Technical Report

UCAM-CL-TR-62
ISSN 1476-2986

Number 62

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Constraint enforcement in a relational database management system

Michael Robson

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

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted March 1984 by the author for the degree of Doctor of Philosophy to the University of Cambridge, St John's College.

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

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Summary

The dissertation describes the implementation of the data structuring concepts of domains, intra-tuple constraints and referential constraints in a relational database management system (DBMS). The need for constraints is discussed and it is shown how they can be used to capture some of the semantics of the database's application. The implementation described was done within the framework of the particular DBMS CODD, the main features of which are presented.

Each class of constraint is described and it is shown how each of them is specified to the DBMS. The descriptions of the constraints are stored in the database giving a centralised data model, which is used in the enforcement of the constraints. This data model contains descriptions not only of static structures but also of procedures to be used to maintain constraints. A detailed account is given of how each type of constraint is maintained.

The main focus of the dissertation is on referential constraints since inter-relational structure is an area in which relational systems are particularly weak. Referential constraints impose a network structure on the database and it is shown how referential constraints can be maintained by interpreting this network, using the data-pipelining facilities provided by CODD. It is also shown how referential constraints can be used to construct generalisation hierarchies, themselves an important data modelling tool. Further, some extensions to referential constraints, which allow them to capture more semantics, are suggested. The usefulness of referential constraints is illustrated by presenting a real database example (that of the University Computing Service), on which some of the ideas described in the dissertation have been tested.

Preface

I wish to thank the Science and Engineering Research Council and St. John's College for their financial support and my employers, Shape Data Ltd, for their understanding while I have completed the dissertation. My thanks also go to Professor Needham and the members of the Computer Laboratory for providing a pleasant and stimulating environment in which to work.

Of the many people who have helped and guided my research I would like to thank particularly Ken Moody, Charles Jardine and Tim King. I would also like to acknowledge the help and general encouragement provided by Dr. Karen Sparck Jones, Branimir Boguraev and John Tait. Finally I would like to thank all my friends who have made my time as a research student enjoyable.

I hereby declare that this dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university. I further state that no part of my dissertation has already been or is being currently submitted for any such degree, diploma, or any such qualification.

Except where otherwise stated in the text, this dissertation is the result of my own work, and is not the outcome of work done in collaboration.

CONTENTS

1 INTRODUCTION 1
1.1 The Relational Model 2
1.2 Other Data Models 6
1.2.1 Abrial: Binary Relational Models 7
1.2.2 The Semantic Hierarchy Model 7
1.2.3 The Entity-Relationship Model 8
1.2.4 DAPLEX 9
1.2.5 The Network Model (CODASYL) 10
1.2.6 Semantic Networks 10
1.2.7 Programming Languages 11
1.3 The Work to be Done 11
1.4 Summary 13

2 THE STRUCTURE OF CODD 14
2.1 A Brief History 14
2.2 Pipelines in CODD 15
2.3 Database Structure and Integrity 16
2.3.1 Database Integrity: The State Page, Commit and Rollback 17
2.3.2 Basic Storage Structures in CODD 17
2.3.3 Relations and Inversions 18
2.4 Resource Allocation: Pools 19
2.5 Catalogue Relations 20
2.6 The Phases of Execution of a Transaction 21
2.6.1 Construction of Executable Structure: The STAGE Graph 22
2.6.2 Query execution: SCHEDULER, EXECUTER and KILLER 23
2.7 A Pipeline Node Atlas 24
2.7.1 BUFR and BUFW: The Terminal Producer and Consumer 24
2.7.2 READR: Read a Relation 25
2.7.3 WRITR: Write a Relation 25
2.7.4 COPY: Produce Two Copies of a Stream 25
2.7.5 Flexible Buffers 25
2.7.6 COUNT: Count the Number of Tuples in a Stream 26
2.7.7 UPDATE: Modify a Relation 27

3 DOMAINS 28
3.1 Domains in the Relational Model 28
3.2 Implementation of Domains in CODD 29
3.2.1 Implementation of Domain Checkers 32
3.2.3 The Storage of Code in the Database 33
3.3 Manipulating Domain Elements 34

4 INTRA-TUPLE CONSTRAINTS 35

5 REFERENTIAL CONSTRAINTS: Theory 37
5.1 Introduction 37
5.2 Definitions 38
5.3 Checking Constraints 42
5.4 Examples of Referential Constraints 43
5.5 The Form of the Reference Graph 45
5.6 How is the Cascading of Updates and Deletions Organised? 45
5.7 Comparison with other work 46
5.7.1 Aggregation 46
5.7.2 Addis's Extended Relational Analysis 47
5.7.3 Triggers and Assertions in SEQUEL 47
5.7.4 Codd's Extended Relational Model 48
5.7.5 Functions in DAPLEX 49
5.7.6 CODASYL 49
5.8 Extensions to the Idea of Referential Constraints 49
5.8.1 Replacement of DELETION NULLIFIES 50
5.8.2 References to non-key attributes 50
5.9 Generalisation 52
5.9.1 Generalisation and Referential Constraints 52
5.10 Summary 54

6 REFERENTIAL CONSTRAINTS: Implementation 55
6.1 Basic Strategy 55
6.2 Catalogue Representation of the Reference Graph 56
6.3 Recovery of Cascade Graph from the Catalogues 57
6.4 Checking the Constraints 61
6.5 Construction of the Coroutine Structures 65
6.5.1 An extension 69
6.6 Evaluation of Cascade Structures 70
6.6.1 Acyclic graphs 70
6.6.2 Cyclic graphs 70
6.7 The Definition of Constraints 70
6.7.1 Definition of Constraints to the DBMS 70
6.7.2 Dynamic Definition of Referential Constraints 71
6.8 Implementation of the Extensions Suggested in Section 5.8 71
6.8.1 Replacing NULLIFIES by a computed value 72
6.8.2 References to non-key attributes 72
6.9 A Different Way of Maintaining Inversions 73
6.10 Conclusions 73
6.11 Algorithms for constructing cascade graphs 73
6.11.1 Recovery of Reference Graph Fragment for INSERT 74
6.11.2 Recovery of Reference Graph Fragment for ALTER 75
6.11.3 Recovery of Reference Graph Fragment for DELETE 76

7 AN EXAMPLE DATABASE: The University Computing Service 77
7.1 The World of the Database 78
7.2 The Accounting Structure 78
7.3 Users and Resources Associated with Users 80
7.4 Authorisations: The link between users and the accounting tree 81

7.5 Examples of the Maintenance of Referential Constraints 85
7.5.1 Insertion of a single tuple into Authorisations 85
7.5.2 Insertion of a new project and an authorisation to use it 86
7.5.3 Moving a project from one account to another 88
7.5.4 Deletion of a tuple from Authorisations 89
7.5.5 Deletion of a tuple from Users 89
7.5.6 Deletion of the root of the account-group tree 90
7.6 Experience with this Database 94
7.7 Summary 94

8 CONCLUSIONS 95
8.1 Review 95
8.2 Conclusions 96
8.3 Critique 98
8.2.1 The Maintenance of Referential Constraints 98
8.3.2 Bulk Update 99
8.3.3 User Interface 99
8.3.4 Use in a Real Application 100
8.5 Further Work 100

REFERENCES 101

Chapter 1

INTRODUCTION

This dissertation deals with the enforcement of constraints in a relational database management system.

Constraints are conditions which the contents of a database must satisfy if the database is to remain a plausible model of its application; they express important information about the application. Constraints can be represented in two ways:

(a) Constraints can be built into the basic structures of the data model which is being used (Inherent constraints). An example of this type of constraint is the restrictions on simple data values which are implied by domains in the Relational Model.

