NETWORKS, HIERARCHIES, AND RELATIONS IN DATA BASE MANAGEMENT SYSTEMS

by

Michael Stonebraker and Gerald Held

Memorandum No. ERL-M504

3 March 1975

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

NETWORKS, HIERARCHIES AND RELATIONS IN DATA BASE MANAGEMENT SYSTEMS

Michael Stonebraker and Gerald Held
Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, California 94720
(415) 642-4871

This paper attempts to clearly separate two issues that are often confused in a data base management system context, namely: 1) the appropriate level of a data sublanguage, 2) the model of the data base employed by the user. To this end the three models of data that currently prevail are described. One is the relational view of data proposed by E. F. Codd whereby all data is seen as relations of assorted degrees. The second is the network view advanced by CODASYL in which data is perceived to be the nodes of a graph. A third view is a hierarchical one which is a special case of the network model in which data is restricted to being represented as a tree.

For each view two examples of data sublanguages are presented. One is a lower level procedural language, and the other a high level non-procedural language having no control statements or statements to test for exceptions. It is seen that non-procedural languages for all three views of data are very similar and all hide the details concerning what data structures and access paths are actually implemented

As a result, the main issue to be resolved is the appropriate level of a data sublanguage. CODASYL has advanced a lower level procedural language while relational advocates suggest high level non-procedural ones.

This paper examines the CODASYL Data Manipulation Language (DML) and one relational language and indicates the advantages that each enjoys. These advantages suggest that both language levels are likely to play an important role in data base management systems of the future.

I. INTRODUCTION

What data model to use in a data base management system is currently a hotly debated topic. In fact, a formal debate was held at a recent ACM-SIGFIDET workshop on data description, access and control. At that time Codd and Date [1,2] presented a case that data should be viewed as a collection of relations of assorted degrees. This position was also defended by Whitney [3]. Bachman [4], Sibley [5] and Lucking [6] discussed the advantages of considering data as the nodes of a graph.

Referring to Table 1 we note the three main models presently advocated. For each model we consider two possible language levels and indicate where various data base systems (and proposed data sublanguages) belong. The tabulation is by no means exhaustive.

	RELATIONAL	HIERARCHICAL	NETWORK
high level non procedural	INGRES[7] SEQUEL[8] ALPHA[9] SQUARE[10]	SYSTEM 2000[13] DATA LANGUAGE[14] OTSS[15] HQL[16]	ADABAS[18]
low level procedural	XRM[11] ISAM[12]	IMS[17]	CODASYL[19, 20,21] IDMS[22]

VIEWS OF DATA

TABLE 1

In Sections 2-4 of this paper we will examine each column of Table 1 and indicate an example of a language in each box. In Section 5 we will make some observations concerning the six languages discussed. Among other things we will indicate that high level non-procedural languages using all three views are nearly identical

As a result of this discussion, it can be concluded that the major difference between the positions advanced by the relational and network advocates is one of the appropriate level of the data sublanguage.

Table 2 shows certain points of comparison between a lower level procedural language (as typified by the CODASYL Data Manipulation Language (DML)) and a high level non-procedural language (as typified by the relational language QUEL).

Sections 6-10 justify each entry of Table 2 individually. Lastly, in Section 11 conclusions are drawn and one possible scenario that might evolve in the future is indicated.

ADVANTAGES OF NON-PROCEDURAL LANGUAGES	1) Less source language statements to do a task	
	2) Data independence	
	3) Sophistication of protection and integrity	
ADVANTAGES OF PROCEDURAL LANGUAGES	1) Efficiency	
ADVANTAGES OF NEITHER	1) Expressiveness of data structures	

COMPARISON OF TWO PREVAILING LEVELS OF LANGUAGE FOR DATA BASE APPLICATIONS

TABLE 2

II. THE HIERARCHICAL APPROACH

We treat the example data base shown in Figure 1. Indicated there are four record types with connections that form a tree. Also identified are the data items present in each record type. Each department in an organization has one DEPT record. At the second level of the hierarchy are records for all employees who work in a given department. At the third level are two record types, one containing information on the children of each employee and the other indicating

information on the office in which the employee works. In Fig. 1 the instance of DEPT corresponding to department 17 is also shown. Also present are the three instances of EMP for employees in department 17. Below these instances are instances of CHILD and OFFICE where appropriate. Note that all instances shown (except for the instance of DEPT) have exactly one parent record instance.

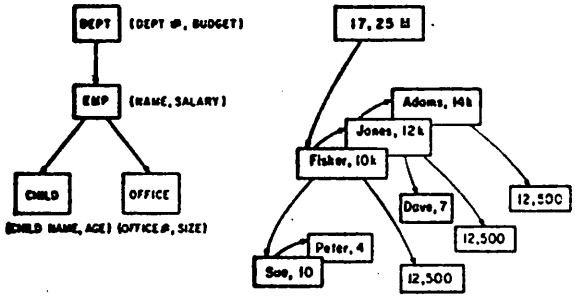

Figure 1. A Hierarchical Structure.

