A Functional View of Data Independence

by

Michael Stonebraker
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California.

ABSTRACT

Many researchers have used the term "data independence" without indicating a precise meaning. One common definition is -- the isolation of a program from considerations of the data which it processes [1,2]. Another is -- the ability of an applications program to execute correctly regardless of the actual storage of its data [3,4]. Although these suggest the general concept, a precise framework is clearly needed. The current paper provides such a framework and explores its ramifications.

A Functional View of Data Independence

I. Introduction

It is evident that "data independence" involves the ability of an applications program to run correctly even after changes have been made in its data storage, as suggested by Date and Hopewell [3,4]. However, they do not precisely define the concept and deal only with relational data bases. Moreover, they suggest that data independence is an absolute term, i.e. that applications programs should execute correctly regardless of their data storage. We, on the other hand, claim that a range of changes to a data base is possible from the more trivial to the more complex. Consequently, there can be a corresponding range of data independence. Thus, the concern will be for classifying relative degrees of data independence instead of defining an absolute term.

In the next section, we define precisely the concept of "a change in data storage." This will be done in terms of functions on physical files. Also, we suggest two definitions of data independence for such functions.

Then we classify possible transformations to a data base into seven categories and demonstrate a hierarchical relationship among the classes. In addition, we present examples of transformations in each class which existing and proposed data base management systems can support data-independently. We argue that the degree of data independence possessed by a data base management system should be defined in terms of the sets of functions from the various categories that obey either of our functional definitions of data independence. As a result, the notion of data independence will not be synonymous with either definition mentioned in the abstract.

It is theoretically possible for a data base management system to insure that an applications program runs correctly after any change in any category. This hypothetical system, of course, would be hopelessly complex. As a result, we introduce a small set of primitive transformations to physical files. Implementation of compositions of these functions data-independently is shown to provide independence for most normally used storage structures.

Lastly, three software packages are examined for their ability to support transformations in the various classes without impacting applications programs. Moreover, their levels are compared to the one provided by the above primitives and found to be far inferior. It is suggested that systems move toward support of the suggested primitives.

II. Physical and Logical Files

A set of users (computations, job steps, proccesses) [5] $U_1, U_2, \ldots$ are assumed to interact with a data base. These users

do so by making calss on a data base management system, D. As noted in Figure 1, D can be assumed to interact directly with the data base. In order to define this data base precisely, we first define a physical file.

Let A be the set of valid addresses in secondary storage of a given computer system. Let C_A be the set of all binary bit strings of length K or less which are possible contents of these addresses. Here, K is a system determined maximum record length. Let $a_i \varepsilon A$ be an address in A and let $C_A(a_i) \varepsilon C_A$ be the contents of address a_i. Consequently, the two-tuple $(a_i, C_A(a_i))$ is an element of $A \times C_A$. Denote by P_n the product space

$$\underbrace{A \times C_A \times A \times C_A \times \ldots \times A \times C_A}_{n}.$$

An element $p \varepsilon P_n$ could be a 2n-tuple corresponding to n addresses and their contents, i.e. $p=(a_1, C_A(a_1), \ldots, a_n, C_A(a_n))$. For such an element, p, denote by A(p) the set of n addresses $\{a_1, \ldots, a_n\}$. Lastly, denote by

$$P = \bigcup_{i=1}^{w} P_i ,$$

the union of all P_i where i is less than or equal to the number of address, w, in A. Hence, the following definition is appropriate:

