*Technical Report*

UCAM-CL-TR-22
ISSN 1476-2986

Number 22

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

# Constraints in CODD

M. Robson

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

*http://www.cl.cam.ac.uk/*

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

*http://www.cl.cam.ac.uk/TechReports/*

ISSN 1476-2986

# Constraints in CODD

M. Robson

University of Cambridge Computer Laboratory
Corn Exchange Street
Cambridge CB1 3QG
England

**Abstract**

The paper describes the implementation of the data structuring concepts of domains, intra-tuple constraints and referential constraints in the relational DBMS CODD. All of these constraints capture some of the semantics of the database's application.

Each class of constraint is described briefly and it is shown how each of them is specified. The constraints are stored in the database giving a centralised data model, which contains descriptions of procedures as well as of static structures. Some extensions to the notion of referential constraint are proposed and it is shown how generalisation hierarchies can be expressed as sets of referential constraints. It is shown how the stored data model is used in the enforcement of the constraints.

Part of this report is to appear as a paper in the Proceedings of the Second British National Conference on Databases, Bristol (1982).

## 1 INTRODUCTION

This paper describes the implementation of some important data structuring concepts within the relational DBMS CODD [5]. The concepts dealt with are domains, intra-tuple constraints and referential constraints. All of these allow the database to capture some of the semantics of its application. Domains capture the different meanings of different columns of relations. Intra-tuple constraints specify conditions which must hold between column values within a tuple. Referential constraints express structure which can span several relations.

These structuring primitives are not often provided within current systems. However, they are required in a DBMS if it is to be used as a base on which to build a more complex, higher level data model such as Shipman's DAPLEX [9] or Hammer and McLeod's SDM [4].

The structure imposed by the concepts described above could be implemented, for a particular database, by a suite of applications programs. However, the approach taken in this paper is to integrate facilities for describing the structure into the DBMS. The descriptions produced are stored in the database. This gives a centralised description of the structure of the database, in keeping with the idea of a conceptual schema. The definitions of domains and intra-tuple constraints can be expressed naturally as procedures for recognising valid domain elements or tuples. If this approach is adopted, their definition requires the ability to store procedures, as well as static structure, within the database. The implementation of referential constraints provides technology which is useful for the implementation of other data modelling facilities, in particular the generalisation hierarchies of Smith and Smith [10]. This paper describes a way in which all three types of constraint can be implemented.

The paper is organised into four main sections. The first describes the concepts of domains, intra-tuple constraints and referential constraints. It also suggests some extensions to the concept of referential constraints and shows how referential constraints can be used to specify generalisation hierarchies. The second section summarises those features of CODD which are relevant to the implementation described. The third section shows how the various forms of constraint are implemented within CODD.

## 2 THEORETICAL FOUNDATIONS

### 2.1 Domains in the Relational Model

Although the idea of a domain is central to much of the theory surrounding the relational model, it is an idea which has often been overlooked by those who have built relational systems, notably the builders of SYSTEM/R. The domains assigned to the different columns of a relation are of great importance in determining how that relation may be manipulated together with others. In particular the domains of a relation determine to which others it may be meaningfully joined. It is not sufficient to distinguish between, say, integers and strings since integers representing age and height have completely different meanings, they only 'look the same'. Such different meanings need to be distinguished.

Domains are in some ways like types in contemporary programming languages. In the same way that it is possible to define new types in some programming languages, it has been argued by McLeod [7], among others, that database management systems should allow the abstract definition of domains. In order to specify a domain to a DBMS it is necessary to provide the following:

(a) a description of the elements of the domain
(b) a specification of how the elements of the domain may be manipulated and compared; in particular, is the domain ordered?
(c) procedures to convert the external representation of the elements to their internal representation and vice versa.

(a) and (b) above are the abstract specification of the domain, whereas (c) is much more concerned with the implementation (of course, the information specified in (a) and (b) will probably heavily influence the way in which a particular domain is represented internally). It should, moreover, be possible to define a new domain as a restriction of a previously defined domain. The restriction is performed by specifying a predicate (filter) which is applied to a candidate value after it has been recognised as an element of the old domain.