More formally, the following definitions are appropriate for hierarchical systems:

A RECORD TYPE is a user defined interpretation of a string of bytes.

An INSTANCE of a record type is a byte string of the appropriate length interpreted in the appropriate way.

A HIERARCHY is a collection of record types which form a tree.

A HIERARCHICAL DATA BASE is a collection of instances of records such that each instance (except instances of the record type at the root of the hierarchy) has exactly one parent. Moreover, its parent is of the same type as its parent record type in the hierarchy.

A procedural language operating on a data base with CHILD, OFFICE, EMP and DEPT record types could look something like the following:

EXAMPLE 2.1 FIND THE NAMES OF ALL EMPLOYEES IN DEPARTMENT 17

```
      FIND DEPT RECORD WHERE DEPT# = 17
         if failure; return "no such department"
      FIND 1ST SON OF CURRENT RECORD
         if failure; return "no employees in this
               department"
LOOP     save name
      FIND NEXT BROTHER OF THE CURRENT RECORD
            WHICH IS OF THE SAME TYPE
         if failure; return "done"
      GO TO LOOP
```

Basically, the programmer "navigates" [23] the tree finding instances of qualifying records and saves (somewhere) the desired information. This language requires control statements (e.g. GO TO LOOP) and tests for failure of instructions (e.g. the three IF statements). It is a procedural language because the programmer creates an algorithm that only requires execution to solve his problem. Stated differently, the FIND statement returns EXACTLY ONE instance of a record each time it is called. Hence, the programmer must execute a proper sequence of FIND commands to obtain needed information.

A non-procedural language answering the same interaction might look like the following.

EXAMPLE 2.2

```
      FIND ALL NAME OF EMP
        CONTAINED IN DEPT
           WHERE DEPT# = 17
```

In this case the FIND statement returns information from all records satisfying the indicated condition at once. Basically, instead of the programmer "navigating" the tree structure, the data base system must transform the above statement into a navigation algorithm (such as the one given in EXAMPLE 2.1) which can then be executed.

III. THE NETWORK APPROACH

One observes a problem with hierarchical systems when dealing with data having the structure of Figure 1 namely: office information must be repeated for each employee that works in a given office. In Figure 1 (12,500) must be repeated three times. This wasteful repetition is forced on one by the requirement that each instance of a record have exactly one parent. The network approach indicated in Figure 2 avoids this restriction.

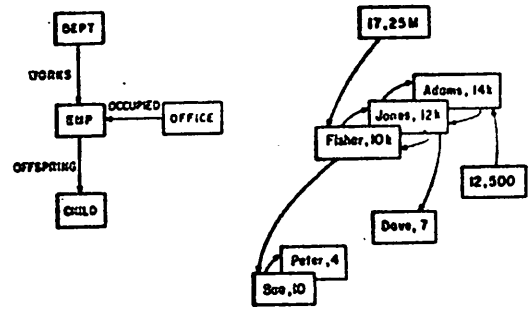

Figure 2. A Network Structure.

Carefully compare Figures 1 and 2 and note that (12,500) is not duplicated in Figure 2 but only at the expense of an instance of a record having more than one parent.

More formally, the following definitions can apply to network systems:

A RECORD TYPE is a user defined interpretation of a string of bytes.

An INSTANCE of a record type is a byte string of the approriate length interpreted in the appropriate way.

A SET TYPE is an association between one owner record type and one or more member record types.

An INSTANCE of a set type is a binary relation between one instance of an owner record type and zero or more instances of member record types such that an instance of a member record type can be related to at most one instance of the owner record type.

A NETWORK DATA BASE is a collection of record instances and set instances.

We now indicate an example of a procedural network language.

EXAMPLE 3.1 FIND THE DEPARTMENT NUMBERS OF ALL EMPLOYEES IN OFFICE # 12

```
      FIND OFFICE RECORD WHERE OFFICE# = 12
```

```
            if failure; return "no such office"
LOOP        FIND NEXT MEMBER OF OCCUPIED SET
            if failure; return "done"
            FIND OWNER OF CURRENT EMPLOYEE RECORD USING
                  WORKS SET
            if failure; return "employee exists that
                  is not in a department"
            save dept number
            GO TO LOOP
```

This code is not dissimilar to that in Example 2.1. However, the "navigation" has become somewhat more complicated because the programmer can follow any of the set instances that involve a record instance.

A higher level network language might execute the same problem as follows:

EXAMPLE 3.2

```
FIND ALL DEPT# IN DEPT
   WHERE SOME EMP RECORD VIA WORKS SET
      HAS OWNER VIA OCCUPIED SET
         WHERE OFFICE# = 12
```

Again the programmer has been spared the navigation of the data base and the data base system must provide the algorithms. ADABAS offers a very restricted version of a language with this flavor.

IV. THE RELATIONAL APPROACH

In the relational view the only data structure allowed is a relation, and a data base is simply a collection of such relations. Intuitively, a relation is nothing but a (perhaps giant) array not unlike those found in FORTRAN or APL[24].