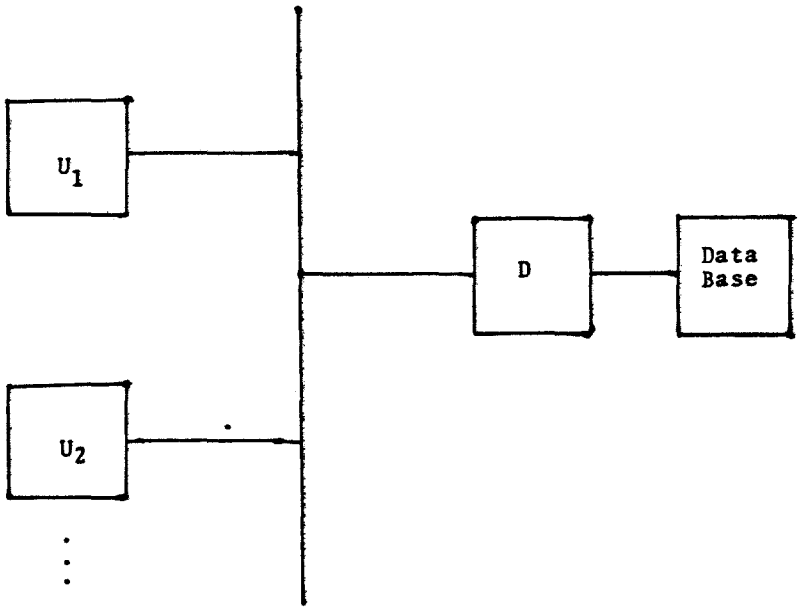
Definition 1: A <u>physical file, p</u>, is an element $p \varepsilon P$ such that $A(p)=\{a_1, \ldots, a_n\}$ for some n has the property that $a_i \neq a_j$ for $i \neq j$.

A physical file is an ordered set of addresses and their contents. This set must be smaller than the maximum number of secondary storage addresses and each address must be distinct.

Definition 2: A <u>data base, b</u>, is a set $p_1, \ldots, p_n$ of physical files $p_i \varepsilon P$ such that $A(p_1) \cap A(p_2) \cap \ldots \cap A(p_n) = \phi$.

Consequently, a data base is a set of distinct physical files. There is no requirement that

$$\bigcup_{i=1}^{n} A(p_i) = A .$$



System Structure

Figure 1

key 1	data 1
key 2	data 2
← m →	← t →
key n	data n

ISAM Logical File

Figure 2

Hence, extra secondary storage space may exist which is unused or used for files not in the given data base.

The program, D, however, usually does not let users see $p_1,\ldots,p_n$ or even a subset of these files. If it did, no software assistance would be available to users and they would physically interact with the data base. Rather, D usually presents users with a set of different spaces $\{h_1,\ldots,h_m\}$ in which to program interactions with the data base. These will be termed logical files. Let E be a set of addresses (often $\{1,2,3,\ldots,r\}$) and C_E be the set of valid contents of an address. Again, let

$$H_n = \underbrace{E\times C_E\times\ldots\times E\times C_E}_{n}, \quad H = \bigcup_{n=1}^{v} H_n ,$$

and for $h\in H$ let $E(h)$ be the set of addresses. Here, v is the system determined maximum number of addresses in a logical file. Hence, the following notions of a logical file and a logical data base are appropriate.[1]

Definition 3: A <u>logical file, h</u>, is an element $h\in H$ such $E(h)=\{e_1,\ldots,e_n\}$ for some n has the property that $e_i\neq e_j$ for $i\neq j$.

Definition 4: A <u>logical data base, ℓ</u>, is a set $h_1,\ldots,h_n$ of logical files $h_i\in H$.

Note that for logical files there is no requirement that addresses in the various files be distinct as was the case for physical files. Note also that specific systems may have additional constraints on a logical file. In [6], for example, logical addresses for a file h, $E(h)$, must be the set $\{1,2,\ldots,s\}$ for some positive integer $s \leq 2^{20}$. Lastly, note that interactions with logical files may be restricted. In fact, if users interact with D through a non-procedural language, then they may not even be able to retrieve the contents of a given address in h.

In order to further illuminate the concepts of physical and logical files, we offer an example. Here, we suggest the two

[1] Most data base management systems provide logical files with considerably more structure than suggested here. This definition could be expanded to include any given system without affecting the remainder of the paper.

views p and h for a single ISAM file [7].

The ISAM user perceives h as shown in Figure 2. Here, in each of n locations, the first m bits are a key [7] and the remaining t are other data. A logical file is arranged in collating sequence such that key 1 $\leq$ key 2 $\leq \ldots \leq$ key n. The user has a set of commands on this space including reading or writing the contents of the address corresponding to a given key and reading or writing the address after the current one.

Of course, ISAM maps this logical file into a physical one including a track index, a cylinder index and overflow areas which are transparent to the user.

