THE RELATIONAL AND NETWORK APPROACHES: COMPARISON OF THE APPLICATION PROGRAMMING INTERFACES

C. J. Date
IBM United Kingdom Laboratories
Hursley, England

E. F. Codd
IBM Research Laboratory
San Jose, California

*Due to the absence of C.J. Date, this paper was presented by D. Tsichritzis.

THE RELATIONAL AND NETWORK APPROACHES: COMPARISON OF THE APPLICATION PROGRAMMING INTERFACES

by

C. J. Date
IBM United Kingdom Laboratories
Hursley, England

E. F. Codd
IBM Research Laboratory
San Jose, California

ABSTRACT: For some time now there has been considerable debate in the field of database systems over the fundamental question of the underlying design philosophy of such a system. The controversy has centered on the structure of the programmer interface, though of course the design chosen for this interface has repercussions throughout the rest of the system. Two approaches to this problem have received particular attention: the network approach, which is typified by the proposals of the CODASYL Data Base Task Group (DBTG), and the relational approach, which is advocated by the present authors (among others). The purpose of this paper is to give some comparisons between these two approaches (primarily from the application programming viewpoint), and to show what the authors believe to be the advantages of the relational approach. The reader is assumed to have a basic familiarity with the two approaches.

Computing Reviews Categories: 3.5, 3.7, 4.2

Key Words and Phrases: Data Base Management Systems
Relational Data Base Management
CODASYL Data Base Task Group

The Application Programming Interfaces

0. INTRODUCTION

It is commonly accepted that application programs which operate in a database system should not be written in terms of the data as stored but rather in terms of a logical view of the data. The programmer's interface to the database thus consists of this logical view, together with a language which enables him to manipulate this view. For some time now there has been considerable debate over the precise form this view and the associated language should take. In particular the Data Base Task Group (DBTG) of the CODASYL Programming Language Committee have proposed a network view, together with a corresponding network-handling language [2], whereas E. F. Codd and others have proposed a relational view and language [5-10,15]. It is the aim of this paper to compare these two approaches, and to show what the authors believe to be the advantages of the relational approach.

The reader is assumed to be reasonably familiar with these two approaches (see [11] for a tutorial on both). However, we start by introducing a few basic concepts and defining the appropriate terminology in each case.

The database is the data as stored. The programmer's view of the database is known as the data model. (This term is not used in [2]; the term 'database' appears to be used for both database and data model.) The data model is defined by a schema (the DBTG term) or by a data model definition (the relational term). In DBTG the schema defines 'the areas, set types, record types and associated data-items and data-aggregates' [2] that together constitute the data model. (Henceforth we shall refer to DBTG sets as owner-coupled sets where appropriate, to emphasize the fact that the construct concerned is not the same as a mathematical set - a point which frequently causes confusion.) In the relational approach the data model definition (DMD) defines the relations and underlying domains that together constitute the relational model.

The language for manipulating the data model is known as the data manipulation language (the DBTG term) or data sublanguage (the relational term.) In DBTG the data manipulation language (DML) provides facilities for operating on the constructs of the DBTG data model, and in particular for traversing the network structure (i.e., "navigating" to some desired target [20]). In the relational approach there are several candidates for consideration as a suitable data sublanguage (DSL); for simplicity we restrict ourselves to one of these only, viz. (a form of) Codd's 'DSL ALPHA' [7], which provides facilities for retrieving, creating, updating and deleting tuples of the relational model.

In practice most programmers will be interested only in a small portion of the total data model at any one time. The facility is therefore provided to extract a data submodel (DSM) from the complete data model by means of a data submodel definition (DSMD) or sub-schema (the DBTG term). A sub-schema or DSMD is essentially a subset of the corresponding schema or DMD; however, it should in general provide the ability to restructure

the data as well so that each programmer may see the data in the form most suitable for his application. We shall generally ignore the data submodel level in this paper. Figure 0.1 is a summary of these terms.

We now present a discussion of what we consider to be the relative advantages of the relational approach, under the following headings:

1. Simplicity
2. Uniformity
3. Completeness
4. Data independence
5. Integrity and security.

To be candid it is not always clear which heading a particular point should come under, but this list represents an attempt to impose some structure on the argument.

One further introductory remark: all DBTG examples etc. are based on the April 1971 proposals [2]; the revisions of [3] and [4], though defining a number of changes in syntax, do not materially affect the discussion.

1. SIMPLICITY

To a large extent simplicity may be considered the justification for the relational approach. Most of the other arguments in its favor may be viewed as aspects or corollaries of this fundamental point. However, there are some points which deserve specific mention under this heading.

1.1 The data model

First let us consider an example. Figure 1.1.1 shows an example of a relational model (suppliers-and-parts); figure 1.1.2 shows the corresponding relational DMD. Figure 1.1.3 shows the same data in the form of a network. To represent such a network using the facilities of DBTG we require two owner-coupled sets, S-SP (owned by record S) and P-SP (owned by record P), with a 'linking' record SP as member in both; this structure is illustrated in the form of a Bachman diagram [19] in figure 1.1.4. The actual DBTG data model (part only, for reasons of space) is shown in figure 1.1.5, and the corresponding DBTG schema is given in figure 1.1.6.

The comparative simplicity of both the data model itself and its definition in the relational approach should be apparent from this example. We make the following specific points.

