Relational Data Dictionary Implementation

I A Clark, IBM United Kingdom Scientific Centre, Peterlee, UK

Abstract

The paper presupposes a team of application developers using an application generator served by a relational database (RDB). The application grows by including not only routines for input/output, but by accumulating new relations, the latter representing data-definition activity by the developers.

A data dictionary (DD) is needed

(1) to interrelate relations,

(2) to relate these to routines, input streams and reports,

(3) to produce auditing reports and clerical procedures manuals.

The benefits and technical problems of maintaining the DD itself as a RDB are treated.

INTRODUCTION

This paper assumes a development team using an application generator served by a relational database (RDB). The application grows not only by adding I/O and processing routines, but also by accumulating new relations. Such relations may be derived from already existing relations in the database, as well as being inserted independently as a set of tuples.

We do not want to argue here why we consider an application generator

together with a relational data base. Suffice to say that we believe this combination to be an attractive one for use by a team of non-data-processing professionals. By 'non-DP professionals' we shall mean a group of highly skilled individuals who wish to innovate within their own discipline by making use of the computer, without being diverted from their true purpose by considerations of a purely technical nature to do with data-processing. In particular such individuals will not wish to be deflected by questions of choosing data pathways or the best data structure for their particular purpose (hence the relational database), nor get involved with the wide choice of techniques for doing essentially standard programming operations (hence the application generator).

We shall consider a relational database similar to a research prototype developed and used at the IBM UK Scientific Centre, Peterlee, called PRTV (1). The chief feature of PRTV is that of deferred operation, that is, a new relation derived from existing relations is not materialised into a set of tuples until these tuples are explicitly called for; eg, to open the relation as a read-only file, or to find out how many tuples it contains. A new relation can be defined during a terminal session by entering an expression which contains names of existing relations, acted on by the relational operations:

UNION	PROJECTION
INTERSECTION	SELECTION
DIFFERENCE	JOIN

The result is a named entity within the user's workspace which we shall call a '(relational) value'. It is not our purpose to describe how this entity is implemented. Suffice to say that it is a character string which specifies briefly but conveniently to the routines which materialise the tuples just how to go about doing so. Within this relational value there exist, as intact substrings in fact, either the names, or the values, of the relations from which it was derived.

However, note that by the term: 'derived relation', we shall mean specifically one whose relational value contains the name of another relation, say 'A', rather than just the value of A. This is because, in PRTV, there is no way of effectively recognising that, say, B has been obtained from A in the latter case. If for instance relation A

were bulk-loaded from cards, next relation B created and simply assigned the value of A, there would be nothing inherently different about A and B. Indeed, in PRTV as it stands there would be no way of telling which came first! Moreover either A or B could be reassigned another value, leaving the other unchanged. This is clearly not the case if B were derived from A. Then whenever A changed its value, B would change correspondingly.

Since relational values are relatively small entities compared with the large sets of tuples they can potentially represent, one must not think that a computer process which forms new relations at run time out of existing relations is necessarily going to be extravagant. Thus PRTV allows one to formulate as much of one's application as one cares to in a relational algebra, which on the face of it performs set-theoretic operations upon whole sets of tuples. However, the operations are really performed on the relational values we have just described, with the result that the operation of forming the union, say, of two large sets of tuples is deferred until one actually lists a relation, or opens a sequential file based on that relation and scans the file. We are going to formulate, in a relational algebra, processes which experienced programmers would not consider handling in terms of elementary operations which combine entire sets of tuples, or as they would see them, sets of records.

Instead of a relational algebra, a relational calculus may of course be used instead, eg the ALPHA language of E F Codd (2). PRTV does not yet support ALPHA, nor any such relational calculus. However, as Codd has shown elsewhere, it is in principle feasible to translate from one to the other in a natural way. An ALPHA expression resembles a theorem in the Propositional Calculus. To a logician, this represents a natural and general way of making an assertion about a given computer process. Other professionals have their own languages within their own disciplines. Whether or not they can understand a Propositional Calculus expression does not matter: their own languages are likewise amenable to machine translation into the relational algebra.

Consider an application which accepts a batch of input and produces reports (invoices, cheques, etc). It is conceivable in principle to load the input straight into a number of relations, then print out the reports directly from relations derived from the input relations. How far one progresses towards this limit depends in practice on whether

it appears easier to implement a given step using the relational algebra, or a conventional programming language. A non-DP professional is unlikely to be predisposed towards the programming solution, particularly if provided with an application generator which constructs the relational algebra for him out of more familiar specifications.