There are, not surprisingly, several possible collections of relations that express the same information as in Figures 1 and 2. Figure 3 indicates one such collection. Note that a certain amount of redundancy is present in Figure 3; for example, NAME appears three times. We will discuss this issue of redundancy again in Section 8.

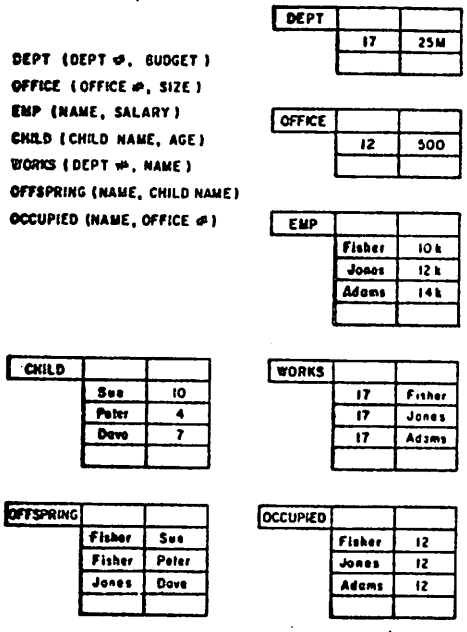

Figure 3. A Relational Structure.

More precisely, given sets $D_1,\ldots,D_n$ a RELATION is a subset of the Cartesian product $D_1*\ldots*D_n$. In other words, R is a set of TUPLES (Ti) where Ti = $(Ti_1,\ldots,Ti_n)$ and Ti_j is an element of D_j. The sets $D_1,\ldots,D_n$ are called DOMAINS. Usually relations are restricted to being NORMALIZED i.e., the members of $D_1,\ldots,D_n$ cannot themselves be relations. Further properties of relations are discussed in [25-29]. We now do Examples 3.1 in two relational languages. In a procedural language it might resemble:

EXAMPLE 4.1

```
            FIND FIRST OCCUPIED TUPLE WHERE OFFICE# = 12
                  if failure; return "no such office"
LOOP              save name
            FIND WORKS TUPLE WHERE NAME = saved name
                  if failure; return "employee exists
                        that is not in a department"
                  save dept number
            FIND NEXT OCCUPIED TUPLE WHERE OFFICE# = 12
                  if failure; return "done"
            GO TO LOOP
```

In a non-procedural language the query might be programmed as follows:

EXAMPLE 4.2

```
FIND ALL DEPT# IN WORKS
     WHERE NAME =
           NAME IN OCCUPIED
                 WHERE OFFICE# = 12
```

V. OBSERVATIONS

1) A hierarchical data base can be thought of as a special case of a network data base in which each instance of a record type (except instances of the root record type) is a member of exactly ONE instance of a set type. This set type can be thought of as PARENT OF RECORD or CONTAINED IN.

2) A relational data base can be thought of as a special case of a hierarchical data base in which each record instance is a member of ZERO sets.

3) All procedural languages look much the same as they all have control statements, search statements and exception testing statements. They differ only in the navigational possibilities for the programmer: the programmer navigates a structure where records belong to no sets in a relational system, to one set in a hierarchical system and to perhaps more than one set in a network system.

4) All non-procedural languages look much the same. They differ mainly in the way records (or tuples) can be associated. In a relational system tuples are associated by specifying a relationship which must hold among domains (in the example above, equality on NAME in two different relations). In a hierarchical system, record instances are associated by membership in the set CONTAINED IN. In a network system record instances are associated by participation in a given set.

5) Note very clearly that a non-procedural hierarchical programmer cannot tell the difference between:

a) an implementation such as Figure 1 and a data base system that converts interactions such as Example 2.2 into ones such as 2.1 which it executes.

b) an implementation in which an interaction is translated into a statement in a non-procedural relational language. If CONTAINED IN is considered a binary relation for each record type and each record type is considered a relation, this translation is straightforward. How these relations are actually implemented need not be

visible to the non-procedural relational programmer, let alone the non-procedural hierarchical programmer.

A similar statement holds for the non-procedural network programmer. Therefore, the following conclusion can be safely drawn:

ALL THREE TYPES OF NON-PROCEDURAL PROGRAMMERS CAN, IN FACT, BE DEALING WITH FRONT ENDS TO A RELATIONAL DATA BASE SYSTEM.

6) It is entirely possible that a non-procedural relational system may translate certain interactions into a non-procedural network or hierarchical language which may be implemented by algorithms such as those required to translate Examples 2.2 and 3.2 into Examples 2.1 and 3.1. As a result the following conclusion can be drawn:

A NON-PROCEDURAL RELATIONAL PROGRAMMER MAY, IN FACT, BE DEALING WITH A FRONT END TO A HIERARCHICAL OR NETWORK DATA BASE SYSTEM.

7) As a result of the above observations we suggest that any of the non-procedural programmers can usually be accomodated on any of the three types of data bases. All are shielded from the details concerning how data is stored and what access paths are present. Hence, it is not a crucial issue to resolve which data model is preferable.