In the relational approach the only type of structure the programmer has to understand is the table - a table, moreover, in which each column is self-identifying and in which row order is completely immaterial. In DBTG, by contrast, the programmer must understand areas (to some extent), records - which may or may not be normalized [5] - and owner-coupled sets. He must also understand the concept of ordering within such a set (and

RELATIONAL TERM	CONCEPT	NETWORK TERM (DBTG)
database	stored data	[database]
data model	(community) programmer"s view	database
data model definition [DMD]	definition of the above	schema
data submodel	(individual) programmer"s view	---
data submodel definition [DSMD]	definition of the above	sub-schema
data sublanguage [DSL]	programmer"s language	data manipulation language [DML]

Figure 0.1: comparative terminology

S

S#	SNAME	STATUS	CITY
S1	SMITH	20	LONDON
S2	JONES	10	PARIS
S3	BLAKE	30	PARIS
S4	CLARK	20	LONDON
S5	ADAMS	30	ATHENS

P

P#	PNAME	COLOR	WEIGHT
P1	NUT	RED	12
P2	BOLT	GREEN	17
P3	SCREW	BLUE	17
P4	SCREW	RED	14
P5	CAM	BLUE	12
P6	COG	RED	19

SP

S#	P#	QTY
S1	P1	3
S1	P2	2
S1	P3	4
S1	P4	2
S1	P5	1
S1	P6	1
S2	P1	3
S2	P2	4
S3	P3	4
S3	P5	2
S4	P2	2
S4	P4	3
S4	P5	4
S5	P5	5

Figure 1.1.1: the suppliers-and-parts data model (relational approach)

```
DOMAIN      S#          CHARACTER (5)
            P#          CHARACTER (6)
            QTY         NUMERIC   (5)
            SNAME       CHARACTER(20)
            STATUS      NUMERIC   (3)
            CITY        CHARACTER(15)
            PNAME       CHARACTER(20)
            COLOR       CHARACTER (6)
            WEIGHT      NUMERIC   (4)

RELATION        S       (S#,SNAME,STATUS,CITY)
                        KEY (S#)
                P       (P#,PNAME,COLOR,WEIGHT)
                        KEY (P#)
                SP      (S#,P#,QTY)
                        KEY (S#,P#)
```

Figure 1.1.2: the suppliers-and-parts DMD (relational approach)

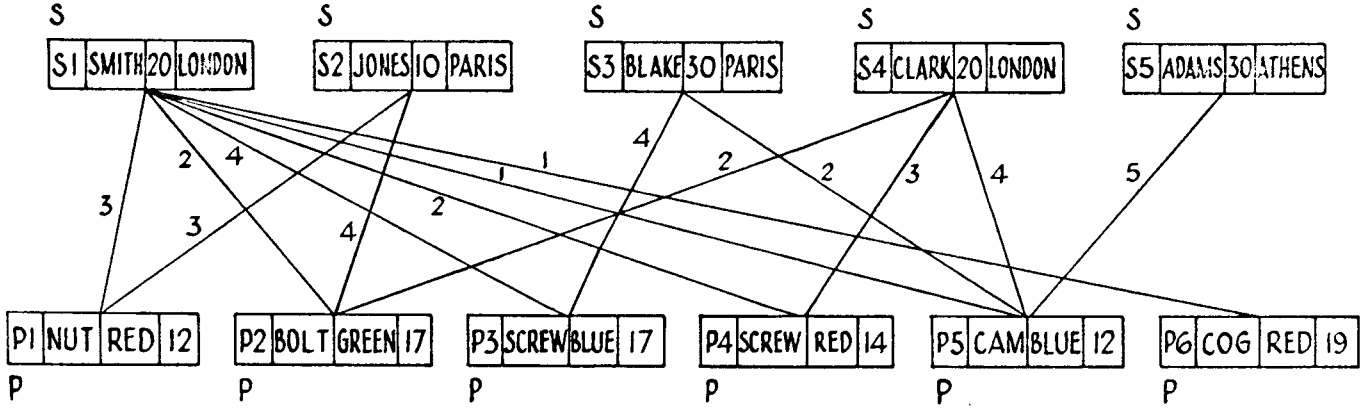

Figure 1.1.3: the suppliers-and-parts data model (network approach)