The main problems which the application generator will have to handle are those of making the work of one team member available to another in an orderly fashion, and to stop them unsuspectingly cutting the ground away from under each others' feet.

This can so easily happen if the result of one individual's work, embodied in a relation, is passed to another, who incorporates it into a derived relation which is in turn passed on. It becomes a heavy administrative task to keep track of what changes to the original relation are safe, permissible, or are nonsense in terms of the real-world application.

Note that with this remark we do not distinguish between application development and operational running of the application.

One possible way of coping with this task is for the application generator to administer a data dictionary. Since the task involves much cross-indexing, and the application generator is already served with a relational database, it is attractive to investigate maintaining the data dictionary itself as a relational database.

A range of tasks may be undertaken by the data dictionary, from the simplest to the most ambitious. Examples are:

(1) reporting upon all relations which are affected by updating a given relation,

(2) preventing or otherwise qualifying an order to destroy a relation upon which further relations are defined,

(3) enforcing semantic constraints imposed by the nature of the application at either application development time, eg, to prevent the insertion of 'nonsense' relations into the database, or at run-time, eg, to ensure that tuples are not inserted into a given relation without corresponding tuples being present in another relation.

(4) producing listings of all routines and reports relating to a given database relation, for auditing purposes,

(5) maintaining an up-to-date clerical procedures manual. This often requires cross-referenced lists of fields on input documents, reports, and domains in the database.

These tasks are represented in order of increasing severity. We shall treat the first three only, discussing some theoretical and technical problems which the data dictionary has to face. The remaining two topics, although ambitious in practice, are theoretically much simpler than the first three.

(1) REPORTING UPON UPDATE DEPENDENCIES

For the moment we are primarily concerned with update dependencies between relations in the course of application development. The other sort of update dependency, that between records, or tuples in our case, will be treated later under the heading of 'semantic constraints'.

This facility is straightforwardly achieved by maintaining a DD-relation, call it RDEPEND, on the domains RELID1, RELID2, DEPTYPE. By 'DD-relation', we mean 'data dictionary' relation, to distinguish it from the relations belonging to the application itself. DD-relations may or may not be kept in the same database as application relations: for research convenience the former is recommended due to the facility for bootstrapping the data dictionary, the latter advisable however for security.

Note that we require some means of referring to distinct occurrences of the same domain within the component list of a relation. We do this here by postfixing 1, 2, etc, to the domain name (eg, RELID1, RELID2, etc, for the domain name RELID). RDEPEND contains a tuple for each derived relation, stating what relation it depends on (RELID2) and in what capacity (DEPTYPE). Where a relation is derived from a number of other relations, that number of tuples is present in RDEPEND. Furthermore, if the relation uses another in more than one capacity, more than one tuple for that pair of 'RELIDs' occurs.

Now comes the advantage of using a relational database for the data

dictionary. The relation RDEPEND is transitive in a logical sense. Thus by joining it to itself repeatedly we recover a relational value which carries a tuple for all the implicit dependencies, as well as those appearing explicitly in RDEPEND.

Let us introduce notation to present an example. This notation is based on the relational algebra, ISBL, used by PRTV, although we modify it freely in order to make it better illustrate our points.

In PRTV a user manipulates relations within his workspace by expressions of the form:

```
C = A * B
C = N!A * B
```

The first command would construct a relation with a RELID of 'C' (the named entity introduced earlier with its symbolic 'value'; no tuples are accessed as yet) and a value equal to the 'join' of the values of 'A' and 'B'.

The second command would incorporate the RELID: 'A' into the relational value formed for 'C' instead of the value of A. 'N!A' should be read as 'name-A'.

Suppose we have defined 'F' by the following sequence of commands:

```
C = N!A
D = N!B
E = N!C
F = N!E * D
```

Then RDEPEND would contain the following tuples:

```
RDEPEND ( RELID1  RELID2  DEPTYPE )
            C       A        N
            D       B        N
            E       C        N
            F       E        N
            F       D        V
```

In order to obtain tuples for every dependency of F one might join RDEPEND with itself repeatedly until no further tuples appeared (detected by testing its cardinality). The type of 'join' operation required is one called an 'equi-join'. This means that the tuples from each relational operand which are to be concatenated are chosen by collating equal values within certain specified domains. It is a matter of notational design to specify an equi-join elegantly. Here we show the required components to 'overlap' by placing component names beneath each other. Thus:

```
  RDEPEND|RELID1 RELID2 DEPTYPE
* RDEPEND|       RELID1        RELID2 DEPTYPE
```

represents a relational value with five domain occurrences. Each tuple in the set so defined is formed by taking a pair of tuples from RDEPEND for which RELID1 in one tuple equals RELID2 in the other. There is a combined tuple for all such pairs.