Now we are in a position to precisely define a data base management system, D.

$$\text{Let } \mathcal{B} = \bigcup_{i=1}^{q} \mathcal{B}_i \text{ where } \mathcal{B}_i = \underbrace{P \times P \times \ldots \times P}_{i} \text{ and } \mathcal{L} = \bigcup_{i=1}^{m} \mathcal{L}_i$$

$$\text{where } \mathcal{L}_i = \underbrace{H \times H \times \ldots \times H}_{i}.$$

Here, q and m are upper bounds on the possible number of physical and logical files respectively. Moreover, let $B \subset \mathcal{B}$ and $L \subset \mathcal{L}$ be subsets of $\mathcal{B}$ and $\mathcal{L}$, respectively. Note that a logical data base, ℓ, is an element of $\mathcal{L}$ and a physical data base, b, an element of $\mathcal{B}$.

Definition 5: A <u>data base management system</u>, D, is a set of functions $X = \{x_1, \ldots, x_s\}$ and $Y = \{y_1, \ldots, y_r\}$ such that for all i,

$$x_i : B_{x_i} \rightarrow \mathcal{L}$$

$$y_i : L_{y_i} \rightarrow \mathcal{B}$$

$$\text{where } B_{x_i} \subset \mathcal{B} \text{ and } L_{y_i} \subset \mathcal{L}$$

A data base management system is a set of functions which map collections of logical files into collections of physical files and vice versa. Unlike ISAM which supports only one physical to logical transformation, many systems allow a set of such functions. By varying the physical to logical transformation, this feature will allow different logical data bases to correspond to a given physical data base. At any given moment, however, only one member of X is used. This is one portion of the state of the data base management system defined as follows.

Definition 6: The state, S, of a data base management system, D, is the four tuple (b,ℓ,x_i,y_j) where

$b \in \mathcal{B}$ is a set of physical files

$\ell \in \mathcal{L}$ is a set of logical files

$x_i \in X$ is the current physical to logical transformation

$y_j \in Y$ is the current logical to physical transformation

User interactions with ℓ must be mapped into interactions with b and any result (as in a retrieval request) must be mapped from b back to ℓ. Hence, the data base management system is completely specified by the current sets of logical and physical files and by the two transformations currently in use to map between them.

Definition 7: A state $S=(b,\ell,x_i,y_j)$ is consistent if $b \in B_{x_i}, \ell \in L_{y_j}$ and $x_i(b)=\ell$, $y_j(\ell)=b$.

Note that a state is consistent if the two transformations currently in effect successfully map back and forth between the current sets of physical and logical files. Maintaining consistency is one goal of all data base management systems.

There are two basic ways that a change in the state of data base management systems can be instituted.

1. A state change can be caused by a user making a change to the logical data base. In this case, $\ell \in \mathcal{L}$ is changed to $\ell' \in \mathcal{L}$ by an update, insertion or deletion. The addition and deletion of logical files also fall in this category.

2. The physical data base may be changed (for efficiency reasons, for example). In this case, $b \in \mathcal{B}$ is altered to $b' \in \mathcal{B}$.

After either type of change, the state of the system may be consistent or inconsistent. In case 1, a consistent state may be created in three ways:

a) by changing the physical data base i.e. $S=(b,\ell,x_i,y_j) \rightarrow S'=(b',\ell',x_i,y_j)$ where S' is consistent. This approach will work, in general, only if $x_i=y_j^{-1}$.

b) by changing the data base transformations i.e. $S=(b,\ell,x_i,y_j) \rightarrow S'=(b,\ell',x_m,y_n)$ where S' is consistent.

c) by changing both the physical data base and the transformations i.e. $S=(b,\ell,x_i,y_j) \rightarrow S'=(b',\ell',x_m,y_n)$ where S' is consistent.

In all cases we note that ℓ changes to ℓ' and that a user program U may or may not execute correctly on ℓ' given that it executes correctly on ℓ. Little can be said further concerning insertions, deletions and updates except that U should be made to survive as many such changes as possible and that there exist ℓ' on which U will execute incorrectly.