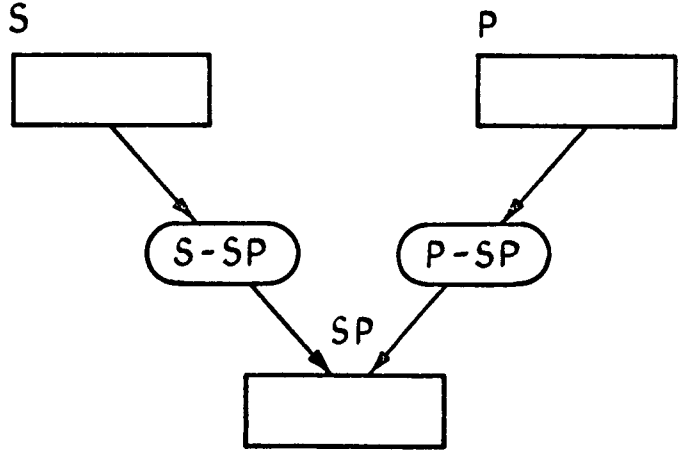

Figure 1.1.4: suppliers-and-parts data structure diagram

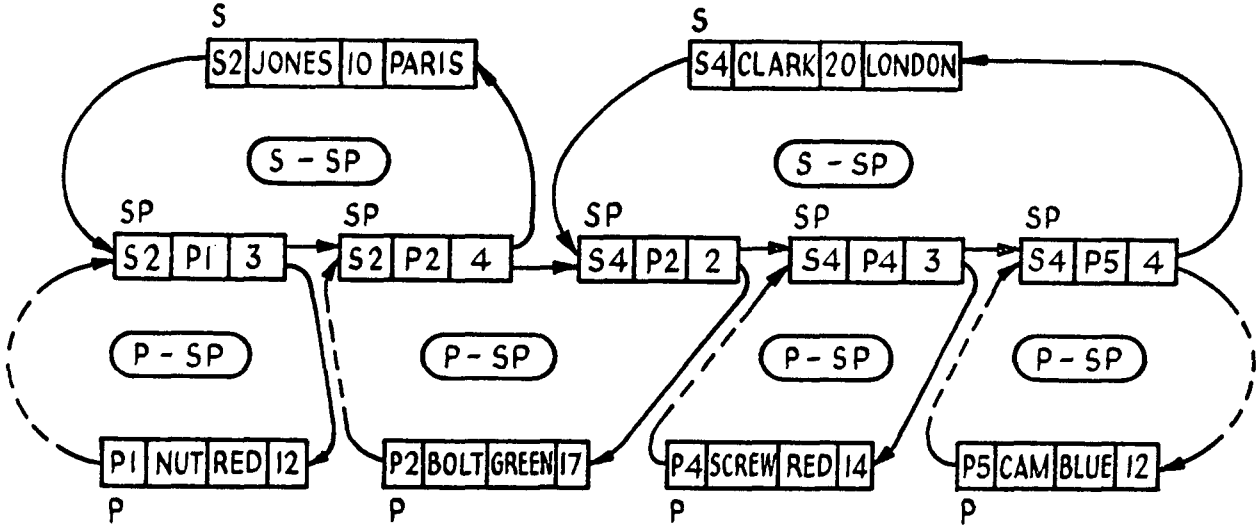

Figure 1.1.5: the suppliers-and-parts data model (DBTG part only)

```
1    SCHEMA NAME IS SUPPLIERS-AND-PARTS.
2
3    AREA NAME IS BASIC-DATA-AREA.
4    AREA NAME IS LINK-DATA-AREA.
5
6    RECORD NAME IS S;
7        LOCATION MODE IS CALC HASH-SNO USING SNO IN S
8                         DUPLICATES ARE NOT ALLOWED;
9        WITHIN BASIC-DATA-AREA.
10       02 SNO    ; TYPE IS CHARACTER 5.
11       02 SNAME  ; TYPE IS CHARACTER 20.
12       02 STATUS ; TYPE IS FIXED DECIMAL 3.
13       02 CITY   ; TYPE IS CHARACTER 15.
14
15   RECORD NAME IS P;
16       LOCATION MODE IS CALC HASH-PNO USING PNO IN P
17                        DUPLICATES ARE NOT ALLOWED;
18       WITHIN BASIC-DATA-AREA.
19       02 PNO    ; TYPE IS CHARACTER 6.
20       02 PNAME  ; TYPE IS CHARACTER 20.
21       02 COLOR  ; TYPE IS CHARACTER 6.
22       02 WEIGHT ; TYPE IS FIXED DECIMAL 4.
23
24   RECORD NAME IS SP;
25       WITHIN LINK-DATA-AREA.
26       02 SNO    ; TYPE IS CHARACTER 5.
27       02 PNO    ; TYPE IS CHARACTER 6.
28       02 QTY    ; TYPE IS FIXED DECIMAL 5.
29
30   SET NAME IS S-SP;
31       MODE IS CHAIN;
32       ORDER IS SORTED;
33       OWNER IS S.
34       MEMBER IS SP
35            OPTIONAL AUTOMATIC;
36            ASCENDING KEY IS PNO IN SP
37                 DUPLICATES ARE NOT ALLOWED;
38            SET OCCURRENCE SELECTION IS THRU
39                 LOCATION MODE OF OWNER.
40
41   SET NAME IS P-SP;
42       MODE IS CHAIN;
43       ORDER IS SORTED;
44       OWNER IS P.
45       MEMBER IS SP
46            OPTIONAL AUTOMATIC;
47            ASCENDING KEY IS SNO IN SP
48                 DUPLICATES ARE NOT ALLOWED;
49            SET OCCURRENCE SELECTION IS THRU
50                 LOCATION MODE OF OWNER.
```

Figure 1.1.6: the suppliers-and-parts schema (DBTG)

there are several different types of ordering); he may have to employ several such sets, all defined over the same records, simply to obtain several distinct orderings (a confusion between the mathematical notion of a set as a collection of objects and the concept of an access path to data). He also has to be aware of numerous rules and restrictions concerning owner-coupled sets, such as the meaning of OPTIONAL, AUTOMATIC and so on, and the fact that the same type of record cannot be both owner and member of the same type of owner-coupled set.

It may be argued, in fact, that the relational model is a representation of the data in terms of its natural structure only - it contains absolutely no consideration of storage/access details (pointers, physical ordering, indexing or similar access techniques,...); in a word, no 'representation clutter'. In the network approach, inasmuch as some sets are defined purely to provide particular orderings (and because of the nature of the DML), the programmer does have to be aware of some storage/access details. In DBTG in particular he also has to know the significance of LOCATION MODE IS CALC (an access technique which very definitely affects the way the program must be coded); he has to be familiar with two distinct methods of representing a hierarchical relationship (either as an owner-coupled set or as an un-normalized record); and he is involved with some aspects of areas and of database-keys.

The relational model is a particularly suitable structure for the truly casual user (i.e., a non-technical person who merely wishes to interrogate the database, for example a housewife who wants to make enquiries about this week's best buys at the supermarket). In the not too distant future the majority of computer users will probably be at this level. The system must therefore be capable of supporting such a view at the casual user level - and this in itself is a strong argument for providing the same view at the programmer level, since any operation at the casual user level must be implementable at the programmer level too.

One other point which may be made with respect to the simplicity of the relational model is its closeness to traditional 'flat files' - i.e., to the enormous number of files which currently exist and are organized sequentially, especially on tape. The incorporation of such files into a relational database system should thus be a comparatively painless process. (Note: if sequence in such a file is genuinely information-bearing - instead of being merely an access convenience - it may be necessary to add a sequence number to each record before converting the file into a relation.) Contrast the difficulty of converting a collection of sequential files into an equivalent network structure - probably a much more disruptive process.

1.2 The data sublanguage

Again we first consider some examples. For reasons of space we restrict ourselves to examples of retrieval only. Figures 1.2.1 through 1.2.4 show four sample queries against the suppliers-and-parts data model

relational DSL	network DML (DBTG)
GET INTO W (SP.P#) WHERE (SP.S#="S4")	MOVE "S4" TO SNO IN S. FIND S RECORD. IF S-SP SET EMPTY GO TO NONE-SUPPLIED. NXT. FIND NEXT SP RECORD OF S-SP SET. IF ERROR-STATUS=0307 GO TO ALL-FOUND. GET SP. (add PNO IN SP to result list) GO TO NXT.
answer P# P2 P4 P5	

Query is of form S#="S4", P#=?, QTY=don"t care.