(b) Explicit constraints can be specified, either by grafting a constraint specification language on to the basic data model or by embedding the constraints in programs which manipulate the database. Explicit constraints provide a flexible mechanism for augmenting the ability of a data model to express constraints. [Eswaran, Chamberlin 75], [Hammer, McLeod 76], [Stonebraker 75] and [Weber, Stucky, Karszt 83] all give proposals for explicit constraint schemes for the Relational Model.

It should be noted that even models which can express a considerable amount of semantic information in their basic structure allow themselves a way to specify explicit constraints (e.g. DAPLEX [Shipman 81]). [ISO 81] and [Tsichritzis, Lochovsky 82] provide detailed comparisons of different data models and the structures which they provide; the latter also serves as a comprehensive tutorial on the aims and techniques of data modelling.

Constraints in a database serve two purposes.

(a) They define rules to the DBMS which can then be enforced, preventing the insufficiently informed (or possibly malicious) user or program from modifying the database in a way which would make the database inconsistent.

(b) They inform the user of the rules which the content of the database must satisfy.

The latter purpose is better satisfied if the constraints are described in the data model, rather than being hidden in the programs which manipulate the database.

This dissertation investigates the implementation of three types of constraint in a relational DBMS, namely:

- o Constraints on atomic data values (domains)
- o Constraints on atomic sentences (intra-tuple constraints)
- o Functional constraints between members of (different) classes (referential constraints)

Referential constraints are the class of constraint discussed at greatest length in this dissertation; it will be shown that they form a base from which other data modelling constructions can be built.

The work described in this dissertation has been carried out within the framework provided by the Relational Model and with a particular database management system (CODD) which has a novel mechanism (data pipelining) for query evaluation; this mechanism has been exploited in the work described.

1.1 The Relational Model

The Relational Model of data has been the subject of considerable interest over the last ten years; however, it is notably deficient in the facilities which it provides for expressing and maintaining constraints.

When the Relational Model was introduced by Codd [Codd 70] it seemed very attractive, owing to its firm mathematical foundation and the high level operators which it provides for the manipulation of data. There was a great deal of debate at the time as to the relative merits of the Relational Model and the network model (CODASYL) which was being developed at much the same time. Really the debate was concerned with the techniques typically used to interrogate the database, the Relational Model, on the one hand, providing high level operators for manipulating sets of data items and the CODASYL model, on the other hand, providing mechanisms for navigating around a network of records. Also the Network Model could not point to a firm mathematical foundation in the way that the Relational Model could. The debate could hardly have been about storage structures since (with the benefit of hindsight) the file structures which are found in the relation storage system of System/R [Astrahan, et al 76] bear a close resemblance to those found in CODASYL systems. In retrospect, therefore, it is the high level operators which are the really attractive, important feature of the Relational Model; they give a well defined framework in which to

discuss the problem of querying a database. However, it should be noted that high level interfaces have also been developed for CODASYL databases; for example the work by Gray on ASTRID [Gray 81] and the work by Buneman on FQL [Buneman, Frankel, Nikhil 82] (the latter was demonstrated on a CODASYL system although the work is more general). Although the Relational Model provides high level operators for database query, it provides only low level operations for database modification, i.e. insert, delete and modify tuples of relations.

The Relational Model provides two structures for capturing semantic information, namely domains and relations.

- Domains are the sets from which the atomic data values stored in the database are obtained; the sets of valid salaries, part numbers or dates are examples of domains. The values of a domain are independent of the content of the database.

- Relations describe sets of entities or relationships in the application; a relation may represent the fact that people have ages or that some people are married. The elements of relations (tuples) express facts about particular entities of relationships. For example, tuples express facts like "Andy is 28 years old" and "Andy is married to Lynne". In some cases there are constraints between the values in tuples; for example, people cannot be married to themselves.

However, the Relational Model does not provide any way of describing the connections between relations. This is a serious defect of the model: it has been remarked above that relations represent facts; facts rarely exist in isolation; they are connected with other facts. The information about the connections between the facts represented by different relations should be known when a database schema is being designed. At what point in the database design of a relational schema is the information lost? Consider the design of a relational database schema.

A tool which is used in the construction of relational database schemas is the functional dependency. A functional dependency describes a functional relationship between attributes within a relation; for example if a person's name determines his date of birth then the person's date of birth is functionally dependent on his name. Functional dependency will be denoted as:

Person.[Name] ==> Person.[Date of birth]

Functional dependencies are used for the identification of relation keys and as an aid in converting a relational database schema into third normal form; this latter process is called normalisation. The aim of third normal form is to achieve the state of "one fact, one place". If the database schema has this property then it is free from a number of

data manipulation anomalies which can be identified by considering the implications of inserting, deleting or modifying tuples of a relation.

The data manipulation anomalies which can arise are best illustrated by use of an example. Consider the database researchers' old friend, the suppliers and parts database; we start with the relation 'FIRST' with attributes:

FIRST : [Supplier, City-of-business, Part, Quantity-in-stock]

which represents the facts that suppliers are based in cities and have particular quantities of particular parts in stock.

The data manipulation anomalies can now be illustrated as follows.

- o Insertion - when a supplier starts to supply a new part then it is possible that his city of business will be specified incorrectly
- o Deletion - if a supplier stops trading then if he was the only current supplier of widgets the information that widgets are parts is lost.
- o Modification - to change the city of business of a supplier who supplies a large number of parts is not simple, all tuples which refer to the supplier have to be modified.

These anomalies arise because the relation 'FIRST' contains the functional dependencies:

FIRST.[Supplier] ==> FIRST.[City-of-business]

and

FIRST.[Supplier,Part] ==> FIRST.[Quantity-in-stock]

These functional dependencies can be used to normalise 'FIRST' into the two relations 'SUPPLIERS' and 'SUPPLIERS-PARTS' defined as:

SUPPLIERS : [Supplier, City-of-business]
SUPPLIERS-PARTS: [Supplier, Part, Quantity-in-stock]

If the current population of known parts is of interest, as is implied by the discussion of manipulation anomalies, then there will also be the relation 'PARTS', defined as:

PARTS : [Part]

During this normalisation the information that suppliers in 'SUPPLIERS-PARTS' must also be present in 'SUPPLIERS' and that parts in 'SUPPLIERS-PARTS' must be known parts (i.e. in 'PARTS') has been lost. When normalisation is complete the only functional dependencies which

remain in the relational schema are those which define the keys of relations.

The above example illustrates a problem with normalisation; the problem is that in the quest for freedom from some simple data manipulation anomalies, new anomalies are introduced. The new anomalies are the result of constraints between the content of different relations. These constraints were captured by functional dependencies which were discarded once the objective of "one fact, one place" was achieved. This is sad, since these functional dependencies capture a great deal of information about the application.

Often normalisation is presented as a mechanical process for designing relational database schemas which takes as its input a large collection of attributes and a set of functional dependencies and produces as output a normalised relational schema. This view of schema design is criticised heavily and correctly in [P.King 80], which points out that in practice schema design proceeds by synthesis rather than by decomposition. King also remarks that a database relation is not purely a formal object (i.e. a set of tuples), it also has associated with it a well defined natural language predicate which specifies the meaning of the relation. Similar comments are also made in [Borkin 80a] and [Smith, Smith 77b]. However, regardless of criticisms of normalisation when it is regarded as a purely formal process, functional dependencies are a useful tool for analysing a schema for data manipulation anomalies.

