Copyright © 1975, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.
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
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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SQUARE[10]</td>
<td>HQL[16]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALM[9]</td>
<td>IMS[17]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SQUARE[10]</td>
<td>SQL[16]</td>
<td></td>
</tr>
</tbody>
</table>

COMPARISON OF TWO PREVAILING LEVELS OF LANGUAGE FOR DATA BASE APPLICATIONS

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

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.

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" 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 inter-
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.

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.

Example 4.2

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 actually implemented need not be
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**

| VERS [31] |
| ELI [32] |
| ALGOL64 [31] |
| ALGOL60 [30] |
| FORTRAN |
| ASSEMBLY CODE |
| MACHINER CODE |

**Figure 5**

| RELATIONS |
| CODASYL |
| DBTG |
| LOGICAL I/O |
| MACHINER CODE |
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, CHILDMAME, 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 database 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

(SET 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 entirely depends 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.

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 entirely depends 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 OFFICES 12 since this relation can be expressed as:

\[
\text{RANGE OF } E \text{ IS EMP', PROTECT E.SALARY WHERE } E.\text{OFFICE} = 12
\]

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:

\[
\text{RANGE OF } E \text{ IS EMP', RANGE OF D IS DEPT, INTEGRITY WHERE D.BUDGET < $25M OR D.DEPT = E.DEPT AND E.SALARY > 10000}
\]

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


[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 Struc-
tures 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

[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] Tsichritzis, D., "A Network Framework for Rela-
tion Implementation," Computer Science Dept.,
University of Toronto.

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

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,

[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 Work-
shop on Data Description, Access and Control, Ann

[47] Chamberlin, D. and Boyce, R., "Using a Struc-
tured English Query Language as a Data
Definition Facility," IBM Research Report

[48] Codd, E., "Recent Investigations in Relational

[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)

and Locking in a Relational Data Base System,"
Proc. 1975 National Computer Conference,
Anaheim, Ca., May 1975. (to appear)

in a Relational Data Base Management System by

Approach to Secure Data Base Access," IBM
Los Angeles Scientific Center, G320-2662,
May 1974.