Figure 1.2.1: find part numbers for parts supplied by supplier S4

relational DSL	network DML (DBTG)
GET INTO W (SP.QTY) WHERE (SP.S#="S4" & SP.P#="P5")	MOVE "S4" TO SNO IN S. FIND S RECORD. MOVE "P5" TO PNO IN SP. FIND SP VIA CURRENT OF S-SP USING PNO IN SP. IF ERROR-STATUS=0326 GO TO NOT-SUPPLIED. GET SP. (extract QTY IN SP)
answer QTY 4	

Query is of form S#="S4", P#="P5", QTY=?

Figure 1.2.2: find the quantity of part P5 supplied by supplier S4.

<table>
<tr><th>relational DSL</th><th>network DML (DBTG)</th></tr>
<tr><td>GET INTO W (SP.P#)
WHERE (SP.P#="S1" & SP.QTY=1)</td><td rowspan="2">MOVE "S1" TO SNO IN S.
FIND S RECORD.
MOVE 1 TO QTY IN SP.
FIND SP VIA CURRENT OF S-SP
USING QTY IN SP.
NXT. IF ERROR-STATUS=0326 GO TO EXIT.
GET SP.
(add PNO IN SP to result list)
FIND NEXT DUPLICATE WITHIN S-SP
USING QTY IN SP.
GO TO NXT.</td></tr>
<tr><td>answer
<table><tr><th>P#</th></tr><tr><td>P5
P6</td></tr></table></td></tr>
</table>

Query is of form S#="S1", P#=?, QTY=1

Figure 1.2.3: find part numbers for all parts supplied by supplier S1 in quantity one

relational DSL

```
GET INTO W (P.PNAME,S.CITY)
     WHERE ∃ SP(SP.S#=S.S#
               & SP.P#=P.P#)
```

answer

PNAME	CITY
NUT	LONDON
NUT	PARIS
BOLT	LONDON
⋮	⋮

(note elimination of duplicates)

network DML (DBTG)

```
        MOVE 0 TO NTH.
NXTP.   ADD 1 TO NTH.
        FIND NTH P RECORD OF P-SET SET.
        IF ERROR-STATUS=0307 GO TO EXIT.
        IF P-SP SET EMPTY GO TO NXTP.
        GET P.
NXTSP.  FIND NEXT SP RECORD OF P-SP SET.
        IF ERROR-STATUS=0307 GO TO NXTP.
        FIND OWNER RECORD OF S-SP SET.
        GET S.
        (insert PNAME, CITY pair into ordered
             result list unless already present)
        GO TO NXTSP.
```

Note singular set P-SET necessary in DBTG solution.

Figure 1.2.4: find all valid part-name/supplier-city pairs

as they might appear (a) in the relational language DSL ALPHA, (b) in the DML of DBTG. Notice in the fourth example (figure 1.2.4) that it is necessary to introduce a singular system-owned set P-SET linking all P occurrences in order that we may process all P occurrences one by one. Similar considerations may cause us to introduce a singular set for each type of record in the data model, in the extreme case. The comparative simplicity of the relational approach is immediately apparent.

The relational DSL is close to natural language. This is particularly true for retrieval; a case can also be made out for the storage operations - creating, updating and deleting - although of course these are not illustrated in figures 1.2.1 through 1.2.4. Apart from the advantage to the casual user, this is a good argument for giving the programmer such a language as well: it may be viewed as another step in the general historical trend towards programming languages of ever higher level. All the usual arguments apply: reduction in the number of decisions the programmer has to make, reduced scope for errors, reduced maintenance, increased productivity and so on.

It can be claimed, in fact, that generally speaking the complexity of any given statement in DSL ALPHA is in direct proportion to the complexity of the operation it represents. It is certainly true that 'simple' operations are genuinely simple to perform: see figure 1.1.1, for example. This is in sharp contrast to the situation in DBTG, where complexity is often introduced by the nature of the schema and DML even when it is not intrinsic to the operation to be performed.

The relational DSL is highly non-procedural. This has the effect of removing the burden of arbitrary and error-prone decisions from the programmer's shoulders. The nature of networks, however, forces programs to be extremely procedural. The DBTG notion of currency in particular has many consequences on programs in this area. For example, to INSERT a record occurrence R into an owner-coupled set occurrence S, the programmer must (a) find the set occurrence S, then (b) find the record occurrence R, and then (c) issue an INSERT to link the latter into the former. The point here is that, although steps (a) and (b) are quite unrelated to each other, they must be performed in this sequence (because INSERT operates on the current of run-unit).

Lastly, the relational DSL operates in terms of sets (mathematical sets, not the DBTG construct); that is, the programmer simply states a definition of the set - actually the relation - he wants to access, and leaves all details of how the accessing is done to the system. This has two significant corollaries. First, optimization is possible: since the access strategy is not embedded in the program, the system has a chance to make dynamic decisions as to the best way of actually performing the operation, based upon parameters such as the current storage structure and current distributions of data values. Some interesting work has been done on this problem by Palermo [17] and by Rothnie [21]. In the network approach, of course, many of the details of access strategy are firmly embedded in the logic of the application program.

The second corollary of the orientation towards set handling in the relational DSL is that it will lead to more effective use of communications lines if the database is distributed. Basically this is because the system receives one request for each (mathematical) set of records required, rather than one for each individual record. Again this is not the case in the network approach, where programs operate entirely in a 'one-record-at-a-time' mode.