It is interesting to note that in some early work on constraint enforcement in the Relational Model (e.g. [Stonebraker 75]) the problem illustrated above is described in terms of subset constraints, that is for the suppliers and parts database, constraints like:

The suppliers in 'SUPPLIERS-PARTS' are a subset of those in 'SUPPLIERS'

rather than the functional constraint which was expressed by the original functional dependency. The functional form of the constraint is captured nicely by the notion of Referential Integrity [Date 81a], which plays an important part in Codd's extended Relational Model, RM/T [Codd 79]. [Date 81a] is a concrete proposal for extending the Relational Model. Date's proposal considers the inter-class links as constraints which must be satisfied if the database is to remain consistent and suggests a set of rules which govern how the constraints interact with the insertion, deletion and alteration of tuples in relations. The greater part of this dissertation (chapters 5 to 7) is devoted to referential integrity and its maintenance in a relational database management system. It is also worth noting that the functional form of the constraint appears in a number of data models, for example DAPLEX (see section 1.2.4).

In [Borkin 80a] it is remarked that the simplicity of the Relational Model with respect to update operations is gained only at the expense of failing to represent important information concerning the application. The absence of inter-relational links from the Relational Model makes update hard, since the information about the connections between classes is important if the database is to be modified sensibly. The DBMS needs to know about the inter-relational connections if it is to assist the user in modifying the database and if it is to be able to fault him when he tries to make some inappropriate modification. Note that it is not sufficient merely to state that inter-relational links exist, it is also necessary to know both how modifications to tuples in a relation are constrained by inter-relational links and how tuples in other relations may be affected when tuples in a relation are modified. Sometimes data models express the fact that inter-class links exist, but not how the links interact with update.

Update is hard; in general a great deal of knowledge about the database is required if it is to be done correctly. The problems of update are often neglected by people proposing data models; it is often either ignored or dismissed by a few sentences towards the end of the paper. When describing the Relational Model Codd gives only a brief description of INSERT, DELETE and MODIFY, although in the description of RM/T he provides much more discussion of update, its problems and the rules which must be obeyed when performing modifications.

Knowledge of the connections between classes in the database is also important when constructing natural language interfaces to databases. The lack of such information can make the task of constructing the database difficult [Boguraev, Sparck Jones 83], forcing the interface either to be naive or to contain a description of the database in its own terms, a description which hopefully corresponds to the actual database.

1.2 Other Data Models

The Relational Model is not the only data model which has been investigated during the last ten years. Several of these other data models are discussed below to illustrate this point. Many of the other data models emphasise the connections between the different classes of objects within the database. These inter-class connections are represented naturally by a network-like description and can be expressed by referential constraints in the Relational Model.

Codd's extended Relational Model, RM/T, is not discussed explicitly, but the influence which other work had on RM/T is indicated in the text.

1.2.1 Abrial: Binary Relational Models

[Abrial 74] presents a data model based on the database being described as a set of categories together with a set of binary relations between the categories; Abrial's model is classified in [Tsichritzis, Lochovsky 82] as a binary data model. The two directions of the relation are labelled and are termed access functions (although they are not really functions since they can yield multiple values like the multi-valued functions of DAPLEX). There is also a facility for specifying the cardinality of the result that an access function can yield; hence if necessary, access functions can be made to be true functions. New categories can be generated to collect together information in other categories into aggregate objects (compare this with aggregation as described in [Smith, Smith 77a]). The data model can be described conveniently as a labelled graph.

Abrial also describes a programming language which is used to manipulate and interrogate the database and data model.

Abrial admits to being heavily influenced by knowledge representation formalisms (of which semantic networks were one of the earliest).

1.2.2 The Semantic Hierarchy Model

In 1977 Smith and Smith presented two proposals for extensions to the Relational Model [Smith, Smith 77a&b]. The extensions, aggregation and generalisation, form the basis of the Semantic Hierarchy Model which has been investigated further more recently in [Ridjanovic, Brodie 82]. Both aggregation and generalisation express information about the connections between the different classes in the database. Aggregation and generalisation are included in Codd's RM/T.

Aggregation is a process whereby larger meaningful units are synthesised from smaller ones. For example, a car may be considered to be an aggregate consisting of an engine and a chassis; note that both an engine and a chassis are objects which can exist independently of being parts of cars.

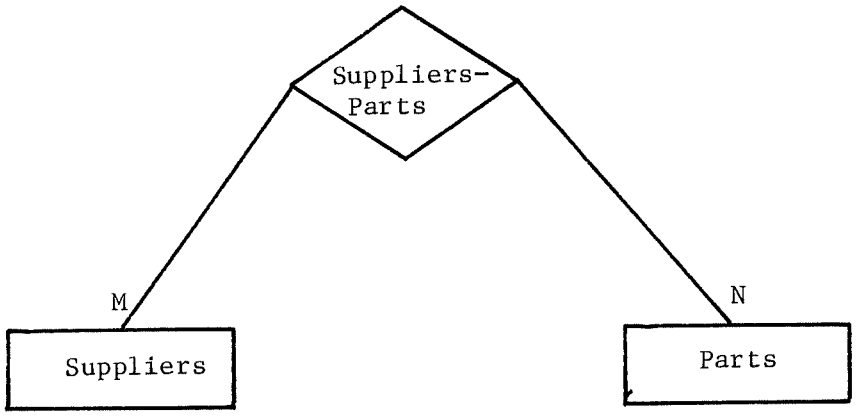
Generalisation is a process which allows hierarchies of classes to be constructed. For example, the classes secretaries, managers and electricians can be generalised to the class employees, which inherits the common properties of its sub-classes.

[Smith, Smith 76b] also presents the notion of a well-formed relation, which is a relation which can be named by a simple noun, i.e. represents a simple thing. A more useful definition may be that a relation be described by a simple sentence; this area was investigated by Borkin [Borkin 80a&b] in his Semantic Relation Model, which is based on the natural language processing notion of case grammars. This rule

is important since otherwise it is possible to construct relations which are collections of totally unrelated attributes; such relations will satisfy the conditions for being in third normal form (a purely formal test), but will nevertheless be totally meaningless.

1.2.3 The Entity-Relationship Model

The Entity-Relationship Model [Chen 76] is a popular and accepted tool for database design. The model is usually expressed as a labelled graph (entity-relationship diagram), this being the definition of the data model. The graph consists of a collection of entity sets linked via relationship sets; for example, the suppliers and parts database would be represented as:



In the above example, 'Suppliers' and 'Parts' are entity sets and 'Suppliers-Parts' is a relationship set. The labels 'M' and 'N' on the arcs denote the fact that the relationship between 'Suppliers' and 'Parts' ('Suppliers-Parts') is many to many (M to N).

Entity and relationship sets may also be linked to properties which are the value sets on which the attributes of entity and relationship sets are defined; properties are like domains in the Relational Model. Both entity and relationship sets may have properties. Examples of properties would be "Quantity-in-stock" associated with the relationship set 'Suppliers-Parts' and "City-of-business" associated with the entity set 'Suppliers'. Codd's RM/T inherits the ideas of entity and relationships (associations) from the Entity-Relationship Model.

For the purpose of describing the data model the data is thought of as being stored in a collection of relations (entity relations and relationship relations). [Chen 76] presents rules which govern the

checks which need to be performed when new entity or relationship tuples are inserted, updated or deleted. These rules require the DBMS to have access to the (graphical) definition of the data model.

1.2.4 DAPLEX

DAPLEX ([Shipman 81]) is a language which describes the 'Functional Data Model'. A DAPLEX database consists of a collection of classes and a collection of functions between classes. Functions in DAPLEX may be either single-valued or set-valued, and are referred to in DAPLEX as single-valued functions and multi-valued functions respectively. DAPLEX distinguishes syntactically between multi-valued and single-valued functions. For example, in the 'Suppliers-Parts' database there would be the classes 'Suppliers' and 'Parts' together with a function 'Supplies-Parts' which when applied to an instance of the class 'Suppliers' yields the set of parts which the supplier supplies; 'Supplies-Parts' is a multi-valued function.