### 2.2 Intra-tuple constraints

Intra-tuple constraints express conditions which must hold between the values of the attributes of a relation. For example, in a relation representing airline bookings the number of seats sold should not exceed the number of seats available. The intra-tuple constraints form part of the stored data model, in the same way as information about domains and referential constraints.

The form of the predicate which can be specified as an intra-tuple constraint is the same as that which can be specified as a test for the relational operation of selection. The difference is the time at which they are applied and the action taken if the tuple does not satisfy the

predicate. In the case of a selection predicate the tuple is discarded, whereas in the case of an intra-tuple constraint the update is rejected and the transaction aborted.

## 2.3 Referential Constraints

### 2.3.1 Definition

Referential constraints were proposed by Codd as part of his extended relational model RM/T [1], and modified for both theoretical and practical reasons by Date [2]. They are defined as follows.

There is a referential constraint between two relations R1 and R2 if:

(a) some subset of the attributes of R1 (the referencing attributes) form the key of R2;
(b) for every tuple in R1 where the referencing attributes are not NULL, there exists a tuple in R2 whose key is specified by the referencing attributes.

The relation R1 is called the referencing relation, and the relation R2 the referenced relation. The referencing and referenced attributes must, of course, have matching domains. The attributes specified in referential constraints are 'natural' attributes over which to perform joins in the database.

Referential constraints may be used to specify that the tuples in one relation must be a subset of the tuples in another (subset constraints). They may also express the fact that tuples of a relation represent associations between tuples of other relations. For example referential constraints could be used to specify that an assignment is an association between a student and a project.

All the referential constraints specified for a particular database form a dependency graph. This dependency graph may contain cycles which require the independent insertion of several tuples in order that the final database state will satisfy the constraints. Therefore, at some point the database will have to pass through an inconsistent state. Hence, referential constraints are checked at the end of transactions after all changes have been made to the database.

One relation may reference many others with the same set of referencing attributes. In such cases a quantifier may be specified by the user which indicates whether referenced tuples must exist in all, some of, or exactly one of, the referenced relations for the constraint to be satisfied. The default quantifier is "EXACTLY ONE OF".

A further part of the specification of a constraint is a statement of what changes must be performed on tuples in the referencing relation should referenced tuples be updated or deleted. The alternatives allowed

(with their names in brackets) are:

(a) that the update be disallowed if any references to the target tuple exist (RESTRICTED)
(b) that the referencing tuple be modified to maintain the link or that it be deleted as well (CASCADES)
(c) that the referencing tuple have its referencing attributes set to NULL (NULLIFIES).

The default for both update and deletion is RESTRICTED. The same update and deletion rules apply to all of the referenced relations for a particular constraint.

The cascade operation of (b) may produce further cascades. This occurs if the referencing attributes intersect the primary key of the referencing relation and the referencing relation is the referenced relation in some other constraint. For the rest of this paper the following syntax will be used for the definition of referential constraints:

constraint-name: ref ->> quantifier (ref) (ref)*
update-rule deletion-rule
where:
<ref> is relation-name . [attribute set]
Attribute sets are ordered lists of column names, and those on the left and right hand sides of the definition of a constraint must agree elementwise on domain.
'*' is arbitrary repetition.
<quantifier> is either empty or one of ALL, SOME, EXACTLY ONE OF
<update-rule> is either empty or "DELETION rule"
<deletion-rule> is either empty or "UPDATE rule"
<rule> is one of RESTRICTED, CASCADES or NULLIFIES

Relations are defined as follows (omitting the specifications of the domains associated with the attributes):

<relation name> : [attribute, ...¦ attribute, ..., attribute]
The attributes to the left of the '¦' are the key of the relation being defined.

## 2.3.2 Extensions

The following extensions to the definition presented above are useful.

(a) The effect of DELETION NULLIFIES is often not what is required by the semantics of an application. Consider the following example: within a college students have a tutor assigned to them. If a tutor leaves then all of his students are assigned, at least temporarily, to a special tutor, the senior tutor, the identity of whom is a property of the particular college.
Here what is required is not NULLIFIES but the retrieval of a suitable value from the database by means of evaluating a suitable

query. It must be ensured, however, that this "default" value is unique for a given database state.