Since the non-procedural network or hierarchical programmer can see only a limited number of inter-record associations (sets), he has less flexibility than the non-procedural relational programmer who is free to associate any domains in any relations. As a result it may be preferable to allow the flexibility of a relational system. Non-procedural users who feel more comfortable with languages such as in Examples 2.2 and 3.2 can be accomodated in many situations without difficulty by software on top of a relational system.

On the other hand, non-procedural network or hierarchical systems may have implementations that support different internal data structures than relational systems (for instance, sets). In certain situations, such structures may result in a performance advantage. The issue of performance is further considered in Section 8.

8) We suggest, however, that a very important issue is whether data base applications should be programmed in procedural languages (such as those of Examples 2.1, 3.1 and 4.1) or in non-procedural languages (such as those of Examples 2.2, 3.2 and 4.2).

Consequently, the next five sections examine individual entries in Table 2. For each entry we contrast the leading candidate for a procedural language (the CODASYL DBTG proposal and its Data Manipulation Language (DML)) with one non-procedural relational system (INGRES and its data sublanguage QUEL).

VI. DATA STRUCTURE EXPRESSIVENESS

QUEL presents the user with a collection of normalized relations as discussed in Section 4 while the CODASYL DBTG Data Manipulation Language (DML) gives him a network structure as in Section 3. The purpose of this section is to briefly show that the expressiveness of the data structures is the same for both systems.

It is clear that a set of normalized relations is also a set of DBTG record types and the tuples of a relation are instances of a given record type. Therefore, any valid relational structure can be transformed to a valid DBTG structure (assuming that each system allows individual data items of the same generality).

The reverse transformation is also assured as noted in [2]. Basically, each set in DBTG is a binary relation between the primary key of instances of the owner record type and the primary key of instances of the member record type. When a set has members of more than one type, a relation must be defined for each type of member. (It might be noted that performance may improve if an alternate transformation is followed; that of simply propagating the primary key of the owner record type to each member record type. This is the same technique used in [25] to normalize unnormalized relations). If a set is defined between record types that do not have primary keys, the procedure is somewhat more complicated and involves a generalization of the above key propagation technique.

A given record type in DBTG is a simple hierarchy; consequently, the algorithm in [25] can be used to normalize such a structure to a collection of relations. As a result any DBTG structure can be transformed to a group of relations.

VII. CODING EFFICIENCY

From the previous sections it is clear that non-procedural languages result in a significant savings of source code. This results partly from the absence of control and exception testing statements and partly from the increased scope of FIND statements. Other examples which illustrate the same point are presented in [2]. Presumably, programmer productivity will increase substantially if a procedural system is replaced by a non-procedural one. We suspect productivity would increase at least as much as (if not much more than) the amount of code reduction.

We see non-procedural languages as part of a trend over the past twenty years toward higher and higher level languages. This is illustrated in Figure 4 for general purpose programming languages and in Figure 5 for data sublanguages.

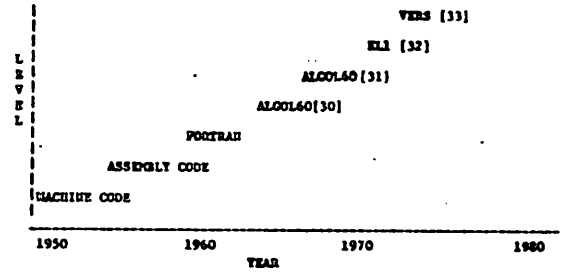

Figure 4

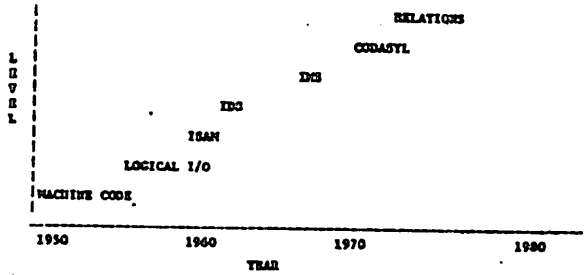

Figure 5

VIII. MACHINE EFFICIENCY

In this section we are restricted to making conjectures concerning the relative performance of non-procedural and procedural systems since no performance comparisons have been attempted. (Later this year such a test between IDMS and INGRES may be possible.)

Our conjecture (contention) is that situations exist for which a DBTG implementation will run faster than an implementation of a non-procedural language such as QUEL. In order to elaborate, some implementation details on both kinds of systems must be introduced.

Included as Figure 6 is a simplified version of some of the required data definition statements for a DBTG version of Figure 2. Note that DEPT and OFFICE are accessed by address calculation on a key (respectively DEPT# and OFFICE#) and that EMP and CHILD are normally accessed by following respectively the OCCUPIED and OFFSPRING sets.