DAPLEX does not contain a great deal which is new; many of the ideas which it contains have been around for some considerable time. However, DAPLEX is important since it brings together many ideas in a uniform framework and presents a neat, uniform syntax for expressing the data model.

DAPLEX functions specify the structural constraints of references between classes. Therefore, DAPLEX is a network data model in the same way as the Entity-Relationship Model and the Semantic Hierarchy Model are network models. In many ways DAPLEX is similar to the model proposed by Abrial, which is surprisingly not referenced in [Shipman 81].

In addition to the constraints which it captures implicitly in the schema definition, DAPLEX also provides a trigger mechanism for the enforcement of constraints together with a mechanism for the specification of explicit constraints.

[Shipman 81] provides little discussion of the interaction of the facilities to update the database with the functions in the database; indeed, details of the facilities provided for update are only found in in the appendix! This leads one to wonder if it was added as an afterthought.

One criticism which can be made of DAPLEX is that the paper stands well back from the problems of implementation, although Shipman does state that he sees DAPLEX partly as a design tool for database schemas and not, necessarily, as something to be implemented in its own right. Implementations have subsequently been produced (e.g. [Atkinson, Kulkarni 83]).

1.2.5 The Network Model (CODASYL)

The CODASYL model is often neglected or maligned by database researchers, This is possibly because it cannot point to a firm mathematical foundation or a high level query interface in the way that the Relational Model can, or possibly because it is firmly associated with COBOL (which is not a language close to the hearts of most Computer Scientists). Regardless of this CODASYL does form the basis of a considerable number of implemented and heavily used DBMSs.

CODASYL, unlike the Relational Model and like the other data models discussed in this section, provides explicit specification of the links between the different classes in the database (RECORD types). The links are specified by CODASYL set types which have owner and member record types. For a given instance of a CODASYL set type there is only one occurrence of a record of the owner record type. Note that although a record can only be a member of one instance of a given set type, it can be a member of instances of many different set types. The data model also provides a wide range of tests which can be made when a new member record is added to a set instance. Hence a CODASYL database consists of a network of connected records.

A detailed account of the data modelling facilities provided by CODASYL and a comparison with other data models can be found in [Tsichritzis, Lochovsky 82].

1.2.6 Semantic Networks

Database management is not the only area of Computer Science which deals with the problems of describing information. Another area is that of Artificial Intelligence, which has produced a number of formalisms for describing knowledge.

A tool which is widely used by the Artificial Intelligence community for representing knowledge is the Semantic Network. However, there is not a standard Semantic Network, rather there is a class of representational techniques which share features which can identify them as semantic networks. The characteristic features of semantic networks are that they are labelled graphs the nodes of which represent concepts or objects and the arcs of which represent relationships between the nodes. An arc between two nodes 'A' and 'B' often represents information such as "A is a property of B", "A is a part of B", "A is a generalisation of B" and "A is an instance of B". The previous sections have illustrated that data models represent similar information. The history, development and current status of semantic networks is discussed in [Brachman 79].

Semantic networks have in the past influenced database researchers; for example, Rousopolous recognised the descriptive power of semantic networks for database description and in [Rousopolous 77] he describes the production of relational database descriptions from a semantic network description of the database. Also, [Weber 76] uses a semantic network formalism to analyse constraints.

It is interesting to note that few papers on data model reference work on knowledge representation; however, recently there has been a realisation that there should be a greater interchange of ideas between researchers in the areas of Database, Artificial Intelligence and Programming Languages. A move in this direction was a workshop held at Pingree Park in 1980 ([Brodie, Zilles 80]). The workshop brought together workers from all three disciplines and although no firm conclusions were reached, there emerged a desire to continue talking.

1.2.7 Programming Languages

A recent, novel idea in databases is the idea of persistent programming ([Atkinson 78]). Persistent programming attempts to integrate the data and the programs which operate on the data within a consistent framework. A language which embodies the idea of persistent data is PS-Algol ([Atkinson, Chisholm, Cockshott 82]), which has been used successfully in a number of applications, including implementations of DAPLEX [Atkinson, Kulkarni 83] and RM/T [Hepp 83].

This approach to incorporating database management facilities into a programming language as a basic feature is different from the approach of embedding the data manipulation facilities for a particular data model into a programming language (e.g. SQL in PL/1 in System/R, DAPLEX in ADA to give ADAPLEX, and the addition of the data type "relation" to Pascal to give Pascal/R).

Although PS-Algol is not a data model, its data structures emphasise the connections between the data represented. PS-Algol also provides referential integrity; this is the nature of programming language record structures - in general pointers to things which do not exist are not allowed.

1.3 The Work to be Done

As has already been mentioned, this dissertation deals with the enforcement of three types of constraint in a relational DBMS.

The concept of a domain is central to the Relational Model. The first type of constraint dealt with is those implied by domains in the Relational Model. Although in the basic definition of a relation a

domain can be any set, it is conventional to restrict domains to be sets of atomic, that is nondecomposable, values; this is the requirement for first normal form, a requirement to which this work adheres. Given that the database is in first normal form, domains define the atomic data values which can be present in the database; the values in a domain are fixed and are independent of the content of the database. The definitions of domains impose constraints on the data values which can be present in the database. The enforcement of domain constraints helps to prevent erroneous data from being entered into the database. It must be remembered that the definitions of domains are part of the database, not part of the DBMS and that the definitions of domains should form part of the data model. The earlier part of the dissertation describes a technique for defining domains and for enforcing the constraints which they imply.

The second type of constraint which is dealt with is intra-tuple constraints. This class of constraint expresses constraints between the values in a tuple. The constraint may depend either only on the content of the tuple or on the relationship between the old value of the tuple and the new value (e.g. salaries should increase).

The final type of constraint dealt with is referential constraints. Section 1.1 discussed the problems posed by the lack of information about inter-class connections in the Relational Model; section 1.2 illustrated that description of the connections between the classes of data in a database is an important part of many data models. In the terms of the Relational Model referential constraints provide a way of expressing these inter-class connections. The majority of this dissertation is devoted to an investigation of referential constraints and techniques for enforcing them in a relational DBMS. The investigation shows how referential constraints can be used as building blocks from which generalisation hierarchies can be constructed and suggests how the notion of referential constraint can be extended.

The techniques which are used in this work to enforce referential constraints, were influenced heavily by the development of the relational DBMS CODD at Cambridge [King, Moody 83]. A basic feature of CODD is the way in which it uses networks of coroutines to evaluate database queries expressed as directed graphs of relational algebra operations. Since the connections between relations described by referential constraints can be expressed as a network, it seemed reasonable to investigate whether this coroutine technology could be applied to the maintenance of referential constraints.

Some of the work described has already been reported in [Robson 82a] and [Robson 82b].

1.4 Summary

Chapter 2 describes CODD, the DBMS which was used to test the ideas presented in the dissertation. This chapter introduces those aspects of CODD which are important for the work described in the rest of the dissertation. Although most of the detail in chapter 2 is not required until chapter 6, the material is presented as a coherent whole, rather than as sections scattered throughout the dissertation.

Chapter 3 deals with domains. It is shown how domains are defined to the DBMS, how the definitions are stored in the centralised data model, and how the definitions are used by CODD. Chapter 4 deals briefly with intra-tuple constraints.

The two subsequent chapters are devoted to referential constraints. Chapter 5 discusses the theory of referential constraints and suggests some extensions to the basic notion of referential constraints, showing how referential constraints can be used to build generalisation hierarchies. Chapter 6 is a detailed description of an implementation of referential constraints which uses the data pipelining provided by CODD.

Chapter 7 presents an example database based on the database used by the University of Cambridge Computing Service. The example is used to illustrate the part referential constraints play in the description of a database. Examples are given of the coroutine structures which are constructed for a number of operations on this database.