(b) The definition of section 2.3.1 restricts references to be to the key attributes of a relation. However, references to non-key attributes can also occur. Consider the example database fragment:

```
Lecturers: [ Name ¦ ... ]
Courses  : [ Course ¦ Lecturer, .... ]
Exists-Lecturer: Courses.[Lecturer] ->> Lecturers.[Name]
```

In addition to the referential constraint 'Exists-Lecturer' there is also the constraint that every lecturer must teach at least one course. This constraint may be regarded as a reference from 'Lecturers' to the 'Lecturer' attribute of 'Courses'.

The need for the extra constraint is a consequence of the fact that referential constraints provide a way of expressing M(>=0):1 links. However, what is often required is M(>0):1 links, such as in the example above. It therefore seems reasonable to allow the definition of a referential constraint to specify that the link is M(>0):1 if this is what is required. Note, however, that DELETION RESTRICTED and UPDATE RESTRICTED become meaningless for such a constraint, since there must always be a referencing tuple.

(c) Having shown that 'backward' links occur when the referencing attributes are not the key of the referencing relation, such links certainly occur very often when the referencing attributes are the key of the referencing relation. Consider the example:

```
Students : [Name ¦ ... ]
Lecturers: [Name ¦ ... ]
People   : [Name ¦ date of birth, address, ... ]
Student-Is-Person: Students.[Name] ->> People.[Name]
                          DELETION CASCADES
Lecturer-Is-Person: Lecturers.[Name] ->> People.[Name]
                          DELETION CASCADES
Gen: People.[Name] ->> EXACTLY ONE OF (
                          Students.[Name]
                          Lecturers.[Name] ) DELETION CASCADES
```

The constraints in fact define part of a generalisation hierarchy, in which 'Students' and 'Lecturers' are specialisations of 'People'. Although in the example 'Students' and 'Lecturers' are disjoint sub-classes of 'People' this need not be the case (eg. replace EXACTLY ONE OF in GEN by SOME OF). Note that for sets of referential constraints which specify generalisations the update and deletion rule is always CASCADES.

The method of describing generalisation hierarchies used above is unsatisfactory, apart from being very long winded, since:

(i) it does not make clear that a particular set of referential constraints represent a generalisation;
(ii) it is not really desirable to name each link in a generalisation hierarchy. Therefore, such a shorthand syntax such as:

```
Is-Person: People.[Name] <->> EXACTLY ONE OF
                          ( Students.[Name],
                            Lecturers.[Name] )
```

is also unsatisfactory.

However, the fact that generalisation hierarchies can be represented by sets of referential constraints suggests that it is possible to maintain both by a common underlying mechanism. Such a common mechanism should not require that generalisations and referential constraints be represented in the same way in the data model. Indeed, since generalisations and referential constraints are semantically different they should be represented differently in the data model.

An implementation of these constraints will be described in section 4 but first it will be useful to present some brief details of the internal structure of CODD.

## 3 SOME INTERNAL DETAILS OF CODD

CODD is a general purpose, fully relational DBMS written entirely in BCPL and making heavy use of coroutines [8]. The following sections briefly describe those aspects of CODD which are relevant to the implementation described in section 4.

### 3.1 Basic Storage Structures in CODD

There are two storage regimes, one of which is used for storing fixed length data and the other of which is used for storing variable length data. The second complements the first as described below.

Fixed length data is stored in multi-level indexed sequential files. Care is taken to ensure that the index structure remains balanced when the file is updated. Within the file the data is sorted which makes the location of data items by binary chop very easy.

Variable length data is stored in a hashed storage system based on the dynamic hashing scheme of Larson [6]. The hashing scheme assigns a unique value to each object stored within it. This unique identifier has a fixed length and is called a value set identifier (VID). Duplicates are not stored and therefore two objects which are the same will be given the same VID. This gives a quick test for equality for variable length objects, which is often all that it is required to know. This storage regime can be used to store arbitrary sequences of bytes.

