY INCOME, e.g., is derived from MONTHLY INCOME,
2. The notion of direct relationship; in which, for example, CHILD is related to EMPLOYEE and DEGREE is related to EMPLOYEE, but there is no clear-cut relation between DEGREE and CHILD.

3. Boolean representation of functional relations

• *Boolean functions*

A *Boolean expression* consists of the binary operations (+), (•) and the unary operation (′) acting on a set of literals and the constants 0 and 1. The operations +, •, and ′ are called OR, AND, and COMPLEMENT, respectively. We refer the reader to standard references [12,13] for a detailed exposition of the properties of these expressions and the postulates that govern them.

A *Boolean function* is an expression in which the literals are variables that can each be assigned a value of either 0 or 1. For every assignment of values to variables, a value (again either 0 or 1) can be determined for the function itself, by applying the postulates.

The postulates can also be used to transform a given Boolean expression into numerous equivalent forms. For example the following functions are equivalent:

$$f = ab' + bc' + ca'$$
$$g = a'b + b'c + c'a$$
$$h = a'b + b'c + c'a + ab'\ bc' + ca'$$

• *Relational Forms*

We shall exhibit a useful correspondence between functional relations as defined in Section 2 and a class of Boolean functions. Suppose we are given the relation

$$X \rightarrow Y.$$

We shall associate this relation with the Boolean term xy'. The association is suggestive of the Boolean implication [12]. Using the latter, "x implies y" would be expressed as

$$x' + y.$$

The term above is the complement of the term we associate with a functional relation; that is,

$$xy' = (x' + y)'.$$

We could employ the standard Boolean implication to denote functional relations. In that case the principle of duality [12] shows that we would obtain dual forms of all the results given here regarding the connection between functional relations and their Boolean correspondences. The approach actually followed has the virtue of dealing with notions more familiar to the switching theorist, namely, disjunctive forms and prime implicants, rather than their duals.

In this correspondence if X and Y are compound, then the constituent variables of X and Y are connected by the (•) operation.

If we are given a set of functional relations, say

$$X_j \rightarrow Y_j \qquad j = 1, 2, \cdots, n,$$

where

$$X_j, Y_j \subset \{A, B, C, \cdots\},$$

we shall associate these with the Boolean function

$$f = \sum_{j=1}^{n} x_j y_j'.$$

If Y_j is compound, say

$$Y_j: \{Y_{j1}, Y_{j2}, \cdots, Y_{jm}\},$$

then we have

$$y_j' = (y_{j1} \bullet y_{j1} \bullet \cdots \bullet y_{jm})'$$
$$= y_{j1}' + j_{j2}' + \cdots + y_{jm}'$$

and

$$x_j \bullet y_j' = x_j \bullet y_{j1}' + x_j y_{j2}' + \cdots + x_j y_{jm}'.$$

Therefore the Boolean function associated with a set of functional relations can always be expressed as a sum of ANDed variables in which each AND term contains a single complemented literal. We shall call an expression having this property a *relational form.*

It is apparent that any boolean function in this form uniquely determines a set of functional relations. Given a relational form we simply take each term and place the primed variable (i.e., the attribute associated with it) on the right of the relational symbol ($\rightarrow$), and the unprimed variables on the left.

• *Equivalence of FR's and Relational Forms*

As we have seen, one way in which a Boolean function can be generated is by applying the laws of Boolean algebra to another function. Suppose that the initial function, say f, is formed from a set of functional relations, and suppose that g derived from it is also a relational form. Then the basic result to be demonstrated in this section is: *the relational interpretation of g is valid.* That is, any functional relations implied by g, but not present in f, can be derived from the relations used to form f.

Let us illustrate by an example before discussing the general case. The set of EFR's $\ell_1 - \ell_7$ in Section 2 defines the Boolean function

$$f = pt' + phy' + phn' + hnp' + hny' + hyp' + hyn'.$$

This function of five variables can be represented by a Karnaugh map [13] as in Fig. 2. An equivalent function (i.e., one having the same map) is the following:

$$g = pt' + hyt' + hytn' + pthn' + nhty' + nythp' + nht'.$$

Function g is in canonical form and is associated with the relations:

a. $P \rightarrow T$,
b. $HY \rightarrow T$,
c. $HYT \rightarrow N$,
d. $PTH \rightarrow N$,
e. $NHT \rightarrow Y$,
f. $NYTH \rightarrow P$,
g. $NH \rightarrow T$.

Each of the functional relations a) – g) is a consequence of relations $\ell 1 - \ell 7$. Thus, $\ell 1$ and a) are the same; b) is derivable by transitivity from $\ell 1$ and $\ell 6$; c) follows from $\ell 7$; d) is implied by $\ell 3$; e) is a consequence of $\ell 5$; f) is a weaker version of either $\ell 4$ or $\ell 6$; and g) is obtainable from $\ell 1$ and $\ell 4$.