Finally, chapter 8 presents conclusions and suggestions for further work.

Chapter 2

THE STRUCTURE OF CODD

2.1 A Brief History

CODD (COroutine Driven Database) [King, Moody 83] is a DBMS originally developed by T.J. King as part of his Ph.D. research [T. King 79]. CODD is written entirely in BCPL [Richards, Whitby-Strevens 79]. The design of CODD was influenced very strongly by that of the Peterlee Relational Test Vehicle (PRTV) [Todd 76] which was developed at the IBM UK Scientific Centre, Peterlee. In the spirit of the PRTV, CODD evaluates relational algebra expressions by use of data pipelining techniques, i.e. queries are viewed as directed graphs the edges of which are pipes along which tuples pass between processing nodes. In the PRTV the graphs which could be evaluated were restricted to being trees, whereas in CODD they may be general acyclic directed graphs. By imposing an internal order on the tuples in relations and by ensuring that tuples pass along the pipes in this order, introducing sorts if necessary, relational operations can be implemented in an efficient manner. Usually the pipes contain either only one tuple (the one being passed to the node which will consume it) or the entire stream in the case where the tuples have to be sorted. This technique allows complex queries to be evaluated. During his research it was not uncommon for King to have queries involving several tens of joins, which is a degree of complexity that most other relational systems could not handle in an efficient manner.

Although as it was originally designed CODD had very powerful query evaluation facilities, it had no provision for ad hoc update. The only way of updating a relation was to copy the relation, adding new tuples by set union or removing them by set difference. This may seem to be a serious defect; however, the particular database on which King's research was based was effectively read-only, so the defect was not significant.

The DBMS used as a testbed for the work described in this thesis is a development of King's system and is a result of a redesign of CODD by Glauert, King and myself. The major aim of this redesign was to provide a convenient tool for the work of Glauert and myself, since it had become very difficult to modify CODD in any significant way. The major design aims were:

(a) to provide a portable system with well defined interfaces between the different parts of the system;

(b) to design a system which has a well defined model of the computation which it is performing; this allows an executing pipeline structure to be modified dynamically in a sensible manner;

(c) to make provision for proper update facilities, motivated by my desire to investigate the enforcement of constraints.

In order to achieve the aim (a) above the evaluation of a transaction/query in CODD is divided into several phases between which there are well defined interfaces. The phase structure allows the system to be overlaid in an obvious way on small machines (or even on large machines where it is wished to use as little space as possible for code when the transaction is evaluating). This clear functional division has been exploited to build a version of CODD which runs on several machines on a Cambridge Ring, the functions of the DBMS being distributed between the machines. This work was reported in [Robson, King, Glauert 81]. Details of the various phases of execution within CODD are given in section 2.6.

2.2 Pipelines in CODD

Pipelines are a basic feature of CODD. A query is viewed as a directed graph. The nodes of the graph represent the relational operations to be performed and the edges of the graph represent pipelines passing data between nodes. The nodes are implemented as coroutines [Moody, Richards 80] 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 and parallels can be drawn with lazy evaluation of functional programming languages.

The reasons for using coroutines for evaluating the query graph are given below.

(a) If queries are evaluated by recursive function calls on a 'tuple at a time' basis then it is necessary to have an auxiliary datastructure in which to save the state of operations between invocations. If coroutines are used then this is unnecessary since the state is saved as part of the coroutine representing an operation. Also the cost of reinitialising stack frames once per tuple is eliminated.

(b) Coroutines allow a richer set of evaluation structures than recursive function calls, since the graph to be evaluated is not restricted to being a tree; this is exploited by the COPY operation (see section 2.7.4 below).

(c) Coroutines allow a flexible control structure, (see section 2.6.2 below).

The coroutines which form an executing query graph know the identities of their producers and consumers. This knowledge allows a coroutine to pass information to its consumers and producers by resuming them. The coroutines communicate with one another by using a protocol which allows them to give and get tuples and to signal exception conditions such as 'no more data required' (consumer satisfied) or 'end of data' (producer exhausted).

The communication protocol, which is the work of J.R.W. Glauert [Glauert 81], also provides facilities for dynamically modifying the executing pipeline structure. One of the simplest operations of this type is to remove a set union node from the executing structure if one of its two producers becomes exhausted. A more complicated example of dynamic restructuring is attaching more coroutine structure to some which is already executing. Typically this is done when some data dependent test has been resolved. This can be used to implement a query which calculates the transitive closure of a relation. In order to be able to do this restructuring the DBMS maintains a model representing the computation in progress. A description of this structure and how it is used is given in section 2.6 below. The following sections also give definitions of the function of the pipeline nodes which are important in this thesis and describe how the evaluation of a transaction is managed.

It will be shown in chapter 6 that this technology, which was originally developed for dealing with conditional expressions in queries, and the rest of the pipeline technology is also useful when referential constraints are being maintained.

2.3 Database Structure and Integrity

CODD demands very little from the filing system of the machine on which it runs. It requires merely the ability to read and write blocks of a direct access file. It is felt that the experience of Stonebraker [Stonebraker 80] in using the UNIX filing system in the implementation INGRES vindicates only making minimal demands on the type of file access which the DBMS requires. Therefore, a CODD database is a direct access file, managed by the DBMS, consisting of a fixed number of fixed size pages.

2.3.1 Database Integrity: The State Page, Commit and Rollback

The scheme for database integrity, which gives the ability to commit and abort transactions, is based on recording the state of the database in the first page of the database (page-zero), and controlling the allocation of database pages in a pair of bit maps, the location of which are stored in page-zero. Page-zero also contains the address of the root of the value set system described in section 2.3.2 and the addresses of all of the catalogue relations. The state page is written back as an atomic operation which either commits a new database state or restores the database to the state at the beginning of the transaction. It is only at the point that a transaction commits that information about the old database state is lost. This mechanism provides indivisible updates and ensures that inversions (see section 2.3.3) will be maintained in step even over system crashes.

The two bit maps are termed OLD and NEW. At the start of a transaction these are both the same, and record the allocation of pages in the current, consistent database. OLD is never modified during a transaction since it defines the world which will be restored if the transaction aborts. NEW contains the state of the world which will exist if the transaction commits and pages which will be freed at the end of the transaction are marked free in NEW when they are logically freed. New pages are allocated by locating a page which is free in both OLD and NEW. This scheme relies upon:

(a) having sufficient secondary storage to have up to two copies of the database, although in practice it is usually possible to get away with less free space than this; and

(b) the storage mechanism freeing pages explicitly when they are no longer in use (in fact there is sufficient redundant information available in the database to perform garbage collection, but this should never be necessary).

In order to commit a transaction OLD is replaced by NEW, page-zero is modified to reflect this and page-zero is then written back to the database. Conversely, in order to abort (or rollback) a transaction NEW is replaced by OLD and page-zero is not modified.

2.3.2 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. Within the files the data is sorted according to an internal sorting order. This yields an efficient storage organisation, data items being located easily and efficiently by binary chop. The access methods for these indexed sequential files provide all of the normal file operations (INSERT, DELETE, ALTER, FIND, READ). This storage scheme represents an associative store accessed by data value. It is cleaner than a scheme using pointers, which introduce their own consistency problems.

When it is being manipulated a file is represented by a cursor which indicates the current position in a file in terms of the route through the index tree which needs to be followed to reach the current position. The route is recorded as a set of database page numbers and offsets on those pages at each stage. When a file is modified great care is taken to ensure that the index tree remains balanced, index levels being added and removed as tuples are inserted and deleted. When a file is initially loaded pages are not filled completely so that the first insertion does not result in major index reorganisation.

This file structure is very low level in that it consists of a set of fixed length records. The access methods do not place any interpretation on the types of the fields in the records.