We may further join to this a relation, DTRANS, which contains a tuple matching each pair of values of DEPTYPE which turns up in the above relational value. Each tuple of DTRANS contains a third value from the domain DEPTYPE, representing the resulting (ie, transitive) dependency. After that, we can project out just those domains we wish to see, renaming them in the process. Note that in a relation, all duplicates of a given tuple are suppressed. A relation simply records that, say, three given objects are related in a given way. The ordered set of these three objects is what comprises the 'tuple' (3-tuple, or 'triple' in this case). Thus it makes no sense to talk about more than one 'occurrence' of this tuple. The three objects are either related, or they are not.

We may thus construct the relational assignment statement:

```
RR = RDEPEND|RELID1 RELID2 DEPTYPE
   * RDEPEND|       RELID1         RELID2 DEPTYPE
   * DTRANS |               DEPTYPE1       DEPTYPE2 DEPTYPE3
   %        |RELID1                RELID2           DEPTYPE
```

The resulting relation RR has precisely the domains and domain-IDs of RDEPEND (the final 'project', %, has seen to that), but relates RELIDs once-removed. Thus RR contains the following tuples only:

```
RR ( RELID1  RELID2  DEPTYPE )
     E       A       NN
     F       C       NN
     F       B       V
```

The relation DTRANS can be visualised as a function with two arguments, DEPTYPE1 and DEPTYPE2, returning the corresponding object in the domain DEPTYPE3. Indeed in PRTV it can be implemented either as a PL/I function or as an ordinary relation, with a tuple for every pair of values of DEPTYPE1 and DEPTYPE2. Thus DTRANS might contain the following tuples (among others):

```
DTRANS ( DEPTYPE1 DEPTYPE2 DEPTYPE3 )
         N        N        NN
         N        NN       NNN
         NN       N        NNN
         NN       NN       NNNN
         N        V        V
         V        N        V
```

Note that the last two tuples say, in effect, that if A depends on the

name 'B', and that B has the value of C, then A has only a current-value connection with C. If C is changed, A will not change, and therefore this connection will be lost. On the other hand, if B depends on the name 'C', assigning the (current) value of B to A effectively assigns the current value of C to A. This is a matter of choice of convention.

RR can be incorporated back into RDEPEND (eg, by the expression:)

```
RDEPEND = RDEPEND + RR
```

and the process repeated until the cardinality of RDEPEND grows no more. On the other hand it may be better to derive a new relation, FULL_RDEPEND, by this process each time it is called for, so that RDEPEND may be maintained more easily by simple insertion and deletion of tuples.

When the owner of the catalogued relation, F, wishes to modify it, the command:

```
List FULL_RDEPEND: RELID1 = 'F'
```

might be issued. This lists a selection of just those tuples in FULL_RDEPEND such that RELID1 is equal to 'F'. The relational operator ':' stands for 'SELECT'. Thus:

```
FULL_DEPEND: RELID1 = 'F'
             ( RELID1  RELID2  DEPTYPE )
               F       E       N
               F       D       V
               F       C       NN
               F       B       V
               F       A       NNN
```

(2) QUALIFYING AN ORDER TO DESTROY A RELATION

This might be considered to be a special case of enforcing semantic constraints imposed by the application model upon the developers themselves, a very general topic. However it can also be viewed as a basic facility to be expected of a system which claims to inhibit members of an application development team from cutting the ground from under each others' feet. There is a temptation to build such a facility rigidly into the system itself. This ignores the possibility that what is satisfactory for one application development team may not be so for another.

The simplest such 'qualification' is of course to refuse to destroy any relation from which another relation has been derived, ie, upon which

there is a name-dependency, until those dependencies have been eliminated.

(3) ENFORCING SEMANTIC CONSTRAINTS IMPOSED BY THE APPLICATION MODEL

To the theorist this is probably the most interesting use to which a relational data dictionary might be put.

One objection to the use of relational databases stems from the fact that certain properties of conventional files, such as demanding a unique value in the key field, or being hierarchical, are absent. In conventional programming, these 'structural' properties are exploited to enforce certain semantic constraints arising out of the application model, such as a particular child segment having a single parent. However the skills of a database specialist are often needed to exploit such restrictions inherent in the available structures. It is up to him to ensure that his model of the application in terms of key fields and segment deletion rules behaves like the real-world counterpart: yet it is often rather hard for a business to find a man with intimate knowledge of both realms. Thus it is attractive for our purpose that the traditional restrictions of key-fields and many-one mappings have to be modelled explicitly in PRTV, since the problem of enforcing semantic constraints can then be split off from that of providing a structure capable of holding the data in the first place.