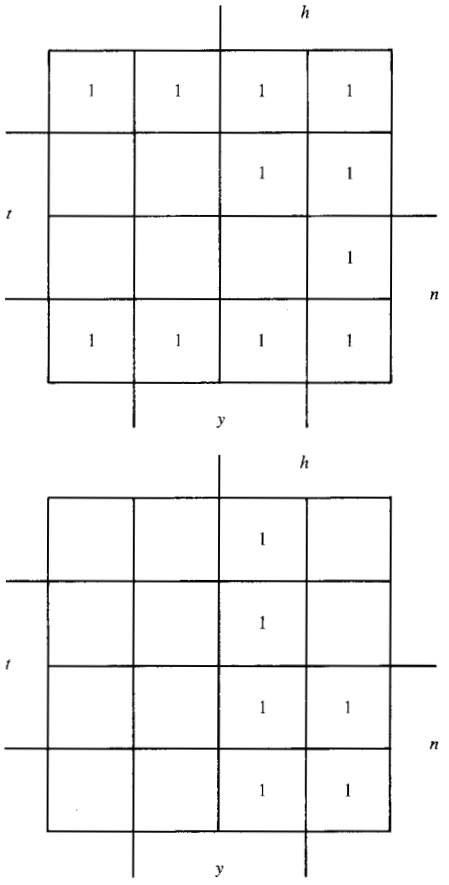

Figure 2 Karnaugh map for the example.

Conversely relations $\ell 1 - \ell 7$ can be deduced from a) – g). In Appendix A we prove the generalization: equivalence of canonical forms in the Boolean domain implies equivalence of the associated sets of functional relations, and conversely.

As a postscript to these remarks we offer the following corollaries, which follow directly from what is proved in Appendix A.

Corollary 1: The "closure" of a set of functional relations is the image of the set of prime implicants of the corresponding Boolean form.

Corollary 2: A "minimum covering" in the relational domain is the image of a minimum covering of the corresponding Boolean form (which is expressible as a form containing only prime implicants, see [13,14]).

We observe that the set of prime implicants and the two minimum covers obtainable from the map of Fig. 2 correspond exactly to the relational closure and the covers given in Section 2.

4. Decomposition of a collection

Any data collection can be represented as a table such as the example of Table 1. Such a table would be, in most cases, highly redundant and inefficient to maintain; however, it is a useful concept for the following reason. If a data base is to be resolved into subfiles, each describing a different subset of the attributes, then it is fundamental that for this resolution to be consistent, it must be possible to regenerate the global table from the collection of subfiles.

Thus, in a hierarchical decomposition, for example, in which the global description is broken down sequentially into smaller and smaller attribute sets, it is axiomatic that at each step of the process the data associated with the attributes being resolved should be regenerable from the several data collections defined.

The join operation, as we have seen in Section 2, is a basic procedure for merging two files. A necessary and sufficient condition for the join to reconstitute a given file is given by the following proposition.

Theorem: If and only if for every $a \in \mathscr{A}$ it is true that

$$\mathscr{BC}|a = (\mathscr{B}|a) \times (\mathscr{C}|a),$$

then

$$\mathscr{ABC} = \mathscr{AB} * \mathscr{AC}.$$

The proof follows directly from the definition of the join. It is clear that a functional relation $A \rightarrow B$ provides one way in which the conditions of the theorem are satisfied, for then $\mathscr{B}|a$ contains exactly one element. On the other hand, satisfaction of the theorem does not imply the existence of a functional relation.

In the following we consider the general situation where a set of functional relations is given, as well as (possibly) a number of additional join relationships not associated with FR's. The latter may arise as illustrated in the (EMPLOYEE,SKILL,CHILD) example discussed earlier. By a *decomposition* of a given data collection we shall mean a set of subfiles such that the original collection can be recreated by means of the join operation. For example the expression:

$$\mathscr{ABCDE} = \mathscr{AB} * \mathscr{BC} * \mathscr{ACD} * \mathscr{DE}$$

implies that the collection $\mathscr{ABCDE}$ can be stored as the set of files $\mathscr{AB}$, $\mathscr{BC}$, $\mathscr{ACD}$, and $\mathscr{DE}$. Our objective is to be able to generate the entire family of file decompositions that are consistent with the given relations.

• *Process for decomposing files*

Assume that $\Lambda = \{A_1, A_2, \cdots, A_n\}$ is the set of all attributes, and let $\{E_i \rightarrow A_i\}$ where $E_i \subset (\Lambda \sim A_i)$ be a given set of FR's. The functional relation $E_i \rightarrow A_i$ permits the decomposition

$$\mathscr{A}_1\mathscr{A}_2, \cdots, \mathscr{A}_n = \mathscr{E}_i\mathscr{A}_i\, \mathscr{A}_1\mathscr{A}_2, \cdots, \mathscr{A}_{i-1}\mathscr{A}_{i+1}, \cdots, \mathscr{A}_n.$$

We can now seek to break down recursively the attribute sets (E_i, A_i) and $(A_1, A_2, \cdots, A_{i-1}, A_{i+1}, \cdots, A_n)$ using the remaining EFR's.