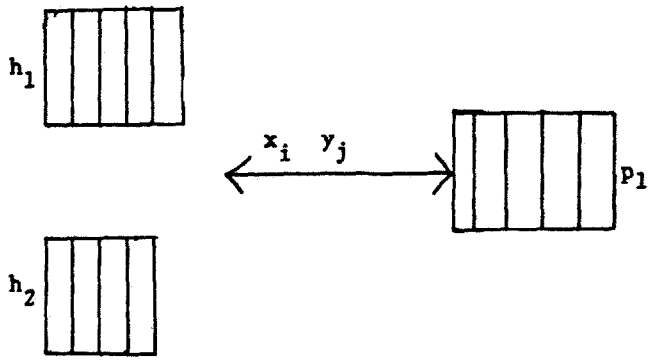
Consider for example the situation of Figure.3. Here, two logical files exist each with one address and 5 and 4 byte contents, respectively. Moreover, h_1 and p_1 have the same contents while h_2 deletes the first byte of p_1. In particular, h_1 and h_2 might be 5 and 4 character representations respectively of a data item stored in p_1. The transformations x_i and y_j can be appropriately defined. If the first byte of p_1 is blank, then users interacting with h_1 can run correctly. On the other hand, if p_1 is updated by a user interacting with h_1 to have a non-blank first byte, then the resulting data base can be made consistent yet user programs interacting with h_2 will in all probability execute incorrectly.

In the case where a logical file, h*, is added by a user, one can guarantee that other users are not affected by simply creating a separate physical file, p*, for that logical file with its separate x_i^* and y_j^*. This can either be a separate data base i.e. $S=(b,\ell,x_i,y_j) \rightarrow S=(b,\ell,x_i,y_j)+S'=(p^*,h^*,x_i^*,y_j^*)$ or an addition to S, i.e. $S=(b,\ell,x_i,y_j) \rightarrow S'=((b,p^*),(\ell,h^*),(x_i,x_i^*),(y_j,y_j^*))$. For efficiency reasons, one might wish to perform a change of case 2 to the resulting data base.

In case two, a consistent data base may be maintained three ways:

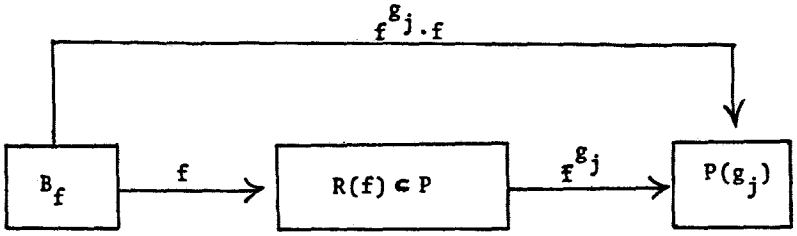
a) by changing ℓ to ℓ' i.e. $S=(b,\ell,x_i,y_j) \rightarrow S'=(b',\ell',x_i,y_j)$ and S' is consistent. As before, user programs may or may not survive such changes.

b) by changing x_i and y_j, i.e. $S=(b,\ell,x_i,y_j) \rightarrow S'=(b',\ell,x_m,y_n)$ and S' is consistent. If S' is made consistent in this fashion, then user programs will assuredly survive the change from b to b'.



A Trivial Data Base

Figure 3



The Transformations of Interest

Figure 4

c) by changing ℓ, x_i and y_j, i.e. $S=(b,\ell,x_i,y_j) \rightarrow S'=(b', \ell',x_m,y_n)$ and S' is consistent.

Cases 1a, 1b, 1c, 2a and 2c all involve changes to the logical data base. Whether user programs survive such changes depends entirely on their internal characteristics. Data independence for such changes does not involve the data base management systems, D. As a result, we shall henceforth be concerned with changes to physical data bases which leave the logical data base unaltered (i.e. case 2b).

Let $f:B_f \rightarrow \mathcal{B}$ be a function mapping a physical data base into another physical data base. We now propose two definitions of data independence over such a storage transformation.

Definition 8: A data base management system, D, provides <u>data independence for a function</u> $f:B_f \rightarrow \mathcal{B}$ <u>and state S</u> $= (b,\ell,x_i,y_j)$ if there exist x_m and y_n such that $(f(b),\ell,x_m,y_n)$ is consistent.