Implementation of sets is expected by pointer chains or pointer arrays [34]. Placement of CALC records is by a hashing algorithm on the given key [35,36] while placement of VIA records is as close as possible to their owner record in the indicated set.

```
RECORD NAME IS DEPT
LOCATION MODE IS CALC USING DEPT#
    DEPT#
    BUDGET

RECORD NAME IS EMP
LOCATION MODE IS VIA OCCUPIED
    NAME
    SALARY

RECORD NAME IS CHILD
LOCATION MODE IS VIA OFFSPRING
    CHILDNAME
    AGE

RECORD NAME IS OFFICE
LOCATION MODE IS CALC USING OFFICE#
    OFFICE#
    SIZE

SET NAME IS WORKS
    OWNER IS DEPT
    MEMBER IS EMP

SET NAME IS OCCUPIED
    OWNER IS OFFICE; ORDER IS SORTED BY DEFINED KEYS
    MEMBER IS EMP; KEY IS ASCENDING SALARY

SET NAME IS OFFSPRING
    OWNER IS EMP
    MEMBER IS CHILD
```

Figure 6. Some of the Required Data Definition Statements.

In Figure 7, we indicate one collection of relations which has the same information as those in Figure 3, but which should offer augmented performance.

```
DEPT(DEPT#, BUDGET)
CHILD(NAME, CHILDNAME, AGE)
OFFICE(OFFICE#, SIZE)
EMP'(DEPT#, OFFICE#, NAME, SALARY)
```

Figure 7. A Collection of Relations.

A QUEL statement is decomposed into an algorithm not dissimilar to the one in Example 4.1 which can then be executed. An individual relation in INGRES is stored using one of five access methods. These are described in detail in [37]. In addition, secondary indices [38-40] to permit augmented performance can be defined on individual relations. Two points should be carefully noted:

1) The basic implementation entity is a relation. Hence, Figure 4 is implemented and not Figure 3.

2) The non-procedural language is decomposed into a language resembling Example 4.1 and not Example 3.1. In particular, there are no sets.

With this prelude we now focus on three situations where INGRES may suffer a performance degradation compared to an implementation of the DBTG proposal.

1) A user may be able to state his interaction in more than one way.

EXAMPLE 8.1 FIND SMITH'S SALARY KNOWING BOTH HIS DEPT AND HIS OFFICE

DBTG-
Two possible access paths can be used:

a) FIND OFFICE record
   follow OCCUPIED set

b) FIND DEPT record
   follow WORKS set

Approach a) is faster than b) (because of the VIA access) and a skilled DBTG programmer can utilize this method.

INGRES-
There are three ways the retrieve can be specified namely:

a) RANGE OF E IS EMP'
   RETRIEVE E.SALARY WHERE E.OFFICE# = value

b) RANGE OF E IS EMP'
   RETRIEVE E.SALARY WHERE E.DEPT# = value

c) RANGE OF E IS EMP'
   RETRIEVE E.SALARY WHERE E.DEPT# = value AND
   E.OFFICE# = value

The relational programmer does not normally know which statement is faster (and in fact the fastest choice may vary as the data base is reorganized). Hence, he may well give an inferior statement of his problem.

2) Non-procedural languages are inappropriate for certain interactions.

EXAMPLE 8.2 FIND THE SECOND HIGHEST PAID MEMBER OF OFFICE 12

DBTG-
FIND OFFICE RECORD WHERE OFFICE# = 12
FIND LAST OF OCCUPIED SET
FIND PRIOR OF OCCUPIED SET
(get and record salary and name)

INGRES-
A messy aggregate is required such as

```
RANGE OF E IS EMP'
RANGE OF E2 IS EMP'

RETRIEVE E2.SALARY, E2.NAME WHERE
  COUNT(E.SALARY BY E2.SALARY
                WHERE E.OFFICE# = 12
                AND E.SALARY ≥ E2.SALARY)
  = 2
```

It is very unlikely that an optimizing language processor can be wise enough to process this aggregate without a performance degradation of orders of magnitude.

In this case, a clue to efficient processing could be provided by including the notion Nth LARGEST in QUEL. The above interaction then might look like:

```
RANGE OF E IS EMP'
RETRIEVE E.SALARY, E.NAME
WHERE E.SALARY =
      2nd LARGEST(E.SALARY WHERE E.OFFICE#
                  = 12)
```

Although a "patch" is possible in this situation, there are likely to be instances where it is very awkward to give such performance hints to the language processor. Also, the number of performance oriented "special cases" may be large.

3) The greater complexity of the DBTG data structures may allow higher performance than available in INGRES.

EXAMPLE 8.3 FIND THE NUMBER OF SQUARE FEET PER EMPLOYEE IN OFFICE 12

DBTG - The set of issued commands resembles the following:

```
    FIND OFFICE RECORD WHERE OFFICE = 12
        save size
        count = 0
LOOP FIND NEXT OF OCCUPIED SET
        if current record is last member of OCCUPIED
             then return size/count
        count = count + 1
    GO TO LOOP
```

Two facts concerning this algorithm should be noted:

a) It is likely that no disk seeks are required to loop through the OCCUPIED set because VIA access attempts to cluster the required set members. In fact, they may all be on the same track (or page) in which case one physical I/O operation may be enough to satisfy all FIND NEXT commands.

b) If a pointer chain is used to implement OCCUPIED, each EMP record contains a direct storage pointer to the next EMP member of the OCCUPIED set. No auxiliary reads are required to obtain the next member of that set. This statement is true whether EMP is organized using CALC or VIA.