## 3.2 Relations and Inversions

Relations are stored as indexed sequential files, with tuples of fixed size. Variable length data within a tuple is stored in the hashed storage system and its VID is stored in the file representing the relation. Typically the file representing a relation will be sorted on its primary key fields, since this allows easy selection on key. However, there are occasions when it is wished to access a relation other than by its primary key. This occurs, for example, when performing a join other than on the key or checking a referential constraint (see section 4.4). For these reasons CODD supports inversions. Relations are represented by a set of files consisting of a primary version (sorted on the key of the relation) plus a number of inversions. All alterations to the relation are automatically performed for all of the files representing it.

When an inversion is defined it is automatically loaded with the relevant data sorted in the correct order. The database integrity scheme described below ensures that, in the event of a system crash, alterations are seen to occur to either all or none of the versions of a relation.

## 3.3 CODD Catalogue Structure

The structure of a CODD database is described by a normalised set of relations, the system catalogue relations. Information about relations is held in two relations. One, called RNAMES, contains the following information:

(a) the name of the relation (this is the key of RNAMES)
(b) a list of pairs (column name, domain name) encoded onto a VID
(c) the degree of the relation
(d) the key of the relation
(e) the cardinality of the relation
(f) insertion predicate for this relation, encoded onto a VID.

The second relation, called INVERSIONS, describes the physical storage of the relation. INVERSIONS relates the relation name and a permutation of its columns to a disc address, the relation name and permutation are the key of INVERSIONS.

Similar catalogue relations are used to store the descriptions of domains and referential constraints (see below).

The use of relations to store catalogue information has the advantage that it requires no special storage structure for the catalogues. Also it provides the ability to manipulate the catalogue relations in the same way as any other relation in the database. However, since their use is highly specialised they are usually manipulated by special purpose code.

## 3.4 Physical database integrity

The scheme for database integrity, which gives the facilities to commit and abort transactions, is described fully in [5]. It is based on the use of bitmaps to record the allocation of database pages, and on having a special page (page zero) which defines the previous consistent database state. Page zero is updated and written back when a transaction commits, and it is only at this point that information about the old database state is lost. Abort is achieved by restoring the DBMS's view of the database to that recorded on the old page zero. This mechanism provides indivisible updates and ensures that inversions will be maintained in step even over system crashes.

## 3.5 Pipelines in CODD

Pipelines are a basic feature of CODD. A query is viewed as a directed graph. The nodes of this graph represent the relational operations to be performed and the edges of which represent pipelines passing data between nodes. The nodes of the graph are implemented as coroutines which contain the algorithms for the particular operation to be performed. This structure operates as a demand driven computation pulling tuples along the pipelines as required.

CODD provides facilities for dynamically modifying the executing pipeline structure. In order to achieve this the DBMS maintains a model representing the computation in progress. This technology was originally developed for dealing with conditionals in queries, but as will be shown in section 4.4 it and the rest of the pipeline technology is also useful when referential constraints are being maintained.

## 4 IMPLEMENTATION OF CONSTRAINTS

This section shows how the various types of constraint discussed in section 2 have been implemented in CODD. In particular it shows how the storage structures and pipelines described in section 3 have been exploited in this implementation.

### 4.1 An Example Database

This is part of a database describing a university:

```
Undergraduates: [Name ¦ Supervisor, Tutor, Year, ... ]
Graduate-students: [Name ¦ Department, Supervisor, ... ]
Lecturers: [Name ¦ Department, Date appointed, ... ]
Projects: [Title ¦ Proposer, Description, ... ]
Courses: [Name ¦ Lecturer, ... ]
Assignments: [Student, Project ¦ Date started, ...]

Assign1: Assignments.[Student] ->> Undergraduates.[Name]
                                   DELETION CASCADES
Assign2: Assignments.[Project] ->> Projects.[Title]
                                   DELETION CASCADES
Exists-Supervisor:
    Undergraduates.[Supervisor] ->> EXACTLY ONE OF
                                  ( Graduate-students.[Name],
                                    Lecturers.[Name]
                                  )
Exists-Lecturer: Courses.[Lecturer] ->> Lecturers.[Name]
```

### 4.2 Domains

One way of defining domains in CODD is by writing a program which accepts valid external representations of the domain elements and converts them into a suitable internal representation. In addition to a program for recognising the domain elements, a routine doing the reverse conversion must be supplied. This routine is typically used when printing values. Domains defined in this way are called base domains. Base domains are assigned an internal type. As was noted in section 2.1 the type chosen will depend upon, among other things, the orderedness or otherwise of the domain.