Definition 9: A data base management system, D, provides <u>data independence for function</u> $f:B_f \rightarrow \mathcal{B}$ <u>and functions</u> x_i,y_j if for each $b \in B_f$ there exists an x_m and y_n such that the consistency of (b,ℓ,x_i,y_j) implies the consistency of $(f(b),\ell,x_m,y_n)$.

Note that definitions 8 and 9 require functions whose domain is all physical files. Clearly, a function f whose domain is less than all physical files can be uniquely expanded to a function on all physical files by applying the identity mapping to the remaining files. Consequently, these definitions can be trivially modified to allow functions on smaller domains. Such functions will appear in the next section.

We now examine seven classes of functions over which existing proposed and possible systems can be made data independent according to one or both of the above definitions.

The first three classes of transformations map $B_f \subset P \rightarrow P$. Consequently, they alter the storage of a single physical file.

III. Data Independence Classes

Class 1 - the set of one-to-one functions $f:B_f \rightarrow P$ such that if $p'=f(p)$, then

1) p and p' have the same number of addresses

2) for each address, a_i, in p there exists one address a'_j in p' such that $C_A(a_i)=C_A(a'_j)$

Functions in this class allow physical files to be relocated and their addresses to be permuted. Data base management systems usually provide independence over many of the functions in this class. Some systems, however, require addresses in a physical file to be contiguous. Hence, functions which can create a p' violating this constraint would not be supported data-independently. At least one file system [6] partly relaxes this constraint. This first class of data independence might appropriately be called "device independence."

In order to introduce class 2, we require the concept of a non-redundant transformation. Consider a function $f:B_f \subset P \to P$ and let R(f) be the range of f. Let $P(g_j) \subset P$ be the following set.

$$P(g_j)=\{p\text{-}(a_j,g_j) \mid p\in R(f) \wedge P_i \text{ for } i\geq j\} \cup \{p \mid p\in R(f) \wedge P_i \text{ for } i<j\}$$

Here, g_j is all or any portion of the contents of $C_A(a_j)$ and $P(g_j)$ results from R(f) by deleting this information. Define $f^{g_j}:R(f) \to P(g_j)$ to be the obvious restriction of R(f) to $P(g_j)$. Denote by $f^{g_j}\cdot f:B_f \to P(g_j)$ the composition of f^{g_j} and f. Figure 4 diagrams these transformations.

Definition 10: A function $f:B_f \to P$ is <u>non-redundant</u> if f is one-to-one but $f^{g_j}\cdot f$ is not one-to-one for any f^{g_j}.

Non-redundant transformations have the property that nothing can be discarded from their range without destroying the one-to-one property. Consequently, they create files with exactly the same information in them as the original ones. A situation not satisfying this property is the transformation made to a sorted file to create an ISAM file. Here, the ISAM cylinder index may be entirely discarded without affecting the possibility of recreating the original sorted file.

Class 2 - the set of non-redundant functions $f:B_f \subset P \to P$ such that if p'=f(p) then

1) p and p' have the same number of addresses
2) for each address, a_i in p there exists one address a'_j in p' such that $(a_i,C_A(a_i))$,

$$(a'_j, C_A(a'_j)) \in B_f \text{ and } f(a_i, C_A(a_i)) = (a'_j, C_A(a'_j))$$

Functions in class 2, like those in class 1, require that there be a one-to-one correspondence between addresses in p and those in p'. However, the contents of a location in p' can be a non-redundantly recoded version of its counterpart in p. A mechanism to support a restricted subset of such coding functions in a data independent way was suggested in MacAIMS [8].

One can easily note that class 1 functions are a special case of class 2 functions. In Figure 5, this inclusion property is noted by an arrow from class 2 to class 1. Both classes 1 and 2 have the requirement that p and p' must have the same number of locations. Class 3 allows more general transformations, as follows.