INGRES - The set of commands into which the QUEL statement is decomposed looks not unlike the following:

```
    FIND OFFICE TUPLE WHERE OFFICE = 12
        record size
        count = 0
LOOP FIND NEXT EMP' TUPLE WHERE OFFICE = 12
        if none return size/count
        count = count + 1
    GO TO LOOP
```

Two points should be noted about this code:

a) No VIA access is supported in INGRES. Hence, the OFFICE tuple and EMP' tuples required are NOT clustered on the same page. In INGRES at least two disk reads are required.

b) The first FIND can proceed at the same speed as the first DBTG FIND, if OFFICE is hashed on OFFICE# in INGRES. However, only if EMP' uses OFFICE as a key will the required EMP' tuples be clustered and obtained rapidly. Otherwise, a scan of EMP' to find required tuples is required or a secondary index on OFFICE# (if one exists) must be utilized. Either way this is slower than the direct pointers used in DBTG.

In summary, INGRES does not support VIA access and does not implement sets. VIA access could be supported quite easily; however, efficient use of sets would require QUEL to decompose into a language such as Example 3.1 and INGRES to support storage structures such as those indicated in Figure 2.

Although this is a possible implementation approach, an optimizing language processor has yet to be demonstrated. A proposal along this lines is given in [41] and some of the difficulties in this approach are stated in [42]. We hope to see explorations into this and other approaches to implementing non-procedural systems so more definitive statements concerning this third situation can be made.

Three examples have been presented where a DBTG implementation may run faster than a relational system. They are indicative of a small class of interactions for which DBTG is especially well suited. For the remaining interactions INGRES can probably be made to run no slower than a DBTG implementation.

In applications where most applications fit into the "small class" above, DBTG should enjoy a performance advantage. The exact magnitude of this advantage is, of course, application dependent, and we decline to conjecture a specific value.

To conclude this section, we briefly mention the subject of the space required to store a data base in either of the two models. It was noted that a collection of relations with the same information as a network data structure usually had repeated domains. Whether these repeated domains are actually stored depends entirely on the implementation of the relational system. For example, if the system implements a network storage structure, presumably no space penalty is required compared to a DBTG implementation. Since INGRES does not support such storage structures, there may well be repeated domains in that system. However, provisions are made in the access methods to optionally code tuples in order to reduce space requirements [7,37]. A general survey of appropriate compression schemes is given in [43]. Consequently, whether a relational system or a network system requires more space depends at least on the application present, the implementation of the relational system and the compression techniques applied to data in either model.

IX. DATA INDEPENDENCE

We turn now to exploring data independence. Basically, we mean by this term the ability of programs to run correctly after changes have been made in the way data is stored. A discussion of this subject is presented in [44,45,46].

The DBTG proposal supports two forms of data independence:

1. A user program uses a SUBSCHEMA to describe the data base whereas the data base itself is described by a SCHEMA. A record type in a subschema is (almost) restricted to be a proper subset of the data items in some record type in

the schema. Therefore, data items and new record types can be added to a network structure and the appropriate subschemas can simply ignore them. In this way the data base is allowed to grow over time without affecting programs written for previous versions of it.

2. The implementor has some flexibility in the way he implements sets and the address calculation involved in locating records. Presumably these can change without impacting applications programs.

Changes that are NOT supported include:

1. most rearrangement of the logical hierarchy of a record type.

2. any changes to what data items are in what record types and what sets are implemented between what record types.

BASICALLY THE DATA BASE CAN GROW BUT NOT BE RESTRUCTURED IN ANY WAY.

In relational systems the following two mechanisms are supportable:

1. The ability to define views [47-50]. A view is a relation which is not present in the data base but can be defined from relations which are. A subschema consists of "virtual" record types which can be defined from real record types in the schema. A view, however, is not restricted to a proper subset of the domains in a stored relation but can be much more general. HOWEVER, THERE IS GREAT DIFFICULTY SUPPORTING THE UPDATING OF VIEWS THAT ARE MORE GENERAL THAN PROJECTIONS (OR PROJECTIONS AND RESTRICTIONS) OF EXISTING RELATIONS. This problem is discussed in [49,50]. Consequently, support for growth of the number of domains in a relation is easy; support for more general views cannot be easily provided except in RETRIEVE ONLY SITUATIONS.

2. Complete flexibility exists in the way one can implement relations. Since no access paths are seen by the applications programmer, any such changes in storage structures can be supported.

For these reasons we state that both models allow the data base to grow (although the relational approach allows greater flexibility in retrieve only situations). The relational approach allows greater flexibility in structuring the internal data. Hence, we say the relational model supports MODERATELY HIGH data independence while DBTG supports only MODERATE data independence.