There are four internal types, two of which are ordered and two of which are unordered. The ordered types are integers and fixed length character strings. The unordered types are arbitrary sequences of bytes and boolean values.

The code for recognising and printing elements of a domain is known to the database via the name of the domain. Given this name the DBMS can find and load the required code. The runtime system of BCPL provides facilities for these routines to be accessed and used. The binding between domain and type is achieved by the command:

CREATE DOMAIN domain name TYPE internal type
eg
CREATE DOMAIN number TYPE integer

Ideally programs such as 'number' should be part of the database rather than the DBMS since the domains of a database can be application-specific rather than general to all problems. Some such programs will be required so commonly that they should be included as part of the database when it is created; the domains 'numbers' and 'strings' are examples of such domains.

The base domains provide the basis on which to build hierarchies of derived domains, each with a base domain as its root. These are domains defined in terms of previously defined domains by the command:
CREATE DOMAIN Domain name ON Old domain name FROM predicate
the 'Old domain name' may be either a base domain or a previously defined derived domain. The 'FROM predicate' part of the definition may be omitted or be a general condition which will restrict which values of the old domain are permissible in the new domain.
Examples:

(a) CREATE DOMAIN project-numbers ON numbers FROM [1..10000]
This defines the domain 'project-numbers' to be those integers in the range 1 to 10000.

(b) CREATE DOMAIN age ON numbers
CREATE DOMAIN distance ON numbers
These two domains have the same set of values but are regarded as having different meanings.

(c) CREATE DOMAIN departments ON strings FROM "Physics" | "English"
This defines the domain 'departments' to contain only the strings "Physics" and "English".

The domain predicates are 'compiled' into a format known as indirect threaded code. This is an interpretable code format based on that used in the implementation of macro-SPITBOL [3]. The threaded code for an expression is essentially post-fixed polish representation of the expression, with one extension described below. The interpreter uses a stack for intermediate results and literal data is included in line. The interpreter itself is a branch table indexed by the function codes in the string to be interpreted, and its performance is good in that it evaluates even quite complex predicates quickly. The predicates themselves are stored in the database in the hashed storage system, from which they are unpacked when they are needed. The same technique is used for the construction of predicates for the relational operation of selection.

I mentioned above that there is one extension to the predicates being simple post-fixed polish representations of boolean expressions. This extension is best described by considering as an example the domain 'departments' defined above. Suppose that the list of names of departments

had contained several hundred entries instead of just three. It would be quite inefficient to implement a predicate that performed the test on whether a string was a valid department name as follows:

```
TEST department.name = department1 THEN success
ELSE TEST department.name = department2 THEN success
   .
   .
ELSE failure
```

It is much better to enumerate the set of valid department names in a file and have an instruction in the threaded code of 'look up value in set'. Such lists are stored in CODD as sorted lists of values in an indexed sequential file. If the items are not of fixed length then a set of value set identifiers is stored.

It should be noted that although 'enumeration' lists may look like single column relations they are not treated as such, since unlike the contents of a relation they represent a fixed part of the data model. That is, their purpose and significance is different from that of tuples in relations. In particular, the user is not free to add or delete values at will from domains, although the database administrator may have need to do so. These differences in 'meaning' should not, however, stop the use of similar implementation techniques.

The information about domains is stored, like that for relations, in a catalogue relation which forms part of the stored data model. This relation, called DOMAINS, is defined as:

Domains: [ domain name ¦ Underlying domain name, Type, Predicate ]

with the convention that for base domains

Domain name = Underlying domain name

In CODD the information about domains is used to check that:

(a) operations requiring matching domains have valid arguments;
(b) constant values, which are supplied as selection criteria, are from the same domain as the attributes to which they are being compared;
(c) the values of the columns of a tuple are from the correct domain when the tuple is being updated or inserted.

Some of the operations included under (a) are join, set union and the definition of referential constraints.