Variable length data is stored in a hashed storage system based on the dynamic hashing scheme of Larson [Larson 78]. 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.

For both storage regimes the database page size determines the maximum size of the objects which can be stored.

2.3.3 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 chapter 6).

For the above reasons I added support for inversions to CODD. Relations are represented by a primary version (sorted on the key of the relation) plus a number of inversions. The inversions consist of a copy of the primary version which is sorted on the attributes forming the key of the inversion. The inversions contain all of the data fields of the primary version. All alterations to the relation are automatically performed for all of the files representing it. (The exception to this is, of course, when a new inversion is created and initialised.) When an inversion is defined it is automatically loaded with the relevant data sorted in the correct order. The integrity mechanism described above ensures that changes are reflected in all or none of the inversions of a relation.

If a relation has many data fields then this technique for storing inversions is quite expensive on disc space. However, it would not be much more difficult to maintain inversions of the form:

<inverted attributes, primary key of tuple>

This is a more conventional view of inversions.

When a relation is being manipulated during a database interaction it is represented by a datastructure called a LOCK. LOCKs have the following structure:

(a) the name of the relation;

(b) details of the attributes of the relation, i.e. column and domain names;

(c) the access mode (i.e. READ, WRITE, UPDATE);

(d) If the access mode is UPDATE the update operation, i.e. INSERT, DELETE or ALTER, which is to be performed;

(e) constraint information (see chapters 4, 5 and 6);

(f) a list of the inversions for this relation.

2.4 Resource Allocation: Pools

One of the problems produced by the lack of types in BCPL is that it is impossible to garbage collect the free storage heap. Therefore space which is allocated must be explicitly freed when it is no longer of use. For complex data-structures and scratch workspace this is somewhat tedious. CODD alleviates this problem by allocating resources, which include free storage, in pools. These are collections of resources which can be freed as single units or merged with other pools in order

to aid resource handling. For scratch space each PHASE has a pool associated with it which is freed at the end of the PHASE.

Free space is not the only resource which is allocated in a pool; coroutines, cursors and loaded code segments are all associated with pools. This makes it easier to tidy up allocated resources when it is necessary to recover from an error and restore the internal state of the DBMS to a known, tidy state. Although the use of BCPL to implement CODD forced this strategy on me, it would seem to be generally useful.

2.5 Catalogue Relations

The data model (both conceptual/semantic and storage/ internal) for a CODD database is held in a number of normalised catalogue relations.

This use of relations to store catalogue information has the advantage that it requires no special storage structure for the catalogues. However, I do not claim - as some authors have claimed - that representing the catalogues as relations enables the meta-data to be manipulated in the same way as the ordinary data. Although this is true if the operations performed on the meta-data are the same as those performed on the ordinary data, the meta-data is - in general - most often used in ways in which the ordinary data is not. In particular access to the meta-data usually has side effects which cause data structures for the control of the database transactions to be built or modified. The programs which produce these side effects are necessarily special purpose. Therefore, the convenience of not having to invent a new secondary storage regime is the only advantage of representing the meta-data as relations.

The structure of the catalogues is determined by the usual 'one fact one place' principle of database design, and the description of the physical storage of the data is separated from that of its logical structure. The information required for a basic relational database is stored in the relations 'Rnames', 'Inversions' and 'Domains'.

'Rnames' describes the logical structure of relations and contains:

(a) the name of the relation (this is the key of 'Rnames');

(b) the degree of the relation;

(c) a list of pairs (column name, domain name) encoded onto a VID;

(d) the key of the relation;

(e) the cardinality of the relation.

'Inversions', which describes the physical storage of relations has the following structure:

(a) a relation name;

(b) a permutation of the columns of the relation (encoded onto a VID);

(c) the database page number of the root of the file representing this inversion.

The relation name and the permutation form the key of 'Inversions'. For each relation in 'Rnames' there is at least one tuple in 'Inversions'.

'Domains' gives information about the domains which are present in the database. A complete description of 'Domains' will be given in chapter 3.

'Rnames' contains neither references to itself nor to the other catalogue relations, the locations of which are stored in special locations on page-zero of the database. This was a design decision and not an accident. The reason is that it is important that the user does not see the catalogue relations as ordinary data in his schema, since they represent the DBMS's world not the database's world. However, the user needs - and is provided with - facilities for querying the catalogues to discover what his data model is.

Apart from the ones described above, there are other catalogue relations which are used to represent other kinds of semantic information; for example, the relations 'RGX' and 'RGY' (described in chapter 6) are used to represent the reference graph required for the maintenance of referential constraints.

2.6 The Phases of Execution of a Transaction

The phases of evaluation of a typical transaction are given below.

(a) Perform the syntax analysis of the operations to be performed, producing a list of syntax trees with unbound relation and column names. For interactions which update the database the order of the list of syntax trees is significant and specifies the order in which the updates will applied to the database. The list of syntax trees forms part of the record which describes the transaction.

(b) The list of syntax trees produced in (a) is scanned to produce a list of the relations which will be accessed in this transaction. These relation names are looked up in the catalogues and a LOCK is constructed for each relation. A list

of the LOCKs created is kept as part of the record which describes this transaction. The information from the catalogues is used to bind relation names to the files in the database which represent the relations.

(c) The syntax trees are translated into the graph which is the template for the coroutine graph which will evaluate the expression (see section 2.6.1 below). However, it may not be possible to convert all of the syntax trees, since some of the expressions may contain data dependent tests which must be resolved before the conversion is done. Therefore, any such syntax trees are retained until sufficient information is available to enable them to be converted.

(d) Executable coroutine graphs are constructed from the template produced in (c). These are then initialised and executed, the template being retained as a model for the computation in progress. This step is repeated until there is no further structure to execute.

(e) If, at (c) there were some syntax trees which could not be converted, then go to (c) again and construct more executable graphs, since the data dependent tests should now have been resolved.

(f) If the content of database has been modified by the transaction, the catalogues are updated to reflect the changes. The list of LOCKs constructed at (b) is used to determine which relations have been modified and should therefore have their catalogue entries updated.

2.6.1 Construction of Executable Structure: The STAGE Graph

The execution model is an acyclic directed graph. The nodes of this graph are referred to as STAGEs and the graph is called the STAGE graph. A STAGE contains the information necessary to connect it to its neighbours. Each STAGE has a unique identifier which is used to identify it when the pipeline structure is being modified. These identifiers give the mapping between the executing pipeline structure and its STAGE model. Tests for the safety of the execution graph (see section 2.7.5) are performed on the STAGE graph.

While the STAGE graph is being constructed the links between stages are represented by arc-descriptors which contain both the semantic (domain description) and the physical descriptions (offsets within records) of the tuples which will flow along that arc when it is realised as a pipeline. The STAGE graph is constructed 'from the bottom upwards'; the initial sources of the information in the arc-descriptors

are the relations which are being read in the operation. The semantic information is pulled up the network and is modified as each node in the syntax tree is processed.

When syntax trees are converted into STAGE graphs, the validity of each operation in the syntax tree is checked, using the information contained in the arc-descriptors. These checks include checks on the degree of the inputs to a node, checks that the inputs to a node are 'union compatible', checks that column names referred to at a node are present in the inputs and checks that the constants supplied in selection expressions are of the correct type.

The coroutines which control the execution of a transaction (see below) always ensure that the stage model is an accurate model of the executing coroutine structure.

2.6.2 Query execution: SCHEDULER, EXECUTER and KILLER

SCHEDULER, EXECUTER and KILLER are coroutines which coordinate the execution of transactions. They provide the flexible control structure required for transactions which contain data dependent tests and hence require pipeline structure to be built dynamically. The functions provided by the three coordinating coroutines are:

SCHEDULER

This converts syntax trees into stage structures where this is possible. Otherwise the syntax trees are retained until it is possible either to build stage structures from them or to discard them.