Further specific criticisms may be made of the DBTG DML under the heading of simplicity (or rather lack of it): in particular the concept of currency, and the associated notion of selective currency suppression, contribute greatly to the complexity of the DML. There are of course no comparable concepts in DSL ALPHA. Another complicating factor is the number of different FIND statements which seem to be required.

2. UNIFORMITY

2.1 The data model

The relational model provides a uniform view of data, in the sense that no distinction is made between 'entities' (such as suppliers or parts) and 'relationships' (such as supplier-part links): both are represented in the same way, viz. as tuples. In other words there is one and only one way of representing an entity (as an entity-identifier - the primary key value - together with the associated attributes) in the relational model, on the understanding that a 'relationship' is merely a particular type of entity. A corollary of this is that a significantly smaller set of operations is required in the corresponding DSL.

The network model, by contrast, does make a distinction between 'entities' and 'relationships'. Roughly speaking 'entities' are represented as records in the traditional way, whereas 'relationships' are represented by (owner-coupled) set membership and by (owner-coupled) set ordering. At first sight this distinction may appear fairly natural and indeed attractive - but in fact it is the source of additional complexity, both in the model and the language, and may also lead to disruptive growth. We shall return to this point later.

2.2 The data sublanguage

We have already mentioned that in the relational DSL a very small set of operations is sufficient. This is certainly not the case with the network approach. Consider what is involved in DBTG, for example, in retrieving information about an 'entity' - say supplier S4 - as opposed to retrieving information about a 'relationship' - say that between supplier S4 and part P5. (See figure 1.2.2. A GET after the second statement would suffice for the first of these retrievals; the entire procedure is required for the second.) Consider too the difference between creating an 'entity' (which involves a STORE) and creating a 'relationship' (which requires a STORE and an INSERT, at least conceptually).

The uniformity of the relational model permits 'symmetric exploitation' in the relational DSL. This means that we may access a relation by specifying values for any combination of its domains (and the form of the access statement is essentially the same in all cases). For example, see figures 1.2.2 and 1.2.3, in both of which we are accessing relation SP with values specified for two domains and retrieving corresponding values for the third. Notice that the corresponding DBTG statements are certainly not 'symmetric'.

For practical purposes, except when the operation is extremely complicated, there is effectively one and only one way of expressing any given operation in the relational DSL. (This follows from the fact that the programmer writes a definition of the set he wishes to access. Sometimes it is possible to write such a definition in a number of different ways, but the differences are only superficial. Figure 2.2.1 shows three superficially distinct DSL ALPHA GET statements which are in fact equivalent to each other.) Once again, then, the effect is to reduce the number of decisions which have to be made by the programmer.

```
GET INTO W (S.SNAME) WHERE

(a)    ∃SP(SP.S#=S.S# &  ∃P(P.P#=SP.P# & P.COLOR='RED'))

(b)    ∃P(P.COLOR='RED' &  ∃SP(SP.P#=P.P# & SP.S#=S.S#))

(c)    ∃P ∃SP(P.COLOR='RED' & SP.P#=P.P# & SP.S#=S.S#)
```

Figure 2.2.1: find names of suppliers who supply at least one red part

In the network approach, of course, there are many quite distinct ways of performing any given operation. As a simple example consider figure 1.2.2; the procedure here starts at the supplier (S4), but it could equally well start at the part (P5). Once again this puts an added burden on the programmer. Incidentally these two alternative procedures, though both eventually accessing the same SP occurrence, have different side effects, inasmuch as different currency indicators are set in the two cases - this represents an additional and very serious problem. To reach any specific target data the network normally requires the programmer to choose between many paths, and sometimes between different types of path. In making this choice, he must also try to anticipate how the various currency indicators will be affected.

This is probably a good point at which to note that not all relational DSLs need involve the 'strange symbolism' of predicate calculus. SEQUEL [22] is an example of one which does not. The query of figure 2.2.1 would appear in SEQUEL as follows:

```
SELECT SNAME FROM S
       WHERE S# =
             SELECT S# FROM SP
             WHERE P# =
                   SELECT P# FROM P
                   WHERE COLOR='RED'
```

Even in DSL ALPHA it would clearly be possible to introduce a default mechanism so that the existential quantifier need never be written explicitly.

3. COMPLETENESS

3.1 The data model

The relational model is complete in the sense that, as shown in [6], all data structures commonly employed in database systems can easily be cast into relational form. Hence relations are at least adequate - there are no restrictions on what can be handled.

Moreover the design discipline of third normal form [8,16,18] allows the data model to be non-redundant in the following (somewhat intuitive) sense: each fact is represented once and once only (i.e., in precisely one place). 'Fact' here refers to the association between an entity and one of its attributes, e.g., the 'fact' that supplier S4 is located in London. Hence relations are not only adequate, they are also not too rich. These remarks should not be taken to mean that all possible redundancy is automatically eliminated. In some circumstances, unnormalized records may be considered less redundant than a normalized relation, for example.

Thus relations provide the ability to construct an accurate model of the real world, which consists of a number of entities (of various types), together with their attributes, and nothing else. Notice, however, that we require the concept of primary key to build such a model (a concept which is absent from DBTG). It may indeed be argued that a given third normal form relational model is a canonical representation of the data concerned, in that it contains all the intrinsic properties of the data and nothing besides.

Turning now to the network model, it too is complete in the above sense. However, it is no more complete, despite its far greater complexity (i.e., there is nothing which can be represented in network form and not in relational form). Moreover networks usually do contain redundant information (with the aim of simplifying retrieval, at the expense of complicating maintenance): for example, the 'fact' that supplier S4 supplies part P5 is represented twice in the network of figure 1.1.5, once by the appearance of 'S4' within a particular occurrence of record SP and once by the appearance of this SP within a particular occurrence of owner-coupled set S-SP.