## 4.3 Intra-tuple constraints

In the same way that domain predicates can be stored as interpretable code within the database, so also can predicates representing intra-tuple constraints. They are stored as an attribute of the RNAMES catalogue relation, and are therefore part of the stored data model.

The update predicate forms part of the argument to the update component of CODD. Instead of just performing the update the update program first tests the new tuple value with the update predicate, for the relation being modified, and the tuple is inserted or altered only if this test returns 'true'; otherwise the update is rejected and an error message produced. This use of update filters is similar to that described by Stonebraker [11].

## 4.4 Referential Constraints

There are two components of CODD which are important for the maintenance of referential constraints. These are:

(a) the update program
(b) a constraint checking program. This is, in some sense, a query that is evaluated at the end of any transaction involving an update. The result of this program determines whether to commit or abort the transaction.

For their operation both of the above components rely on having a representation of the dependencies between relations which are imposed by the referential constraints. These dependencies constitute the reference graph. Within a CODD database the reference graph is stored in two relations, the reference graph relations, as proposed by Date. These relations are defined as:

RGX: [ Constraint name ¦ Referencing relation name,
Referencing attributes, quantifier,
update rule, deletion rule ]
RGY: [ Constraint name, Referenced relation name ¦
Referenced Attributes ]

The attributes REFERENCING RELATION NAME and REFERENCED RELATION NAME are lists of column names held as value set identifiers. The reference graph relations for the example database are shown in figure 1.

The syntax used to specify referential constraints to CODD is that used for the examples in this paper. When a constraint is defined it is checked that it is consistent with the stored data model. This requires that the relations specified exist, that the referencing and referenced attributes match in domain (the domain information being held in the catalogues), and that the referenced attributes are the key of the referenced relation.

RGX:

| Constraint Name | Referencing Relation | Referencing Attributes | Quantifier | Update Rule | Deletion Rule |
|---|---|---|---|---|---|
| Assign1 | Assignments | Student | EXACTLY ONE | R | C |
| Assign2 | Assignments | Project | EXACTLY ONE | R | C |
| Exists-Supervisor | Undergraduates | Supervisor | EXACTLY ONE | R | R |
| Exists-Lecturer | Courses | Lecturer | EXACTLY ONE | R | R |

RGY:

| Constraint Name | Referenced Relation | Referenced Attributes |
|---|---|---|
| Assign1 | Undergraduates | Name |
| ASsign2 | Projects | Title |
| Exists-Supervisor | Graduate-Students | Name |
| Exists-Supervisor | Lecturers | Name |
| Exists-Lecturer | Lecturers | Name |

In RGX for the update and deletion rules R is RESTRICTED and C is CASCADES.

Figure 1: The Reference Graph Relations for the Example

When a relation is to be updated the information in RGX and RGY is used to build an in-store representation of the dependency structure applying to the relation. This data structure is a directed graph with two types of node. One of the types of node represents relations and the other represents constraints. The constraint nodes contain information about quantifiers and update and deletion rules. Relation nodes only point to constraint nodes and vice versa.

The structure of a relation node is:

(a) relation description as obtained from the RNAMES catalogue
(b) number of references from this relation
(c) number of references to this relation
(d) for each constraint node for which this relation is the referencing relation:
    (1) pointer to relevant constraint node
    (2) referencing attributes
(e) for each constraint for which this relation is the referenced relation:
    (1) pointer to the relevant constraint node

The structure of a constraint node is:

(a) pointer to referencing relation node
(b) number of referenced relation nodes
(c) update rule
(d) deletion rule
(e) quantifier
(f) for each referenced relation node:
    (1) pointer to referenced relation node
    (2) correspondence between referencing attributes and the key of the referenced relation.

This structure is used to control the cascading of alterations and deletions and the testing of constraints at the end of a transaction. The structure forms part of the environment in which both the update program and the constraint checker function.

The update program takes as input a stream of tuples to be inserted, deleted or altered and produces two outputs. These outputs are:

(a) a stream of keys, to be passed to the constraint checker at the end of the transaction;
(b) a stream of tuples which are the cascaded updates or deletions.