Whenever SCHEDULER is invoked it tries to find syntax trees about which it has sufficient information to convert them into STAGE graphs. These stage graphs can then be passed to EXECUTER. If, after SCHEDULER has converted all of the syntax trees that it can at present, there are any remaining syntax trees to be processed then SCHEDULER will receive control when the currently executing coroutine graph completes execution; otherwise the EXECUTER receives control.

EXECUTER

This checks stage graphs for safety. If the graph is unsafe then it is made safe by the insertion of flexible buffers. After ensuring that the graph is safe (see section 2.7.5) EXECUTER builds and initialises the coroutine graph. EXECUTER is also called whenever a pipeline reorganisation might cause the graph to become unsafe.

KILLER

KILLER performs trivial pipeline reorganisations which do not affect the safety of the graph. If it decides that the operation which it has been requested to perform might make the graph unsafe it calls EXECUTER. The most common use of KILLER is to dismantle pipeline structure as it dies after it has completed execution.

2.7 A Pipeline Node Atlas

This section describes the pipeline nodes which will be referred to later in this dissertation.

2.7.1 BUFR and BUFW: The Terminal Producer and Consumer

Nodes in a query graph which have either no producers or no consumers attached to them are called terminal nodes. In order to provide a uniform treatment of these terminal nodes two special node types, the terminal producer (BUFR) and the terminal consumer (BUFW), are provided. Nodes of these types are attached to nodes which would otherwise be terminal, so that BUFW and BUFR are the only nodes which are terminal.

Given that pipeline structure may be created dynamically, BUFW nodes may be present for one of two reasons:

(a) as true terminal consumers which are never going to have any further structure attached to them;

(b) as 'savers', to which further structure will be attached later when some data dependent test has been resolved.

In case (b) there is always a WRITR node (see below) beneath the BUFW. The WRITR node saves the tuples which would otherwise have been passed directly to its consumer. If the structure attached as producer to the BUFW node completes execution before the test is resolved then the pipeline structure is dismantled and a record is kept of where the tuples written by the WRITR node have been placed, so that they can be read by a new READR node (see below) when the test has been resolved.

Another use of BUFW and BUFR is described in section 2.7.5 below.

2.7.2 READR: Read a Relation

This node reads a database file. It takes as part of its argument a LOCK representing a relation and reads from a specified inversion of the relation. The tuples read are determined by index requests supplied by the node's consumer; the tuple returned is the next tuple greater than or equal to the requested tuple. If the null tuple is supplied as index then the relation is read sequentially. The argument to READR also specifies whether the file read is to be kept or deleted after it has been read. If it is to be deleted then it is read destructively.

2.7.3 WRITR: Write a Relation

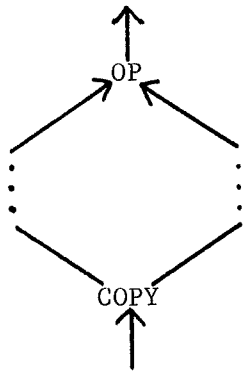
This node writes the tuples supplied by its producer to the end of a file. If the tuples are supplied in a sorted order then the file will be sorted, otherwise it will not be. When the node's producer signals 'end of data' the file which has been written is either closed and the address of its root saved or it is deleted. The above choice is made depending on the argument supplied to the WRITR. If the file is saved then the address of its root is recorded in a LOCK which is provided as part of the argument to the WRITR.

2.7.4 COPY: Produce Two Copies of a Stream

This node produces two copies of its input stream, passing a copy of each tuple that it receives to each of its consumers. It is this node which allows multiply connected query graphs to be evaluated by CODD.

2.7.5 Flexible Buffers

Given COPY it is possible for query graphs to fork and then join again. Consider the graph fragment:



COPY has to pass every tuple it receives to both its left-hand and

right-hand consumers. If the logic of OP is such that it requests many tuples from its left-hand producer before it requests any from its right-hand producer then the tuples passed by COPY to its right-hand consumer need to be stored somewhere until they are required. In the above form the graph is termed unsafe. It is made safe and the necessary storage is provided by the introduction of flexible buffers.

A flexible buffer provides storage for in principle an unlimited number of tuples. The stored tuples may be written to disc if there is insufficient room in store. In order to make an unsafe graph safe a flexible buffer is introduced along each of the arcs leading from a COPY node, if these will join again later. The introduction of flexible buffers is done before the coroutine graph is built.

A flexible buffer is a combination of a number of nodes, rather than a single node. The combination used is:

--> WRITR --> BUFW --> BUFR --> READR -->

The WRITR and READR nodes share a cursor on a database file; the WRITR writes the file and the READR reads it destructively. The BUFW and BUFR nodes provide the control necessary to decide what to do when either the consumer does not want the data yet or the producer has not produced some data when it is requested.

A flexible buffer may also be inserted into a query graph in order to produce a break in the graph. This may be necessary if, for example, there is insufficient space available for all of the coroutines required to perform an operation. In this case only the

--> WRITR --> BUFW

is built initially. The

BUFR --> READR -->

is added later when the first part of the structure has completed execution, thereby freeing the resources which it was using. This use of flexible buffers corresponds to writing the intermediate results of a computation to an anonymous database file, which is later read destructively in order to complete the computation.

2.7.6 COUNT: Count the Number of Tuples in a Stream

Apart from simply counting the number of tuples passing through it and reporting that number when the stream is exhausted, a quota of tuples may be set in the argument for COUNT which when it is satisfied will cause the SCHEDULER to be invoked. This may cause more structure to be built before control returns to the COUNT which provoked the action.

2.7.7 UPDATE: Modify a Relation

UPDATE differs from other pipeline nodes in that it has the side effect of modifying the database's contents. It is also different from most other pipeline nodes in that it can take an arbitrary number of inputs, the number of which is not fixed until the particular UPDATE node is invoked.

UPDATE takes as its input one or more streams of tuples. The output of UPDATE depends on the operation being performed by the UPDATE and is a stream of tuples. This stream consists of:

(a) if the operation is DELETE, the keys of the deleted tuples;

(b) if the operation is INSERT, the keys of the inserted tuples;

(c) if the operation is ALTER, tuples consisting of the old and the new key values for each altered tuple.

Chapter 3

DOMAINS

3.1 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. 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 [McLeod 76], 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, whether the domain is 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.

3.2 Implementation of Domains in CODD

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. The operations which the DBMS will allow on objects in a particular domain will depend on the operations available on the internal type used to represent the domain. Therefore, consideration of the properties of a base domain will influence the choice of the internal type for 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 names of the valid types and the operations which are available on them are currently bound into the code of the DBMS. This is not really satisfactory and if the system were to be redesigned then there would be a catalogue relation to store the information about types.