Moreover, despite the fact that networks do permit completeness of representation, it is usually the case that the network does not contain a direct representation of everything (specifically, of every relationship), simply because it would be extremely complex if it did. As an example, suppose that in the suppliers-and-parts data model (figure 1.1.1) we have an additional piece of information for each part, viz. CITY, representing the city where the part is stored. (For the purposes of the example we assume that each part is stored in one and only one city.) In the relational model this corresponds to adding a new domain CITY to relation P: for example, see figure 3.1.1.

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	NUT	RED	12	LONDON
P2	BOLT	GREEN	17	OSLO
P3	SCREW	BLUE	17	PARIS
P4	SCREW	RED	14	NICE
P5	CAM	BLUE	12	PARIS
P6	COG	RED	19	PARIS

Figure 3.1.1: relation P extended to include domain CITY

An analogous change is made to the network model, viz. a new data-item CITY is added to record P. Now consider the query 'find all suppliers and parts that are co-located, i.e., have the same value for CITY'. See figure 3.1.2.

relational DSL

```
GET INTO  W (S.S#,P.P#)
    WHERE (S.CITY=P.CITY)
```

answer

S#	P#
S1	P1
S2	P3
S2	P5
S2	P6
S3	P3
S3	P5
S3	P6

network DML (DBTG)

```
      MOVE 0 TO NTH.
NXTS. ADD 1 TO NTH.
      FIND NTH RECORD OF S-SET SET.
      IF ERROR-STATUS=0307 GO TO EXIT.
      GET S.
      MOVE CITY IN S TO CITY IN P.
      FIND P VIA CURRENT OF P-SET
                  USING CITY IN P.
NXTP. IF ERROR-STATUS=0326 GO TO NXTS.
      GET P.
      (add SNO, PNO pair to result list)
      FIND NEXT DUPLICATE WITHIN P-SET
                  USING CITY IN P.
      GO TO NXTP.
```

Figure 3.1.2: find all suppliers and parts that are co-located

Notice, incidentally, that in the network case we now need a singular et of suppliers (S-SET) as well as one of parts (P-SET). The point about his example is that the network chosen does not directly represent the elationship which connects suppliers and parts with the same CITY value. By contrast it does directly represent the relationship connecting S's nd SP's with the same supplier number, also the relationship connecting 's and SP's with the same part number.) Of course it is possible to epresent this relationship 'directly', by introducing appropriate wner-coupled sets. However, the authors contend that it is unreasonable o represent all such relationships directly, because of the tremendous omplexity this would introduce into the model. Thus the programmer has o be aware of which relationships are represented directly and which not, ince it very definitely affects the code he has to write. (Note too that e have only considered cases in which the basic comparison operator efining the relationship is equality. The problem is greatly magnified f we consider other operators as well.) Another corollary is that the atabase designer has to be able to anticipate the various uses to which he data is to be put in order to provide the appropriate structure.

In the relational approach, of course, the relationship is 'directly' upported, in the sense that it is specified via a predicate in the GET tatement. (In other words, the connection between suppliers and parts ith the same CITY value is expressed in exactly the same sort of way as he connection between S's and SP's with the same supplier number.)

3.2 The data sublanguage

DSL ALPHA is based on predicate calculus. We therefore know that ny relation definable in terms of the data model by means of a predicate n the relational calculus can be retrieved in one GET statement. In the etwork approach, of course, for all but the very simplest cases it is ecessary to write a procedure. For examples see figures 1.2.1 through

1.2.4 and figure 3.1.2. Once again this is highly significant from the point of view of programmer productivity.

It is not really suggested that the programmer actually express very complex queries in the form of a single GET; of course it is always possible to break such queries down into a sequence of simpler ones. (Nevertheless, the power of the language is such that even the 'simple' queries may actually be quite complex.) More important, however, the relational completeness of DSL ALPHA makes it an extremely suitable candidate as a target for higher-level language translators. (By higher-level language here we mean a language suitable for casual users, not a functionally equivalent alternative to DSL ALPHA such as SEQUEL [22].) Since all the basic retrieval power any such language may need is embodied in DSL ALPHA, we can implement this basic retrieval capability once instead of many times. If instead a network language is used as such a target language, there will be much duplication of function (e.g., in generation of access strategies) in the corresponding higher-level language translators.

4. DATA INDEPENDENCE

It is generally agreed that data independence should be a goal of database systems. However, different people have different ideas as to what this expression means. We mean:

(a) program immunity to change in the storage structure (sometimes referred to as physical data independence) [12]; also

(b) program immunity to growth in the data model definition (sometimes referred to as logical data independence) [13].

We want storage structure independence to allow the storage structure to be tuned to optimize overall performance, to implement new standards in the storage structure, to take advantage of new hardware technology as it is developed, and for many other reasons. We want growth independence to allow the addition of new types of entity and/or the addition of new facts about existing types of entity; in particular this is essential to the gradual introduction of a database system into an organization.

4.1 Storage structure independence

As far as the relational approach is concerned, it is obvious that neither the model nor the language contain any reference to storage constructs of any type, nor to any corresponding access techniques; they are both solely concerned with the information in the database. Specifically, there is absolutely no mention of the stored form of relations and domains or to pointer chains, indexes, hashing algorithms or any other access mechanism. Hence the implementation is free to choose any one of a very large class of possible storage structures and corresponding access techniques, as is shown in [12]. As an example,

MACAIMS [15] actually provides an open-ended set of different storage structures all within the same system; each structure has an associated 'relational strategy module' whose function is to manage that structure and to make it look like the 'pure' (tabular) structure to the rest of the system. Incidentally the lack of fact redundancy in the relational model does not necessarily force 'zero redundancy' in the storage structure; the implementation may certainly introduce redundancy at this level if this is required, say for performance reasons.

By way of contrast, it is the very essence of the network approach that the programmer is involved in tracing access paths (i.e., access strategy is embedded in the application program). It is usual to talk in terms of 'pointers' and 'chains', though as Bachman [1] has indicated these 'chains' need not be implemented as actual stored chains of pointers. However, the programmer may always think of these 'chains' as physically existing, even if they are represented by some alternative method, because of the chain-traversing nature of the data manipulation language. It follows from this that the implementation must provide the 'chains' the programmer sees in the data model. This very severely limits the degree of variation possible in the storage structure.