How can one use the relational algebra here discussed to model these sorts of update constraints?

Suppose we have a standing relation, X, in the database, and a transient relation, UPD_X, holding today's new additions to X. We want to insert into X just those tuples of UPD_X whose values of the key-domain, KEY, do not already occur as values of KEY in X.

X % (KEY), is a relational value, with just one domain, of current keys occurring in X. By joining it to UPD_X we express just those tuples of UPD_X whose keys already occur in X:

```
X % (KEY) | KEY
  * UPD_X | KEY     <OTHER_DOMAINS>
```

By forming the 'DIFFERENCE' of this expression with the original UPD_X we express all those tuples of UPD_X whose keys do not already occur in

X. We now simply 'UNION' these with X to get NEW_X. Ignoring the special domain-overlapping notation, NEW_X is given by:

NEW_X = N!X + (N!UPD_X - (N!UPD_X * (N!X % KEY)))

Note that we have made NEW_X a derived relation by quoting the names of relations (N!) instead of their current values. NEW_X, upon being materialised, will contain the desired set of tuples, which may be used to replace the current value of X in the database. We must then ensure that X is only ever updated in this way. A crude way of doing this is to have the data dictionary keep a list of permissible assignments into given RELIDs, so that the application generator will not accept a command changing X except those, explicitly catalogued, which assign NEW_X into X.

Clearly a similar technique can be used to insert only those tuples into X whose KEYs occur in another relation, W. The tuples which fail to get into X can of course be recovered in the expression: UPD_X - X.

It is a critical business designing facilities for an application developer to impose constraints upon himself or his colleagues. It presupposes that both he and we know what sort of security we are supposed to be offering him. If this question is not resolved, it is so easy to end up with a security system which neither deters deliberate abuse, nor protects adequately against accidental misuse, but appears designed solely to encumber lawful operations. Our intentions with the data dictionary are primarily to reduce the incidence of subsequent mis-modifications to an application. As time goes on, or one gets further away from the designer of a particular component, the modifier of the component is unavoidably less well-informed as to the side-effects such modifications may have. On the other hand we make no attempt as yet to protect against deliberate wrecking.

Compare this with the 'facility' sometimes found in database packages which simply refuses to accept data with duplicate keys. The first non-DP user of such software is inevitably engaged in research, even if he does not know it, and is in the typical research predicament of having a file of grubby data he wishes to load up, precisely to use the sophisticated query facilities the database package may offer to report on such things as duplicate keys. He quickly has to learn a few DP skills, like how to manoeuvre around the trap for duplicate keys.

The relational data dictionary approach allows just that structure to

be put up first inside the database, which suffices to store the raw data, and adequate protection to be devised later for the use of the various relations of the application.

Compared to the task of defining relations to hold and manipulate the data of the application, as exemplified by X, it is much harder to write a satisfactory UPD_X to constrain its use. The latter is as demanding as writing a foolproof macro, which is really what UPD_X is. Later work might concentrate on providing relations like UPD_X 'off the peg', so to speak, that is as a result of some non-procedural specification by the application developer, rather than require him to develop the skill to construct them himself. Such 'constraining' relations may resemble the facilities available with CODASYL/DBTG (3), or IMS. Alternatively the sort of 'semantic constraint' which would be useful in practice might be thought out completely afresh.

Our approach contrasts with the CODASYL/DBTG approach of submitting the update constraints as part of the 'data definition', to be carefully separated from the 'data manipulation' activity. In the environment described it is difficult to distinguish between the two.

SUMMARY

We have tried to indicate a relational approach to many well-known problems of so-called data definitions. The key to this approach is the use of a data dictionary, itself maintained as a relational database.

Many details of this relational data dictionary clearly remain to be finalised. However the essence of a relational data dictionary is that it can be implemented even before such questions need be resolved. Thus the basic structure of, and facilities offered by, such a data dictionary can be changed extensively without the need to reload the stored data. This allows of considerable experimentation within a particular project.

REFERENCES

(1) S J P TODD: PRTV Overview,
IBM UK Scientific Centre report No 75, 1975.

(2) E F CODD: A Database Sublanguage founded on the Relational Calculus,
Proceedings of the 1971 ACM SIGFIDET Workshop on Data Description, Access and Control.

(3) CODASYL DBTG: Data Base Task Group Report April 1971.
Available from ACM, New York.