X. PROTECTION AND INTEGRITY

Specific protection and integrity schemes for relational systems are presented in [49-52]; the DBTG proposal for this subject is contained in [19,20].

In the relational approach, any user can be restricted to a relation or set of relations that can be created in the interaction language. The mechanism in INGRES is by a PROTECT command with the same syntax as a RETRIEVE statement. For example, user Smith can be restricted to accessing salaries of persons who work in OFFICE# 12 since this relation can be expressed as:

```
RANGE OF E IS EMP'
PROTECT E.SALARY  WHERE E.OFFICE# = 12
```

Similarly, any qualification in the interaction language can be imposed as an integrity condition on the data base. For example, "all employees who work in a department with a budget of at least $25M must earn more than $10,000" can be an integrity constraint since it can be expressed by the following qualification:

```
RANGE OF E IS EMP'
RANGE OF D IS DEPT
INTEGRITY    WHERE D.BUDGET < $25M OR
             D.DEPT = E.DEPT AND E.SALARY > 10000
```

In DBTG the sophistication of the protection and integrity schemes is much cruder and user dependent. Basically, user written procedure calls in the access paths to data are proposed which will allow or disallow access to a record. Except for range checks on data items, data base procedures must also be written to check for the integrity of incoming data. Although such "data base procedures" can be complex, it is non-trivial to support the above examples which the relational algorithms in [49,51] handle with ease.

XI. CONCLUSIONS

The preceding sections have attempted to justify the entries of Table 2, usually in the context of the DBTG proposal and INGRES. However, the discussion in Sections 7-10 hinges primarily on the difference in LEVEL between the DBTG proposal and INGRES and only minorly on the differences in the model of data employed. Therefore, we suggest that the relevant information to glean from Table 2 is that a non-procedural system can result in increased programmer efficiency, increased data independence and better protection and integrity at the expense of machine efficiency, compared to a procedural system. Is this price worth paying for the features of such non-procedural systems?

We suspect the answer is "not always." Given that hardware prices are continuing to decline (and will continue to do so) and that software costs are not, one suspects an increasing segment of the data base community will be willing to pay this price as time goes by. However, there will, no doubt, always be users whose interaction rates are so high, whose types of interactions are limited and whose data structures change slowly enough that they will rationally prefer a procedural system. Moreover, there may even be users whose interaction rates are still higher who will rationally prefer assembly language as a data sublanguage. To such a user 10% in efficiency may be the difference between 5 large computer systems and 6.

As a result we see a gamut of rational solutions to data base management problems including non-procedural systems, procedural systems and no special system at all. However, we suspect over the course of time the number of users who choose a non-procedural system will increase substantially.

ACKNOWLEDGEMENT

Research sponsored by the National Science Foundation Grant GK-43024x; U.S. Army Research Office -- Durham Contract DAHC04-74-G0087, the Naval Electronic Systems Command Contract N00039-75-C-0034, a Grant from the Sloan Foundation and an RCA David Sarnoff Fellowship.

REFERENCES

[1] Codd, E. and Date, C., "Interactive Support for Non-programmers: The Relational and Network Approaches," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[2] Date, C. and Codd, E., "The Relational and Network Approaches: Comparison of the Application Programming Interfaces," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[3] Whitney, V., "Relational Data Management Implementation Techniques," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[4] Bachman, C., "The Data Set View vs The Relational View," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[5] Sibley, E., "On the Equivalence of Data Base Systems," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[6] Lucking, J., "Data Base Languages, in particular DDL Development at CODASYL," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[7] Held, G., et al., "INGRES - A Relational Data Base System," Proc. 1975 National Computer Conference, Anaheim, Ca., May 1975. (to appear)

[8] Chamberlin, D. and Boyce, R., "SEQUEL: A Structured English Query Language," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[9] Codd, E., "A Data Base Sublanguage Founded on the Relational Calculus," Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, Ca., Nov. 1971.

[10] Boyce, R., et al., "Specifying Queries as Relational Expressions: SQUARE," Proc. ACM SIGPLAN-SIGIR Interface Meeting, Gaithersberg, Md., Nov. 1973.

[11] Lorie, R., "XRM - An Extended (n-ary) Relational Memory," IBM Cambridge Scientific Center Report 320-2096, Jan. 1974.

[12] ISAM, "OS ISAM Logic," IBM, White Plains, N.Y., GY28-6618.

[13] SYSTEM 2000, "SYSTEM 2000 General Information Manual," MRI Systems Corp., Austin, Texas, 1972.

[14] Marill, T. and Stern, D., "The Data Computer: A Network Data Utility," Proc. 1975 National Computer Conference, Anaheim, Ca., May 1975. (to appear)

[15] Heindel, L. and Roberto, J., "The Off-The-Shelf-System - A Packaged Information Management System," Bell Systems Technical Journal 52, 10 (December 1973).

[16] Fehder, P., "HQL: A Set-Oriented Transaction Language for Hierarchically-Structured Data Bases," Proc. 1974 ACM National Conference, San Diego, Ca., Nov. 1974.