Class 3 - Non-redundant functions, $f: B_f \subset P \rightarrow P$

Three examples of such transformations are:

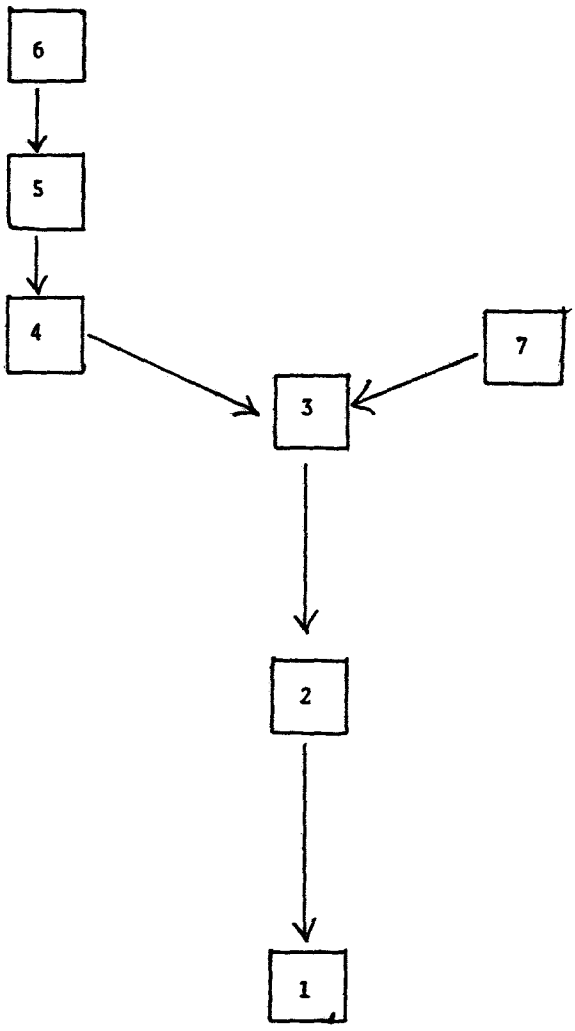
1) Storage of a relation changed from row-by-row to column-by-column.
2) A coding scheme whereby each location stores not $C_A(a_i)$ but the difference between $C_A(a_i)$ and $C_A(a_{i-1})$ (which may require less space).
3) Blocking of records together to form larger physical records. Each of these three transformations is non-redundant and hence belongs in class 3. Other examples of class 3 transformations are the following:

If p contains pointers as data items which allow a logical tree to be constructed, then a class 3 function is a non-redundant transformation from one representation of a tree to another. See [9] for the various implementations. Note that normalizing a tree structure [10] is also a transformation of class 3.

Transformations in classes 1-3 are all non-redundant. Moreover, each maps a single file into another one. Initially, if there is a one-to-one correspondence between physical files and logical files, then transformations in these classes preserve that property.

We now turn to transformations which may violate these conditions. Class 4 will contain those transformations which create extra indices.

In general terms, an index for $p \in P$ is a collection of subsets of addresses in p, each subset of which shares some common property. For example, for each value of a given data item, one could store addresses of all locations in p with the data item having the required value. Alternately, for each location of p,



The Hierarchy of Data Independence

Figure 5

one could store the location of the one other element having the next value in the collating sequence for a given data item. Hence, an index shall be precisely defined as follows for $B_f \subset P$.

Definition 11: An index is a function $f:B_f \rightarrow P$ such that if $p'=f(p)$ then p' differs only by a class 3 function from a file each of whose addresses contains a subset of addresses in p and a subset of the contents of those addresses.

Class 4 functions map $B_f \subset P$ into $B_2(P_1,P_2)$ as follows.

Class 4 - the set of functions $f:B_f \rightarrow (P_1,P_2)$ such that $f=(f_1,f_2)$; $f_1:B_f \rightarrow P_1$ and $f_2:R(f_1) \rightarrow P_2$ such that

1) f_1 is a class 3 function
2) f_2 is an index

Note that a trivial modification of the above definition would allow the possibility of $p_1 \varepsilon P_1$ and $p_2 \varepsilon P_2$ to be subsets of the same physical file as is the case when multilist structures are created.

Of course, if there are no subsets in an index and $p_2=\phi$, then class 4 reduces to class 3, as noted in Figure 5. Class 5 transformations are an obvious extension of class 4 functions.

Class 5 - the set of functions $f:B_f \subset P \rightarrow (P_1,P_2)$ such that $f=(f_1,f_2)$; $f_1:B_f \rightarrow P$ and $f_2:B_f \times R(f_1) \rightarrow P_2$ such that

1) f_1 is a class 3 function
2) f_2 is a function

Note in class 5 that f_2 is not restricted to be an index.