The constraint checking program takes as input a list of objects of the form:

(pointer to constraint node, list of tuples)

one for each constraint to be checked. The constraint node is used to determine the referenced relations and the relevant quantifier. The list of tuples is produced from the referencing attributes of the referencing relation and will be a list of keys of the referenced relation. Consider as an example the constraint node for the constraint 'Exists-Supervisor' in the example of section 4.1. For each input object the constraint checker verifies that the referential constraints are satisfied. If any constraint is violated the whole transaction is aborted, otherwise it is committed.

The keys to be checked for referential constraints are passed between the update program and the constraint checker by means of a flexible buffer. This is a pipeline which can store, in principle, an unlimited number of tuples, those tuples which cannot be held in store being written to disc.

Exactly how the cascading of updates is organised depends on the structure of the part of the dependency graph representing the cascades required for a particular update. There are three classes of graph:

(a) trees
(b) cyclic graphs (eg. a relation referencing itself)

(c) acyclic graphs which fork then join again.

In case (a) it is possible to build update nodes for each relation in the cascade and connect them to their producers by pipes which will carry the tuples they are to insert or delete. In this case all of the updates can be performed in parallel since they do not affect each other. Note that not all of the update nodes may in fact operate since some may get no input.

For case (b) not all of the update nodes can be built before the update commences. More structure will be built if the output of cascaded tuples from the cycle is not empty.

In both of cases (a) and (b) standard CODD pipeline technology can be used to organise the pipes required between update nodes in cascades. Cascades with multiple consumers are dealt with by the use of COPY nodes. These are pipeline nodes which produce two output streams each of which is a copy of a single input stream. The facilities available for conditionally building pipeline structure, which was developed to incorporate data dependent tests into queries, are used to deal with cyclic graphs. In this case flexible buffers are required to hold cascaded tuples until the next level of update routines are built and activated.

There are potential problems with case (c) which have not yet been fully investigated. These are:

(a) if two streams of cascaded updates are to be applied to the same relation because the cascade graph had previously forked, the result of these updates might depend upon the order in which they are applied;
(b) if two streams, one a cascade of deletions and the other a test for deletion RESTRICTED are incident on the same relation, the result may depend on whether the deletions or the test is performed first.

In constructing the graph representing the cascades which are to be performed, the tree is always terminated either by UPDATE/DELETION RESTRICTED or when the referenced relation is not itself referenced. As mentioned above, however, not all the cascades may in fact occur since the altered or deleted tuples may not be referenced, and hence not produce tuples for the cascade.

In the maintenance of referential constraints the presence of inversions allows the tests for the update and deletion rule RESTRICTED to be made easily. Also it is possible to decide on which tuples to add to a cascaded stream very easily. Therefore, the definition of a referential constraint causes the automatic creation of the necessary inversions. For example, the definition of the constraint 'Assign2' would cause an inversion of 'Assignments' sorted on 'Project' to be created.

### 4.4.1 Dynamic Definition of Referential Constraints

In the above discussion it has been tacitly assumed that the data model is a static object. Indeed in an ideal world this would be the case: the data model would be defined once (correctly) and would never need to be altered. However, the world is not like this. Therefore, it must be possible to create new constraints and delete old ones. Deletion of old constraints causes no problems. However, the creation of a new constraint requires that the current database state be tested to check that it satisfies the new constraint. The definition of a new constraint must, therefore, automatically generate such a check, and produce output indicating in what ways, if any, the current state violates the constraint.

### 4.4.2 Replacing NULLIFIES by a computed value

This requires some way of storing in the database the rule for deriving the computed value. In CODD this rule will be in the form of a relational algebra expression. Such expressions can be represented by prefixed polish strings which can then be stored in the value set mechanism and their unique identifiers stored in RGX instead of the simple update or deletion rule. A query can be constructed from the prefixed polish string. If the value which this query will yield is required then the query can be evaluated in the same way as any other query.

### 4.4.3 References to non-key attributes

A solution to this problem, which was described in section 2.3.2, can be illustrated by considering the constraint 'Exists-Lecturer' in the example of section 4.1. Suppose that there is also the constraint that each lecturer must give at least one course.