[17] IMS/360 Applications Description Manual, IBM, White Plains, N.Y., GH-20-0765.

[18] ADABAS ADASCRIPT User's Manual, Software AG, Reston, Va., 1974.

[19] Committee on Data Systems Languages, "CODASYL Data Base Task Group Report," ACM, New York, 1971.

[20] Committee on Data Systems Languages, "Data Description Language," U.S. Dept. of Commerce, National Bureau of Standards, Handbook #112, January 1974.

[21] CODASYL Data Base Language Task Group, "COBOL Data Base Facility Proposal," Specifications Board, Dept. of Supply Services, Ottawa, Ontario, Canada.

[22] IDMS Cobol Data Manipulation Language, B.F. Goodrich, Cleveland, Ohio, July 1972.

[23] Bachman, C., "The Programmer as Navigator," CACM, 16 11 (November 1973).

[24] Falkoff, A. and Iverson, K., "APL/360 Users Manual," IBM, White Plains, N.Y.

[25] Codd, E., "A Relational Model of Data for Large Shared Data Banks," CACM, 13 6 (June 1970).

[26] Codd, E., "Normalized Data Base Structure: A Brief Tutorial," Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, Ca., Nov. 1971.

[27] Codd. E., "Further Normalization of the Relational Data Base Model," Courant Computer Science Symposium, New York, May 1971.

[28] Codd, E., "Relational Completeness of Data Base Sublanguages," Courant Computer Science Symposium, May 1971.

[29] Delobel, C. and Casey, R., "Decomposition of a Data Base and the Theory of Boolean Switching Functions," IBM Journal of Research and Development, September 1973.

[30] Naur, P., "Report on the Algorithmic Language ALGOL 60," CACM 3, 5(May 1960), 299-314.

[31] van Wijngaarden, A., et al., "Report on the Algorithmic Language ALGOL 68," Numer. Math., Vol. 14 (1969), 79-218.

[32] Wegbreit, B., "The Treatment of Data Types in EL1," CACM 17, 5 (May 1974), 251-264.

[33] Earley, J., "High Level Operations in Automatic Programming," University of California, Computer Science Department, Technical Report No. 22, October 1973.

[34] Taylor, R, "When are Pointer Arrays Better than Chains," Proc. 1974 ACM National Conference, San Diego, Ca., Nov. 1974.

[35] Morris, R., "Scatter Storage Techniques," CACM, 11 (1968).

[36] Lum, V., "General Performance Analysis of Key-to-Address Transformation Methods Using an Abstract File Concept," CACM, 16 (1973) 603-612.

[37] Held, G. and Stonebraker, M., "Storage Structures and Access Methods in the Relational Data Base Management System, INGRES," Proc. ACM-PACIFIC-75, San Francisco, Ca., April 1975.

[38] Stonebraker, M., "The Choice of Partial Inversions and Combined Indices," Journal of Computer and Information Science, June 1974.

[39] King, W., "On the Selection of Indices for a File," IBM Research Report RJ 1341, San Jose, Ca., January 1974.

[40] Schkolnick, M., "Secondary Index Optimization," Proc. 1975 ACM-SIGMOD Workshop on Management of Data, San Jose, Ca., May 1974 (to appear).

[41] Tschritzis, D., "A Network Framework for Relation Implementation," Computer Science Dept., University of Toronto.

[42] Kay, M., "An Assessment of the CODASYL DDL for Use with a Relational Subschema," Proc. IFIP-TC-2 Special Working Conference, Namur, Belgium, January 1975.

[43] Gottlieb, D., et al., "A Classification of Compression Methods and Their Usefulness in a Large Data Processing Center," Proc. 1975 National Computer Conference, Anaheim, Ca., May 1975. (to appear)

[44] Date, C. and Hopewell, P., "File Definition and Logical Data Independence," Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, Ca. Nov. 1971.

[45] Date, C. and Hopewell, P., "Storage Structure and Physical Data Independence," Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, Ca., Nov. 1971.

[46] Stonebraker, M., "A Functional View of Data Independence," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[47] Chamberlin, D. and Boyce, R., "Using a Structured English Query Language as a Data Definition Facility," IBM Research Report RJ 1318, San Jose, Ca., Dec. 1973.

[48] Codd, E., "Recent Investigations in Relational Data Base Systems," Information Processing '74, North Holland, 1974.

[49] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Workshop on Management of Data, San Jose, Ca., May 1975. (to appear)

[50] Chamberlin, D., et al., "Views, Authorization and Locking in a Relational Data Base System," Proc. 1975 National Computer Conference, Anaheim, Ca., May 1975. (to appear)

[51] Stonebraker, M. and Wong, E., "Access Control in a Relational Data Base Management System by Query Modification," Proc. 1974 ACM National Conference, San Diego, Ca., Nov. 1974.

[52] Summers, R., et al., "A Programming Language Approach to Secure Data Base Access," IBM Los Angeles Scientific Center, G320-2662, May 1974.