In DBTG in particular there are a number of additional considerations which constrain matters still further. These include programmer dependence on LOCATION MODE and SET OCCURRENCE SELECTION (especially LOCATION MODE IS CALC), also the visibility of areas and of database-keys. See Engles [14].

4.2 Growth independence

Growth can be very clearly defined in relational terms. Adding a new type of entity corresponds to adding a new relation to the DMD; adding a new fact about an existing type of entity consists of adding a new domain to an existing relation in the DMD. Incidentally such a new domain cannot by definition be a constituent of the relation's primary key. Neither of these two types of change will have any effect on any existing program, because no existing program will contain any explicit reference to the new domain or new relation. Inspection of the examples earlier in this paper will show this to be so for retrieval; as for the storage operations, UPDATE and DELETE operations will be unaffected, PUT operations will create tuples which are extended by the system to include null values in positions corresponding to new (unspecified) domains.

In most cases growth can also be satisfactorily accommodated in the network approach, provided that the network is designed on what may be called 'third normal form principles'. Specifically, there must exist a one-to-one correspondence between the tuples of a third normal form relational representation and the record occurrences of the network representation; and in the case of a linking relation (i.e., one with a multi-component primary key), the corresponding record must be a linking record which participates as a member in N owner-coupled sets, where N is the number of components in the primary key.

Even if these principles are adhered to, however, there is an important situation in which growth causes problems in the network approach. This is the case of adding a new fact about an existing type of entity, where the entity concerned is represented by owner-coupled set ordering. For example, suppose two adjacent member record occurrences in a given owner-coupled set represent two adjacent stations on a subway line, and it is required to introduce a new item, viz. the distance between the two stations. This item can be placed in one of the two member records - but which one? Whichever is chosen, the effect will be that the logic involved in manipulating the 'relationship' will be radically different depending on the direction of processing (the algorithm for 'travelling the line' between two stations A and B, for example, will depend on whether A precedes or follows B on the line).

The example of a data model to represent a subway network illustrates a number of points in addition to the one just mentioned. The essence of the problem is that, despite the fact that a subway network is a 'natural' candidate for a network data model, casting it into network form gives rise to several problems which are not present in the relational equivalent. For example, the 'obvious' network model will involve an owner-coupled set for each subway line, having as members the appropriate station records in the appropriate order. However this does not accurately model the case of a circular or closed-loop line: the fact that the first and last stations on the line are adjacent has to be dealt with as a special case. A suitable relational structure for this problem is a relation

R(LINE,UP,DOWN,DIST)

where the primary key is the combination LINE, UP or the combination LINE, DOWN; and UP and DOWN identify two adjacent stations on the line. Notice that this relation may be viewed as representing the line in both the UP and DOWN directions, whereas the obvious network representation is uni-directional. Moreover, this single relation can accommodate circular lines, even if it is not known at the outset that such lines might be built.

5. INTEGRITY AND SECURITY

In addition to creating the data model definition or schema the database designer is also responsible for the specification of various integrity and security constraints. By security constraints we mean, for example, checks which must be applied to ensure that the programmer is authorized to perform the operation he is attempting. By integrity constraints we mean checks which must be applied when the programmer attempts to change the database to ensure that the change is a valid one. Note that constraints of either type (security or integrity) may be arbitrarily complex. The database management system is of course responsible for implementing these constraints as specified; in addition it must include some means of handling the data sharing problem (concurrent

access), another aspect of integrity. In this section we consider the two approaches from the standpoint of integrity and security.

A major advantage of a third normal form relational model is that a clean separation is achieved between the underlying data model and any constraints one may wish to impose on top of it. In the network approach such a separation could be achieved if all constraints were specified by means of database procedures. However, it is normally the case that certain constraints are deliberately built into the data structure itself - indeed, this is one of the main reasons given for the existence of the owner-coupled set construct. Thus, to achieve the separation referred to above, the database designer is faced with the problem of disentangling the security and integrity requirements from the complexities of the data structure. In the relational approach, by contrast, the designer has only one type of structure to consider, and a very simple coordinate system (identification of relations and columns by name, and rows by content) by which he may refer to any individual item or portion of that structure.

As an example of how constraints may be independently superimposed in the case of the relational model, consider the suppliers-and-parts database, in which for consistency it would probably be required that any S# value appearing in relation SP must also exist in relation S. This integrity constraint can be simply expressed within the data model definition as follows:

$$\text{CONSTRAINT } \{SP.S\#\} \subseteq \{S.S\#\}$$

(i.e., the (mathematical) set of S# values from relation SP is a subset of the (mathematical) set of S# values from relation S). This implies checks on PUT operations for relation SP and DELETE operations for relation S.

As an example of the interweaving of constraints and structure in the network model, consider the owner-coupled set structure of figure 5.1.

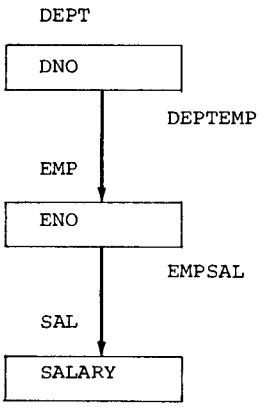

Figure 5.1: the owner-coupled sets DEPTEMP and EMPSAL

For each department we have a set of employees, for each employee a salary history, i.e., a set of salaries. With this structure how can we specify the security constraint that a particular user (a) can see EMP occurrences, (b) can see DEPT-SAL combinations, but (c) cannot see EMP-SAL combinations? The answer is that another owner-coupled set DEPTSAL (owner DEPT, member SAL) must be introduced, and the user granted access to DEPTEMP and DEPTSAL but not EMPSAL. Observe how the structure is dictated by the constraint requirement.