In the scheme described above this is easily tested since the inversion required already exists to deal with the alteration and deletion of tuples in 'Courses' under the constraint 'Exists-Lecturer'. Therefore all that is required is an indication in the catalogues to indicate that whenever a tuple is inserted into 'Lecturers' or deleted from 'Courses' the constraint that every lecturer must teach one course is to be tested. The constraint checking program can easily be extended to perform this check at the same time as other referential constraints are checked.

## 4.5 Generalisation

Given the mechanism described in section 4.4, it is possible to maintain generalisation hierarchies provided that a suitable dependency graph can be constructed from the catalogue relations which represent the generalisations.

A suitable representation for the hierarchy is:

```
Gen-Links: [ Child, Parent ¦ ]
Gen-Nodes: [ Node ¦ Partition/Cover ]
```

where there is a tuple in 'Gen-Nodes' for every type which is the supertype of a generalisation. The attribute 'Partition/Cover' specifies whether or not the populations of the subtypes of this type are disjoint.

From the tuples of these two relations the required dependency structure can be constructed since:

(a) the update and deletion rules are always CASCADES
(b) the referencing and referenced attributes, since they are always the keys of the relations representing the subtypes and supertypes, can be recovered from the catalogue representing the relations.

## 5 CONCLUSIONS

It has been first shown how facilities for describing some powerful data structuring primitives have been incorporated into a particular DBMS, and further how the descriptions they produce can be stored in the database itself. This is an important step forward in the incorporation of semantic information into the database. The centralised database description produced is in keeping with the idea of a conceptual schema. Secondly, some extensions to the idea of referential constraints have been presented and it has been shown that these fit into the implementation described. Thirdly, the paper has demonstrated that generalisation hierarchies can be specified and maintained by using referential constraints as the basic building blocks.

It should be noted that in performing the work described a number of features of the particular DBMS used were of great help; in particular, the pipeline structures which were available. These pipelines provided a natural way to express the cascading of updates required for the implementation of referential constraints. This demonstrates that pipelines are useful for the maintenance of constraints during update as well as for retrieval.

Although, as Codd has stated in [1], the task of capturing the meaning of data is a never-ending one, the work described here represents a useful contribution.

**Acknowledgements**

I would like to thank M.A. Gray, C. Jardine, Dr. J.K.M. Moody and Dr. K. Sparck Jones, both for reading drafts of this paper and for participating in discussions on the ideas described herein.

## REFERENCES


[1] Codd E.F.
Extending the relational model to capture more meaning.
ACM: Transactions on Database Systems Vol 4 No 4 pp397-434 (1979)

[2] Date C.J.
Referential Integrity.
Proceedings of the 7th International Conference on Very Large Databases, Cannes, France. pp2-12. (1981)

[3] Dewar B.K. and McCann A.P
MACRO SPITBOL: A SNOBOL4 Compiler
SOFTWARE: Practice and Experience Vol 7 pp95-113 (1977)

[4] Hammer M. and McLeod D.J.
Database description with SDM: A Semantic Data Model
ACM: Transactions on Database Systems Vol 6 No 3 pp351-386 (1981)

[5] King T.J.
The design and implementation of a relational database for Historical Records.
Ph.D. Thesis. University of Cambridge (1979)

[6] Larson P.A.
Dynamic Hashing
BIT Vol 18 pp184-201 (1978)

[7] McLeod D.J.
High Level Domain Definition.
ACM: SIGPLAN-Notices Vol 11 Special issue, Proceedings of the Conference on Data: Abstraction, definition and structure pp47-57

[8] Moody J.K.M. and Richards M.
A coroutine mechanismn for BCPL
SOFTWARE: Practice and Experience Vol 10 pp765-771 (1980)

[9] Shipman D.W.
The functional data model and the data language DAPLEX.
ACM: Transactions on Database Systems. Vol 6 No 1 pp140-173 (1981)

[10] Smith J.M. and Smith D.C.P.
Database Abstractions: Aggregation and Generalisation.
ACM: Transactions on Database Systems. Vol 2 No 2 pp105-133 (1977)

[11] Stonebraker M.
Implementation of Integrity Constraints and Views by Query Modification.
Proceedings ACM-SIGMOD Conference 1975 pp65-78