The performance of such a decomposition is greatly facilitated by the correspondence between this process and the properties of a class of Boolean functions. The analogy is similar to that explored in Section 3 for analyzing functional relations. Here we construct the Boolean function

$$f(a_1, a_2, \cdots, a_n) = \prod_{i=1}^{n} a_i + \sum_j e_j a_j'$$

in which the terms $e_j a_j'$ correspond to the given FR's as before.

A fundamental property of the function defined above (see Appendix B) is that a prime implicant of f having no complemented variables defines a *key*, or minimal unique identifier, for the overall collection of data. In addition, as shown previously, any functionally dependent data can be projected out of the main collection and into separate files. These properties open the way for a decomposition of the collection in the Boolean domain.

Let us say that a Boolean function g is *included* in another function f if f takes on the value 1 whenever g does. In addition we introduce the notion of a *chain*. A product of Boolean variables

$$b = \prod_{i=1}^{k} b_i$$

will be said to be *chained* to a term x in function f if for each b_i in b either b_i appears in x or else there exists a product of variables c_i such that $c_i b_i'$ is a term in f and c_i is chained to x. In the function $f = r + rs' + st' + tu'$, for example, each of r, s, t, u is chained to r.

We can now state conditions under which a Boolean expression of the form

$$g = \sum_{i=1}^{p} g_i$$

determines a valid decomposition of the data collection associated with $f(a_1, a_2, \cdots, a_n)$. In order for the collection to be regenerable from its files, i.e., in order for

$$\mathscr{A}_1\mathscr{A}_2 \cdots \mathscr{A}_n = \mathscr{G}_1 * \mathscr{G}_2 * \cdots * \mathscr{G}_p$$

to be true, then the following four conditions must hold:

1. Every g_i is included in f.
2. At least one g_i is included in a key term.
3. Every attribute appears, either primed or unprimed, in some g_i.
4. Every variable in g is chained to the key term.

In a typical case there will be a number of decompositions satisfying 1–4. For example, consider data associated with attributes A,B,C,D and possessing the functional relations

$$AB \to C,$$
$$C \to D.$$

Then we have

$$f(a, b, c, d) = abcd + abc' + cd'$$

with unique key term ab. We list below four distinct decompositions of this collection:

$$g = \begin{cases} ab + abc' + abd' \\ abc' + cd' \\ abc' + abd' \\ abc' + bcd'. \end{cases}$$

A designer might wish to investigate any or all of the valid decompositions. However, a decomposition of special note is obtained from f by the following procedure:

1. Find the key terms of f in the manner indicated above.
2. Determine a minimum cover $g_1, g_2, \cdots, g_p$ of the relational form

$$\sum_j e_j x_j'.$$

3. If at least one g_i is included in a key term, then the decomposition is

$$g = \sum_{i=1}^{p} g_i.$$

Otherwise the decomposition is

$$g = (\text{key term}) + \sum_{i=1}^{p} g_i.$$

It can be shown that the function g so found satisfies the conditions for a valid decomposition. In addition we show in Appendix C that the files of g have a special property: if A,B,C are distinct attribute sets satisfying $A \to B \to C$, and $C \not\to B \not\to A$, then A and C cannot coexist in the same file.

Thus, no attribute can be transitively dependent on other attributes in the same file. Every file of g is therefore in the Third Normal Form of Codd [8], and the decomposition avoids certain undesirable update problems pointed out by Codd.

To illustrate, consider the attributes and relations shown in Table 3. The prime implicants of the corresponding Boolean function are also shown. The Boolean cover yields a simultaneous decomposition of the data; however, it is also interesting to view the process sequentially, as in the tree-schema shown in Fig. 3.

The residue (P,D) at the bottom of the tree corresponds to the uncomplemented Boolean term in the cover. This file, although it cannot be resolved further, is also probably not meaningful in the user's reference frame. Diplomas and pets are only related through their connections to some person's name. The physical inclusion of a table of diplomas versus pets in storage would be rather inefficient. Nevertheless, this subfile is perfectly consistent for a logical representation of the data in storage.

To obtain a more meaningful decomposition of the data base the user could have included extra semantic information in his description. For example, in addition to the FR's he could have provided a partial decomposition as follows:

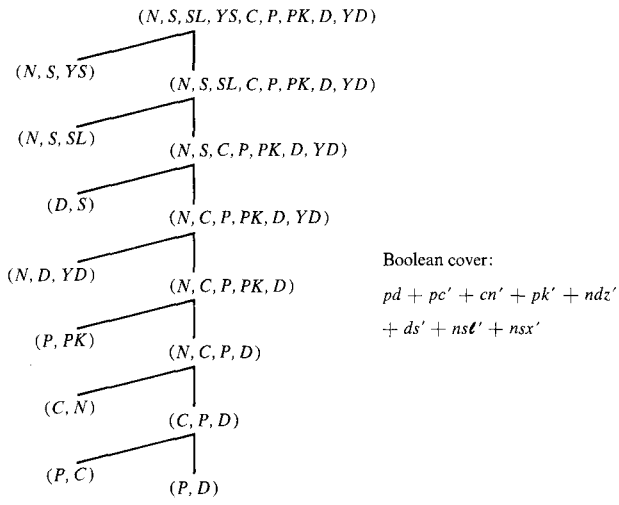

Figure 3 Tree schema for the example of Table 3.