One class 5 transformation is the duplication of an entire file. Another is any transformation which creates redundant pointers in a tree structure for faster access (assuming the trivial modification to class 5 which would allow P_1 and P_2 to be subsets of the same physical file). Clearly, Class 4 is a special case of class 5 as noted in Figure 5.

Transformations exist in classes 4 and 5 which introduce redundancy. However, after a sequence of class 4 and 5 transformations each file in the set of files created can be uniquely

associated with a file in the original set. Class 6 and 7 functions do not require this condition. Class 6 functions map $B_f \subset \mathcal{B}$ into $\mathcal{B}$ as follows.

Class 6 - the set of one-to-one functions $f:B_f \rightarrow \mathcal{B}$ such that if $b \in B_f \cap \mathcal{B}_i$ then $f(b) \in \mathcal{B}_{i+1}$.

Here, i+1 physical files are created from i original ones. However, it is possible that none of the original files can be recreated from a single resulting file. Hence, more general mappings are allowed than can be formed by i class 5 functions. This fact is noted in Figure 5. Two examples of class 6 transformations are the following.

1) Replacing the stored representation of a relation by two of its projections (Under certain conditions this operation is one-to-one.)
2) For a file of variable length records, reducing the record length to less than the maximum required and chaining overflows into a separate file.

The final class of transformations allows physical files to be combined. Here, f will map $B_f \subset \mathcal{B}$ into $\mathcal{B}$.

Class 7 - the set of one-to-one functions $f:B_f \rightarrow \mathcal{B}$ such that if $b \in B_f \cap \mathcal{B}_{i+1}$ then $f(b) \in \mathcal{B}_{i+1}$.

It is easily seen that class 7 contains class 3. A class 3 function occurs if $B_f=(P_1,\phi)$, (i.e., the second physical file is empty), and f is a non redundant function on P_1. Examples of class 7 transformations are the inverses of the two class 6 example transformations. (The inverse for example 1 is the natural join operation.)

Figure 5 illustrates the 7 classes of transformations. When one class contains another class as a subset, this has been indicated by a directed arc from the class to its subset. Note that the classes form a natural hierarchy.

As a general rule, it is claimed by the author that the directed graph of Figure 5 is also indicative of the difficulty in supporting transformations in the various classes data-independently. Usually, for example, class 7 transformations are harder to support than class 3 transformations.

It is also claimed that the data independence provided by any data base management system can be precisely specified in terms of the set of transformations from the various classes which satisfy definition 8 or 9 for that system.

IV. A Primitive Set of Transformations

We turn now to suggesting a primitive set of transformations which appear to provide data independence over most commonly used storage structures. We deal with the situation where users have a relational view of data and suppose that each logical file contains one relation. We also suppose that each physical file contains a storage representation of one relation. The data base management system, D, supports a class of transformations between physical relations and logical relations. We denote by y^* the transformation which maps each logical relation into a corresponding physical file containing a tabular representation of the logical relation as if it were the only relation to be stored. We also denote by x^*, the inverse of y^* and by $p \in P$, the tabular representation of some relation.

We now propose a set of functions, τ, which D can be made to support according to Definition 8 for x^*, y^* and ℓ. τ will be meaningful compositions of the following sets of functions.

1. the set of functions relocating a contiguous physical file to another contiguous set of locations. (class 1)
2. application of a coding scheme such as described in [8] to all or some portion of the contents of addresses in a physical file. (class 2)
3. transformation from p to a particular representation of a tree structure. (class 3 or 5)
4. the inverse of 3. (class 3)
5. transformation from p to a particular representation of a hash table. (class 3)
6. the inverse of 5. (class 3)
7. addition of a redundant index for any set of domains in a relation. (class 4)

The following example indicates one implementation. Consider the relation R(Supplier, Part#, Job, QOH) and suppose a representation of the following table is stored.

Supplier	Part #	Job	QOH
1	2	3	4
1	2	2	5
2	2	1	3
2	2	2	2

An index for (QOH, Job) could be the storage representation of the following.