Another point to consider in connection with security in the network model is the following: the very fact that a particular record (occurrence) is a member of a particular owner-coupled set (occurrence) may itself be sensitive (consider the case of a set of employees in which the qualification for membership is that salary exceeds some specified value). Thus it may not be adequate to provide checks against record retrieval only - constraints may also have to be applied at the owner-coupled set level too. This is another corollary of the fact that there are several structurally distinct ways of representing information in the network approach.

Turning now to the problem of concurrent access, there are two general approaches: the locking technique, in which a program seizes exclusive control of data it wishes to prevent concurrent programs from accessing, and the warning technique, in which the system informs the program if a concurrent program has accessed data which this program has indicated it is interested in. For the relational approach some proposals have been described in [10,24]. Regardless of whether a locking or a warning technique is adopted, tuples and relations provide a very natural basis on which to define (and implement) requirements.

DBTG provides both techniques: locking may be applied at the area level, warning at the record (occurrence) level. Both these features have been criticized (see Engles [14] for example): the first because areas are not an appropriate unit for locking purposes, the second because it places too much burden on the programmer and provides far too much room for error. (This latter criticism also applies, though to a lesser extent, to a warning system as it would appear in the relational approach.) The basic problem with locking at the record occurrence level in a network is that it may then not be possible for a concurrent program to traverse the record as part of a path to some other point (and this in turn may easily lead to deadlock). If a concurrent program can traverse such a record, then that program must continually guard against the possibility that the record it is currently positioned on may be altered by another program: for example, it may be modified so that it moves out of one owner-coupled set occurrence and into another. This would have the effect of changing currency so that the concurrent program would continue along a different path from the one it was travelling before. An ERROR-STATUS code is set in this case, but the program is not forced to test for it; after all, the condition may have been impossible at the time the program was originally written.

6. SUMMARY

By way of summary we list again the major headings under which we have presented our argument:

1. Simplicity - of both the model and the language;
2. Uniformity - likewise;
3. Completeness - likewise;
4. Data independence with respect to storage structure and growth;
5. Integrity and security.

In conclusion we should mention the major disadvantage of the relational approach, which is this: at the time of writing no full-scale implementations exist (though a number of prototype systems incorporating relational ideas have been built at universities and elsewhere, and investigations are continuing within General Motors Research [25] and IBM Research [22-24], to name just two). It is thus not yet possible to state what the performance of such a system will be like, either in terms of space utilization or of response time. However, some interesting work has been done in this area, and there is good reason to be hopeful about the outcome.

ACKNOWLEDGMENT

The authors would like to thank numerous colleagues at IBM San Jose Research Laboratory, especially R. F. Boyce, D. D. Chamberlin, W. F. King and I. L. Traiger, for reading an early draft of this paper and for their many helpful suggestions.

REFERENCES

1. C. W. Bachman: "Implementation Techniques for Data Structure Sets", Proc. SHARE Workshop on Data Base Management Systems, Montreal, July 1973, SHARE Distribution.

2. CODASYL Data Base Task Group: Report, April 1971.

3. CODASYL Data Base Language Task Group: Proposal, February 1973.

4. CODASYL Data Description Language Committee: Journal of Development, June 1973.

5. E. F. Codd: "A Relational Model of Data for Large Shared Data Banks", CACM 13,6 June 1970.

6. E. F. Codd: "Normalized Data Base Structure: A Brief Tutorial", Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control.

7. E. F. Codd: "A Data Base Sublanguage Founded on the Relational Calculus", Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control.

8. E. F. Codd: "Further Normalization of the Data Base Relational Model", In "Data Base Systems", Courant Computer Science Symposia 6, Prentice-Hall 1972.

9. E. F. Codd: "Relational Completeness of Data Base Sublanguages", Courant Computer Science Symposia 6, Prentice-Hall 1972.

10. E. F. Codd: "Access Control for Relational Data Base Systems", Presented at BCS Symposium on Relational Database Concepts, London, April 1973.

11. C. J. Date: "Relational Database Systems: a Tutorial", Proc. 4th International Symposium on Computer and Information Science, Miami Beach, December 1972, Plenum, New York

12. C. J. Date, P. Hopewell: "Storage Structure and Physical Data Independence", Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control.

13. C. J. Date, P. Hopewell: "File Definition and Logical Data Independence", Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control.

14. R. W. Engles: "An Analysis of the April 1971 DBTG Report", Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control.

5. R. C. Goldstein, A. J. Strnad: "The MacAIMS Data Management System", Proc. 1970 ACM SIGFIDET Workshop on Data Description and Access.

6. I. J. Heath: "Unacceptable File Operations in a Relational Database", Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control.

7. F. P. Palermo: "A Data Base Search Problem", Proc. 4th International Symposium on Computer and Information Science, Miami Beach, December 1972, Plenum, New York.

8. P. H. Prowse: "The Relational Model as a System Analysis Tool", Presented at BCS Symposium on Relational Database Concepts, London, April 1973.

9. C. W. Bachman: "Data Structure Diagrams", Data Base 1,2, Summer 1969.

10. C. W. Bachman: "The Programmer as Navigator" (1973 Turing Award Lecture), CACM 16,11 November 1973.

11. J. B. Rothnie: "The Design of Generalized Data Management Systems", Ph.D. Dissertation, Dept. of Civil Engineering, MIT (September 1972).

12. D. D. Chamberlin, R. F. Boyce: "SEQUEL: A Structured English Query Language", Proc. ACM-SIGFIDET Workshop 1974

13. R. Boyce, D. D. Chamberlin: "Using a Structured English Query Language as a Data Definition Facility", IBM Research Report RJ 1318.

14. D. D. Chamberlin, R. F. Boyce, I. L. Traiger: "A Deadlock-Free Scheme for Resource Locking in a Database Environment", Information Processing 74, North-Holland, Amsterdam.

15. V. K. M. Whitney: "Fourth Generation Data Management Systems", Proc. 1973 National Computer Conference, New York.