Table 3 Example of the decomposition process. A hypothetical personnel file is shown, together with the functional relations pertinent to it.

Attributes	*Relational symbol*	*Boolean symbol*
Name	N	n
Skill	S	s
Skill level	SL	l
Experience (years)	YS	x
Child name	C	c
Pet name	P	p
Pet kind	PK	k
Degree	D	d
Year of Degree	YD	z

EFR's	P → C	D → S	P → PK
	N,D → YD	C → N	
	N,S → YS	N,S → SL	

Prime implicants

pd + pc' + ds' + pk' + nd z' + cn' + nsx' + nsl'
+ pn' + nd x' + ndl' + cdz' + scx' + scl' + pdz'
+ spx' + spl' + cdx' + cdl' + pdx' + pdl'

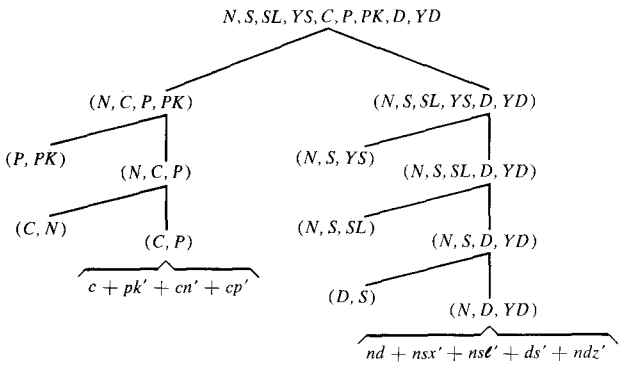

Figure 4 Alternative tree schema. The Boolean covers for each of the two main tree sections are shown beneath.

$$(N, S, SL, YS, C, P, PK, D, YD) = (N, C, P, PK) * (N, S, SL, YS, D, YD).$$

Here two attributes appearing on opposite sides of the join are essentially unrelated.

The Boolean cover and the decomposition tree corresponding to this description are shown in Fig. 4. Note that in this case the decomposition reflects the user's point of view. The left-hand portion of the tree defines a data set pertaining to the employee's children (note in the Boolean cover that CHILDREN is a key attribute for this subcollection). The right-hand portion, having NAME-DEGREE as a key, relates characteristics of the employee himself.

5. Conclusions

In this paper we have shown that Boolean algebra is a potent tool for analyzing the decomposition of a data base induced by certain classes of relations. During the decomposition process semantic concepts are transformed into algebraic form, providing a precise framework for communications between system designers and applications specialists.

The study reported here primarily concerns functional relations. However, the proofs given for the Boolean analogy depend only on several properties of this class of relation, namely the laws of reflexivity, augmentation and pseudotransitivity. It may be important to realize that any relation between attributes that satisfies these three laws can be transformed to the Boolean domain in an identical manner.

6. Acknowledgment

The authors are indebted to Dr. Frank Palermo of IBM Research for pointing out the connection between the analogy explored here and the standard Boolean implication, as well as for several other useful comments.

Appendix A: Equivalence of operations on functional relations and operations on boolean functions

Let a set of functional relations be given on attributes $A_1, A_2, \cdots, A_n$.

$$F = \{B_j \rightarrow C_j | j = 1, \cdots, \ell\}.$$

It can be assumed without loss of generality that the B_j are compounds of the attributes $A_1, A_2, \cdots, A_n$ while the C_j are simple. Suppose that from the relations in F a new functional relation, $X \rightarrow Y$, can be derived by applying the two basic rules: pseudotransitivity and augmentation. Then we have

$$G = \{B_j \rightarrow C_j, X \rightarrow Y\} \sim F,$$

where the equivalence symbol "$\sim$" is to be read in the sense of a two-way implication. That is, G can be derived from F and conversely.

Corresponding to F and G, respectively, are the two Boolean functions, f and g, expressed in our canonical form (only one primed literal in each product term):

$$f = \sum_j b_j c_j' \qquad g = \sum_j b_j c_j' + xy'.$$

We wish to show that for any f, g and F, G that correspond in this way, then

$$F \sim G \text{ iff } f \sim g.$$

The argument is clearly extendable by induction to F and G, or f and g, differing by more than one term.

Thus, if the above is shown to be true, then more generally two sets of functional relations being equivalent, the corresponding canonical Boolean functions are equivalent, and conversely.

1. Proof that $F \sim G \Rightarrow f \sim g$: The additional term $X \rightarrow Y$ is derived from F by applying pseudotransitivity and augmentation. We need only show that the Boolean counterparts of these operations produce corresponding terms in order to prove 1).

Pseudotransitivity: $\{A \rightarrow B, BD \rightarrow C\} \sim \{A \rightarrow B, BD \rightarrow C, AD \rightarrow C\}$.