QOH	Part #	Pointer
2	2	●⟶ to 4th tuple of R
3	1	●⟶ to 3rd tuple of R
4	3	●⟶ to 1st tuple of R
5	2	●⟶ to 2nd tuple of R

The 8th and 9th operations require the following motivation. At least one relational query language, QUEL [15], allows any legal query to be a class 6 function. Here, a new relation satisfying the query is created. More precisely, let q be an element of Q_ℓ the set of legal queries for the existing logical data base. Hence, $q:B_q \rightarrow \mathcal{P}$ such that if $b \in B_q$ then $q(b)=(b,p)$ where p is a physical file containing the answer to q. Note that q is a one-to-one function.

8. the set Q_ℓ for a complete query language [11]. (class 6)
9. the inverses of those functions in 8. (class 7)

It is claimed that τ can be supported without undue difficulty. In fact, one system being implemented [15] will support a somewhat larger class than τ.

V. Examples

We now present examples of the data independence provided by three systems. In all cases we deal with existing systems so that the difference between wishful thinking (since every level of data independence we have talked about is theoretically possible) and actuality does not affect us.

We choose RDMS [12], ISAM [7] and IMS [13,14] for our examples. The following table indicates the transformations provided by the various systems data independently.

Note the limited data independence provided by all systems compared to that suggested in the preceding example. It is hoped that existing systems will move in the direction suggested by that example.

A Functional View of Data Independence

	RDMS	ISAM	IMS
class 1	to limits provided by STAR operating system	to limits provided by OS/360	to limits provided by OS/360
class 2	none	none	none
class 3	storage of a relation by row or by column	various blocking factors	various blocking factors
class 4	none	master index cylinder index	none
class 5	none	none	various redundant pointers can be created for hierarc ical structures
class 6	none	none	*
class 7	none	none	*

* IMS supports the following multifile to multifile transformations

1) transformations among HSAM, HISAM, HDAM, and HIDAM.
2) transformations which map the storage representation of an IMS graph structure into another graph structure for which all logical trees can be obtained by deleting nodes.

Table 1

REFERENCES

1. Sibley, E. and Taylor, R., "A Data Definition and Mapping Language," CACM, Vol. 16, No. 12, December 1973.

2. Collmeyer, A., "Implications of Data Independence on the Architecture of Data Base Management Systems," Proceedings of the 1972 ACM-SIGFIDET Workshop on Data Description, Access and Control, Denver, Col., November, 1972.

3. Date, C. and Hopewell, P., "File Definition and Logical Data Independence," Proceedings of the 1971 ACM-SIGFIDET Workshop on Data Definition, Access and Control, San Diego, Ca. November 1971.

4. Date, C. and Hopewell, P., "Storage Structure and Physical Data Independence," Proceedings of the 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, Ca., November 1971.

5. Brinch Hansen, P., "The Nucleus of a Multiprogramming System," CACM, Vol. 13, No. 4, April 1970.

6. Richie, D. and Thompson, K., "The UNIX Time Sharing System" Proceedings of the 5th Operating System Symposium, Yorktown Heights, N.Y., October 1973.

7. "OS ISAM Logic"; IBM Corp., No. GY 28-6618.

8. Goldstein, R. and Strnad, A., "The MacAIMS Data Management System," Proceedings of the 1970 ACM-SIGFIDET Workshop on Data Description and Access, Houston, Texas, November, 1970.

9. Knuth, D., The Art of Computer Programming, Vol 1, Addison Wesley, Reading, Mass. 1969.

10. Codd, E., "A Relational View of Data for Large Shared Data Banks," CACM, Vol. 13, No. 6, June 1970.

11. Codd, E., "Relational Completeness of Data Base Sublanguages," Report RJ 987, IBM Research, San Jose, Ca., March 1972.

12. Whitney, V., "RDMS: A Relational Data Management System," Report CS 80, General Motors Research, Warren, Mich. December 1972.

13. "Information Management System/360, Version 2 System/Application Design Guide," IBM Corp., No. SH20-0910.

14. "Information Management System/360, Version 2 Utilities Reference Manual," IBM Corp., No. SH 20-0915.

15. MacDonald, M., Stonebraker, M., and Wong, E., "Preliminary Specification of INGRES," Electronics Research Laboratory, University of California, Berkeley, Technical Report No. 740, May, 1974.