The Boolean image of the left-hand side is $ab' + bdc'$, and by the law of absorption

$$ab + bdc' \sim (ab' + ab'dc') + (abdc' + bdc') \sim ab' + bdc' + adc'.$$

The new term produced, which does not change the value of the function, is adc', which corresponds directly with the relation derived by means of pseudotransitivity.

Augmentation: $\{A \rightarrow B\} \sim \{A \rightarrow B, AC \rightarrow B\}$.

In the Boolean domain we may write, correspondingly,

$$ab' \sim ab' + ab'c,$$

which is true by the law of absorption. Thus, augmentation is the relational counterpart of the law of absorption.

Having proved these two correspondences, we have demonstrated 1).

2. *Proof that* $f \sim g \Rightarrow F \sim G$: The proof in this direction is more difficult. Equivalent Boolean forms can be generated using other operations than the counterparts of augmentation and pseudotransitivity. Yet we shall demonstrate that all equivalent *canonical* forms can be produced using just these operations. It then immediately follows that G is derivable from F using the corresponding operations to those that form g from f, for we have shown in 1) that these operations mirror one another exactly.

The proof makes use of the following lemma:

Lemma: All the prime implicants of f can be generated using only the Boolean counterparts of pseudotransitivity and augmentation.

A prime implicant (see [10]) is an irreducible term derivable from f by Boolean operations. It follows that each term of g is either a prime implicant or obtainable from some prime implicant of f by means of the law of absorption. Hence, if the lemma is true, then g can be derived from f using only counterparts of relational operations. This is sufficient to prove 2).

Proof of Lemma: Our proof rests on properties of the ★-algorithm for generating the prime implicants of a given disjunctive Boolean form [14, pp. 156–163]. The algorithm constructs the set of all prime implicants by successive application of two operations. One of these is the law of absorption; the other is called the ★-operation. In order to prove the lemma we need only to show that the ★-operation on canonical forms is expressible in terms of absorption and Boolean pseudotransitivity.

The ★-operation defines a product term from two given terms. This new term does not alter a function containing the given terms; that is, if u and v are product terms:

$$u + v \sim u + v + u \bigstar v.$$

We assume that u and v are defined on variables a_1, a_2, $\cdots$, a_n, and for convenience we represent product terms as an n-vector, for example,

$$u = [u_1, u_2, \cdots, u_n],$$

where each $u_j = a_j$, a_j' or x.

The symbol x denotes that the corresponding variable is absent from the term. Given two such vectors, u and v, the operation $u \bigstar v = w$ is defined as follows:

a. w is undefined (i.e., the ★-operation produces no new term) if there exist distinct indices i and j such that $(u_k, v_k) \in \{(a_k, a_k'), (a_k', a_k)\}$ for both $k = i$ and $k = j$.

b. If the above is not true, then

$$w_j = a_j \text{ if } (u_j,v_j) \in \{(a_j,a_j),(a_j,x),(x,a_j)\}$$
$$= a_j' \text{ if } (u_j,v_j) \in \{(a_j',a_j'),(a_j',x),(x,a_j')\}$$
$$= x \text{ if } (u_j,v_j) \in \{(a_j,a_j'),(a_j',a_j),(x,x)\}.$$

Thus w is determined by nine different possible combinations of values of elements of u and v. Let us lump variables together to obtain representative terms having length $n = 9$, with each possible combination of (u_j,v_j) exhibited once, keeping in mind that any variable indicated in a combination may be composite, or may be absent. Then we have

$$u = a_1a_2\ x\ a_4'a_5'\ x\ a_7a_8'\ x,$$
$$v = a_1\ x\ a_3a_4'\ x\ a_6'a_7'a_8\ x.$$

If u and v represent terms in a canonical expression then there is exactly one primed literal in each. That is, u can have only one of the elements a_4', a_5', a_8'; the others must be absent. The primed variables that are present must be simple. Likewise, v can have only one of the (simple) literals a_4', a_6', a_7'. Thus, the possible forms for u and v are:

$$u(1) = a_1a_2a_4'a_7 \quad u(2) = a_1a_2a_5'a_7 \quad u(3) = a_1a_2a_7a_8'$$
$$v(1) = a_1a_3a_4'a_8 \quad v(2) = a_1a_3a_6'a_8 \quad v(3) = a_1a_3a_7'a_8$$

Now consider the nine possible expressions for w:

1. $u(1) \bigstar v(2) = a_1a_2a_3a_4'a_7a_8 \subset u(1)$,
2. $u(1) \bigstar v(2) = a_1a_2a_3a_4'a_6'a_7a_8 \subset u(1)$,
3. $u(1) \bigstar v(3) = a_1a_2a_3a_4'a_8$,
4. $u(2) \bigstar v(1) = a_1a_2a_3a_4'a_5'a_7a_8 \subset u(2)$,
5. $u(2) \bigstar v(2) = a_1a_2a_3a_5'a_6'a_7a_8 \subset u(2)$,
6. $u(2) \bigstar v(3) = a_1a_2a_3a_5'a_8$,
7. $u(3) \bigstar v(1) = a_1a_2a_3a_4'a_7$,
8. $u(3) \bigstar v(2) = a_1a_2a_3a_6'a_7$,
9. $u(3) \bigstar v(3) =$ undefined (no term generated).

As indicated, the terms produced in cases 1), 2), 4), and 5) can be absorbed back into the generating terms, and this is done whenever possible in the prime implicant algorithm. Thus, we do not have to show correspondences for these cases, nor for case 9), where no new term results. Only cases 3), 6), 7), and 8) need be considered further.

However, these four ★-operations can also be duplicated using Boolean pseudotransitivity. In case 3), for example, we can regroup terms and write

$$v(3) + u(1) = (a_1a_3a_5)a_7' + a_7(a_1a_2)a_4'.$$

In this form we may apply pseudotransitivity to obtain the new term

$$(a_1a_3a_8)(a_1a_2)a_4' = a_1a_2a_3a_4'a_8,$$

which is exactly $u(1) \bigstar v(3)$.

The same equivalence holds for cases 6), 7) and 8). In each instance the new term $u \bigstar v$ is exactly that which is produced by applying the pseudotransitivity rule. Hence, the ★-algorithm for determining prime implicants yields terms that could equally well have been found using relational-type operations. It follows that every prime implicant of a canonical form has a relational counterpart (i.e., every prime implicant has a single primed literal), and that functional relations can be derived from the functional relations corresponding to the given Boolean form.

Appendix B: Determination of keys by Boolean operations

Definition: Assume a data collection defined on attributes $A_1, A_2, \cdots, A_n$. Let J be a subset of the integers $1, 2, \cdots, n$. A subset of the given attributes, $\alpha(J) = \{A_j; j \varepsilon J\}$, is called a *key* for the given data if it is true that

$$\alpha(J) \rightarrow A_r \text{ for } r = 1, 2, \cdots, n$$

and if, furthermore, there exists no set

$$J' \subset J, J' \neq J,$$

such that

$$\alpha(J') \rightarrow A_r \text{ for } r = 1, 2, \cdots, n.$$

This definition agrees with the ordinary notion of a key as an attribute (compound in this case) whose values are in one-to-one correspondence with the items in the data collection.

Theorem: Let $f(a_1, a_2, \cdots, a_n)$ be a relational form, whose prime implicants constitute the set $\{f_i | i = 1, 2, \cdots, m\}$. If a term $p = a_{k+1}a_{k+2}\cdots a_n$, with each variable uncomplemented, is a prime implicant of the Boolean function:

$$g = \prod^{n} a_j + f,$$

then the attributes corresponding to p constitute a key. That is, we have

$$A_{k+1}A_{k+2}\cdots A_n \rightarrow A_r \qquad r = 1, 2, \cdots, n,$$

and there is no subset of $A_{k+1}\cdots A_n$ for which this is true.

Proof: We shall show that the theorem is a consequence of properties of the ★-algorithm. (See Appendix A for the definition of the ★-operation.) The ★-algorithm in this case begins with the set

$$C_0 = \{(\text{terms of } f), \prod^{n} a_j\}$$

and iteratively forms the sets

$$C_{j+1} = C_j \cup C_j \bigstar C_j \qquad j = 0, 1, 2, \cdots$$

All possible absorptions are performed within C_{j+1}, and all elements having more than $(n-j)$ literals present are eliminated.

Thus, prime implicant p must appear by the kth step of this process. Also, p cannot be derived from the terms of f alone, since all the prime implicants of a relational form have one primed literal. Therefore, $\prod^{n} a_j$ must be in the process leading to p.

Now the ★-operation has the property that if terms v_1 and v_2 have m_1 and m_2 literals absent, respectively, then v_1, v_2 has at most $\min(m_{1+1}, m_{2+1})$ literals absent. Therefore, at each step of a sequence of operations on $\prod^{n} a_j$, at most one additional literal can be eliminated. Therefore, in order to arrive in k steps at a term having k literals absent there must be a sequence of the form:

$$(\cdots((\prod^{n} a_j \bigstar w_1) \bigstar w_2)\cdots \bigstar w_k) = p$$

where each w_j may be the product of other ★-operations. We may write this sequence as

$$u_j = u_{j-1} \bigstar w_j \qquad j = 1, 2, \cdots, k,$$

where

$$u_0 = \prod^{n} a_j \text{ and } u_k = p.$$

We shall show that the sequence of u_j and w_j must have the following form (possibly after renumbering of literals $a_1, a_2, \cdots, a_k$):

u_0	a_1	a_2	a_3	$\cdots$	a_{k-1}	a_k	a_{k+1}	$\cdots a_n$
w_1	a_1'	y	y	$\cdots$	y	y	y	$\cdots y$
u_1	x	a_2	a_3	$\cdots$	a_{k-1}	a_k	a_{k+1}	$\cdots a_n$
w_2	x	a_2'	y	$\cdots$	y	y	y	$\cdots y$
u_2	x	x	a_3	$\cdots$	a_{k-1}	a_k	a_{k+1}	$\cdots a_n$
w_3	x	x	a_3'	$\cdots$	y	y	y	$\cdots y$
w_{k-1}	x	x	x	$\cdots$	a_{k-1}'	y	y	$\cdots y$
u_{k-1}	x	x	x	$\cdots$	x	a_k	a_{k+1}	$\cdots a_n$
w_k	x	x	x	$\cdots$	x	a_k'	y	$\cdots y$
u_k	x	x	x	$\cdots$	x	x	a_{k+1}	$\cdots a_n$

In the above, x denotes "variable absent," and y in the jth position denotes either a_j or x.

This sequence illustrates the following two basic properties of the w_j:

1. A variable appearing in complemented form earlier in the sequence must be absent from all later terms in the sequence.
2. Only a single complemented variable can be present in any w_j; otherwise the ★-operation yields a null term.

If some w_j fails to satisfy both conditions then no variable is eliminated at that step, and so the sequence cannot generate a prime implicant under the ★-algorithm.

There are two possible forms to consider for the terms w_i:

1. w_i "contains" $\prod^n a_j$ (by which we mean $w_i + \prod a_j = w_i$), in which case w_i cannot have any complemented literals (otherwise it could not absorb $\prod^n a_j$). But, in this case, $w_j \bigstar u_{j-1}$ fails to eliminate a variable, since neither w_j nor u_j contain any complemented literals. Hence this case never occurs in a sequence leading to p.

2. w_i does not contain $\prod^n a_j$, in which case w_j is contained in some prime implicant of f. But this means that w_j has a valid relational interpretation since it has only a single complemented literal. In correspondence with the sequence of w_i we may write

$$w_i\colon Y_{i+1}^{(i)} Y_{i+2}^{(i)} \cdots Y_m^{(i)} \to A_i \qquad i = 1, 2, \cdots, k,$$

where $Y_j^{(i)}$ = either A_j or else A_j is absent.

But from such a sequence of relations we can easily deduce that $A_{k+1}, \cdots, A_n$ is a key. For by augmentation $W_i \Rightarrow V_i$, where

$$V_i\colon A_{i+1}A_{i+2} \cdots A_n \to A_i \qquad i = 1, 2, \cdots, k.$$

If we define U_i to be the relation

$$U_i\colon A_{i+1}A_{i+2} \cdots A_n \to A_1A_2 \cdots A_i,$$

then we can show inductively that $(V_1, V_2, \cdots, V_k)$ imply that U_k is true, for (V_1, V_2) imply U_2 by pseudotransitivity. If $(V_1, V_2, \cdots, V_j)$ imply U_j for some j, then $(V_1, V_2, \cdots, V_j, V_{j+1})$ imply (U_j, V_{j+1}) and by pseudotransitivity (U_j, V_{j+1}) yields U_{j+1}, completing the proof by induction.

Therefore, if p is a prime implicant of g then U_k is true. It remains to show that a stronger relation cannot be true, i.e., it is not true that

$$A_{k+2}A_{k+3} \cdots A_n \to A_1A_2 \cdots A_kA_{k+1}$$

If this were true then f would contain all the terms

$$t_j = a_j'a_{k+2}a_{k+3} \cdots a_n \text{ for } j = 1, 2, \cdots, k+1$$

and we would have

$$((\cdots(\prod^n a_j \bigstar t_1) \bigstar t_2) \cdots \bigstar t_{k+1}) = a_{k+2}a_{k+3} \cdots a_n,$$

contradicting the hypothesis that p is a prime implicant of g.

Appendix C: Third normal form property of a minimum cover

Let $g = \sum_{i=1}^{p} a_ic_i'$

be a minimum cover of a given relational form. Let the functional relations corresponding to g be denoted by

$$A_i \to C_i \quad i = 1, 2, \cdots, p,$$

where A_i may be an attribute set, but C_i is simple. Then the following proposition is *false:*

There exist index $k \in \{1,p\}$ and attribute set $X = X_1, X_2, \cdots, X_q$ such that the following conditions hold:

1. $X_j \not\subset A_k$ for each $j \in \{1,g\}$,
2. $C_k \neq X_j$ for each $j \in \{1,g\}$,
3. $C_k \not\to X_j$ for each $j \in \{1,g\}$,
4. $X \not\to A$ for any $A \subset A_k$,
5. $A_k \to X \to C_k$.

Proof: Assume the contrary, i.e., there are such an X and a k. It is not restrictive to consider $k = 1$. Let us also define

$$\bar{g} = \sum_{2}^{p} g_i.$$

We shall show that the existence of X and k imply that $\bar{g} = g$ and therefore g is not a minimum cover since the term g_1 can be deleted.

In order to prove the equality of $\bar{g}$ and g it is sufficient to show

a. $a_1x' \subset \bar{g}$,
b. $xc_1' \subset \bar{g}$.

If a) and b) are true then the easily verified identity

$$a_1x' + xc_1' = a_1x' + xc_1' + a_1c_1'$$

proves that $\bar{g} = \bar{g} + a_1c_1' = g$.

It remains only to prove statements a) and b).

Proof that $a_1x' \subset \bar{g}$: Since $a_1x' \subset g$ we must have

$$(a_1x') \cdot g_1' \subset \bar{g},$$

i.e., $(a_1x') \cdot (a_1' + c_1) = a_1c_1x' \subset \bar{g}$,

but $a_1c_1x' = \sum_t a_1c_1x_t'$.

$\therefore$ for each t, $a_1c_1x_t' \subset \bar{g}$.

Then for each t, there exists an index $s = s(t)$, where $s \neq 1$, such that $a_1c_1x_t' \subset g_s$ (i.e., $a_1c_1x_t'$ is included in a term of $\bar{g}$).

The term g_s must have one of the following three forms:

1. c_1x_t'
2. zc_1x_t'
3. zx_t'

} where $a_1 \subset z$, $c_1 \not\subset z$.

But form 1) is impossible since $c_1 \not\to x_t$ and if g_s has form 2) then

$$g_s + g_1 = a_1c_1x_t' + a_1c_1' = a_1x_t' + a_1c_1',$$

so that g_s is not a prime implicant. But this cannot be, since g is a minimum cover and every term must be a prime implicant. Hence g_s has the third form, and so

$a_1 x_t' \subset \bar{g}$ for each t, proving $a_1 x' \subset \bar{g}$.

Proof that $xc_1' \subset \bar{g}$: Since $xc' \subset g$ we must have $(xc_1') \cdot g_1' \subset \bar{g}$, i.e.,

$$(xc_1') \cdot (a_1' + c_1) = xa_1'c_1' \subset \bar{g}.$$

Since A_1 is compound, we can write:

$$a_1 = \prod_r a_{1r},$$

where each a_{1r} is a distinct variable.

Now

$$xa_1'c_1' = \sum_r xa_{1r}'c_1'$$

and so

$$xa_{1r}'c_1' \subset \bar{g}$$

for each r.

In correspondence with each r there is $u = u(r)$, $u \neq 1$, such that

$$xa_{1r}'c_1' \subset g_u.$$

The term g_u can have one of two forms, namely,

1. za_{1r}'
2. zc_1'

} where $x \subset z$.

But 1) is impossible since then $Z \subset X$ and $Z \rightarrow A_{1r}$, and so $X \rightarrow A_{1r}$, which is ruled out by hypothesis. Therefore, g_u has form 2), and we have $xc_1' \subset zc_1' \subset \bar{g}$.

References

1. CODASYL Systems Committee, *Feature Analysis of Generalized Data Base Management Systems*, ACM, New York, May 1971.
2. D. L. Childs, "Description of a Set-Theoretic Data Structure," *AFIPS Conference Proceedings*, Fall Joint Computer Conference, 1968.
3. E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* **13**, 6, 377–387 (June 1970).
4. A. J. Strnad, "The Relational Approach to the Management of Data Bases," *MAC Technical Memorandum 23, Project MAC*, Cambridge, Mass., April 1971.
5. C. Delobel, "Aspects theoriques sur la structure de l'information dans une base de donnees," *Revue Informatique et de Recherche Operationnelle*, No. B. 3., pp. 37–64 (1971).
6. F. Peccoud, "MODSIN" DGRST (National Office for Scientific and Technical Research) 103, rue de l'Université, 75 Paris 7 éme, France, Contract No. 65, FR 201, July 1967.
7. J. Bottiaux, "Etude Mathematique des relations d'un Ensemble de Notions," *Contract DGRST (National Office for Scientific and Technical Research) No. 67-01-015*, 103, rue de l'Université, 75 Paris 7 éme, France, July 1969.
8. E. F. Codd, "Further Normalization of the Data Base Relational Model," in *Data Base Systems*, Courant Computer Science Symposia Series, Volume 16, Prentice-Hall, Englewood Cliffs, N.J., 1972.
9. I. J. Heath, "Unacceptable File Operations in a Relational Data Base," 1971 ACM SIGFIDET Workshop, Association for Computing Machinery, New York, November 1971, pp. 19–68.
10. J. Rissanen and C. Delobel, "Decomposition of Files: A Basis for Data Storage and Retrieval," *IBM Technical Report*, to be published.
11. A. Bouchet, "Etude Combinatoire des ordennes finis." Thesis 1971, University of Grenoble.
12. G. Birkhoff and S. MacLane, *A Survey of Modern Algebra*, Ch. XI, The Macmillan Co., New York, 1953.
13. M. P. Marcus, *Switching Circuits for Engineers*, Prentice-Hall, Englewood Cliffs, New Jersey, 1962.
14. R. E. Miller, *Switching Theory*, Vol. 1, John Wiley and Sons, New York, 1965.

Received January 19, 1973

Revised April 17, 1973

Dr. C. Delobel is located at the University of Grenoble, France; the work reported here was done during a period of resident study at the IBM Research Division Laboratory, San Jose. Dr. R. G. Casey is at the IBM Research Division Laboratory, Monterey and Cottle Roads, San Jose, California 95193