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
e.g.
CREATE DOMAIN numbers TYPE integer
```

Ideally programs such as 'numbers' should be part of the database (see section 3.2.1) 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 [Dewar, McCann 77]. The threaded code for an expression is essentially a post-fixed polish representation of the expression, with one extension described below. The interpreter for the code 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. In [Hammer, McLeod 80] Hammer and McLeod make this same point. They distinguish between the current population of values and the set of allowable values of a domain. Enumeration lists are an example of a set of allowable values whereas the projection of a relation on a particular column give the current population of the values in a domain. 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:

(a) the name of the domain - this is the key of the relation;

(b) the name of the underlying domain for this domain, with the convention that for base domains
'Domain name = Underlying domain name';

(c) the internal type for this domain;

(d) the predicate which differentiates this domain from its underlying domain.

Currently, the predicates which can be specified are restricted to combinations of selections of single values and ranges of values from the underlying domain. However, the definition language and the interpreter could be extended to allow more general predicates; for example, a predicate to test that data values satisfy a redundancy check.

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.

To summarise: the system described above provides facilities for defining the structure of domain elements and the relationships between different domains; it also provides checks on data values before they are entered into the database, thus ensuring a certain degree of database consistency. For these reasons - together with the ability to check the validity of operations such as join - I consider the system to be both useful and interesting.

3.2.1 Implementation of Domain Checkers

The procedures for recognising elements of base domains are arbitrary BCPL programs which take a character representation of a domain element and return TRUE or FALSE according to whether or not the value is acceptable. Associated with each checker there are programs for converting the value into the internal representation for the domain and for converting such values back into character form.

As I stated in section 3.2, these programs are logically part of the database rather than the DBMS, since the domains are application specific, and there are an unknown number of domains as far as the DBMS is concerned. The problem arises of how to make these programs accessible to the DBMS when it needs to use them. Clearly the domain checkers are going to be represented by separately compiled modules. Therefore, use must be made of the features available from the implementation language for linkage between separately compiled modules. In BCPL this linkage is performed via the global vector. BCPL systems provide facilities for dynamically loading code modules into the free storage area. When a module is loaded any global vector entries which are defined (as routines) in the module are reset. Facilities are also provided to reset and unset the globals associated with a loaded module, given a handle on the module.

Given the facilities outlined above, a simple way to implement the domain checkers would be to associate a set of global vector slots with each domain checker. However, since it is unknown how many base domains will be defined for a particular database, it is not possible simply to reserve a fixed number of global vector slots. A solution to this problem is to associate a single set of global vector slots with all of the domain checkers. After any of the domain checkers is loaded the contents of these global slots are immediately copied to a different place (typically to a record on the heap). This record is then used as the route via which all accesses to the routines comprising the domain checkers are made. The DBMS maintains a record of the domain checkers which are currently loaded. If a currently loaded checker is requested then, rather than loading a new copy of the code, the shared global vector slots are reinitialised and their contents copied in the usual way.

The DBMS thus has a uniform way in which it can use the domain checkers. To check a value for acceptability it uses the function ACCEPT.DOM which takes as one of its arguments the checker record, which it uses to access the correct code to check the value which it has been asked to accept. Similarly there are functions CONVERT.DOM and UNPACK.DOM which perform the transformations to and from the internal representation of the domain elements. The final function provided (DESCRIBE.DOM) gives a textual description of the structure of valid elements of the domain. For a derived domain the description is qualified by any restrictions which were placed on the domain when it was defined.

For recognising elements of derived domains the filter, which represents the predicate to be applied to the values in the base domain, is evaluated against the value to be checked after it has been converted into internal format. When a filter is defined (i.e. when a derived domain is defined) any constant values contained within the filter must be checked to see if they are elements of the underlying domain. If they are not then the filter is not a valid filter and the domain definition is rejected.

3.2.2 The Storage of Code in the Database

Since the domain checkers are part of a particular database rather than the DBMS their code should be stored in the database rather than somewhere else. In CODD a scheme to do this has been implemented. The scheme works by storing the compiled code, together with any relocation information, as a sequence of fixed length objects in the normal indexed sequential relational storage structure. The code can be easily loaded into this structure by just writing the file sequentially in the same way as a relation is loaded from a set of sorted tuples. The code is compiled separately before being loaded although in an ideal world the compiler ought to be part of the DBMS. The code stored in the database is then loaded as required by a component of the DBMS, the interface to which is similar to that for ordinary code loading. The indexed storage mechanism is used mainly to avoid having to invent a new storage regime within the database simply to be able to store code. Apart from this, using indexed sequential files has no advantage over any other storage structure, and indeed the information contained in the index of high and low values on pages is not of any practical use.

Being able to store code in the database provides an alternative method for storing domain predicates. This is to compile them into machine code rather than into the interpretable code which is currently used. Although I consider this approach to be worthy of investigation, I have not had time to do so.

I am indebted to Ken Moody for implementing the loading system used by CODD on the IBM machine at Cambridge. Clearly the code loading parts of CODD are machine dependent and would have to be modified when the system is transferred to another machine. However, from the experience of implementing it for an IBM/370 this is not a very difficult task.

3.3 Manipulating Domain Elements

It was stated in section 3.2 that the operations available on elements of a domain are those provided by the DBMS on objects of the internal type of that domain. Access to these operations is via the (interpreted) query language which knows what operations can be applied to each type. One thing which it is desirable to do is to restrict which operations on the internal type are available for a particular domain. For example: both the domain 'years' and the domain 'numbers' may be represented by the internal type 'integer'; while it makes sense to multiply two elements of 'numbers' it does not make sense to multiply two elements of 'years'. In order to remedy this problem, the operations which can performed on domain elements must form part of the domain definition. CODD does not provide this facility, although a simple system in which a list of valid operations is stored as part of the domain definition could be implemented relatively easily.

However, the real problem is the small number of internal types provided by the DBMS together with the difficulty of defining new types and the operations on them, and the difficulty of providing a suitable secondary storage representation for them. The first two problems are mainly a result of the lack of types in BCPL, although it is possible to provide a tagged typing system. The third problem would be solved by a language which incorporated the idea of persistent data, e.g. PS-Algol [Atkinson, Chisholm, Cockshott 82].

Chapter 4

INTRA-TUPLE CONSTRAINTS

This brief chapter discusses the maintenance of intra-tuple constraints. These are constraints between the values of the attributes of single tuples. These constraints can be divided into two classes.

(a) The first type of intra-tuple constraint is that which makes a statement about the static structure of tuples. For example, in an airline database the number of seats sold on a flight must not exceed one and a half times the number of seats available on the flight. Constraints like these must be tested whenever a new tuple is added to the relation or whenever a tuple in the relation is modified.

(b) The second type of intra-tuple constraint, which I will call dynamic, is tested only when a tuple is modified. This type of constraint is a condition which must hold between the new and old values of the tuple. An example of this may be found in a company database where salaries are not allowed to decrease.

Both types of constraint form part of the abstract specification of the relation and hence should be represented in the centralised data model. This is achieved by adding two new fields to the catalogue which describes relations. These new fields will contain descriptions of the static and dynamic intra-tuple constraints which apply to each relation. When the relations catalogue is accessed at the beginning of a transaction these descriptions are recovered and will form part of the argument to the UPDATE node which will modify the relation.

The conditions which can be specified in case (a) are exactly the same as those which can be specified in tests in the relational operation of selection. Indeed, the same syntax as is used for selection predicates may be used for static intra-tuple constraints. Dynamic intra-tuple constraints are different in that they require access to both the old and new values of the tuple. Therefore, the syntax needed to specify them will be slightly different. However, the predicates representing both types of constraint can be evaluated in the same way.

This is to use the same mechanism as is used for the evaluation of selection expressions and domain filters. The predicates are converted into a post-fixed polish representation, using information about the

physical structure of tuples to convert column names into offsets and sizes within a tuple. This 'compiled' predicate forms part of the argument to update and is interpreted against the actual tuples being inserted and modified in order to decide whether the update should be committed or aborted. In the relations catalogue the compiled predicates are stored, packed into value set identifiers, in the same way as domain filters are stored in the DOMAINS catalogue.

The reasons for using this interpretative approach to check these constraints are that:

(a) the interpreter was an existing, efficient component of CODD and was suitable for the job;

(b) while compiling the predicate the type checking necessary to decide if the predicate is valid can be done easily;

(c) the interpretable code is known to be safe, in that it cannot - as an arbitrary program could - have any undesirable side effects;

(d) the compiled predicate can be stored in the database easily.

Chapter 5

REFERENTIAL CONSTRAINTS: Theory

5.1 Introduction

In describing the real world two important abstractions are "X is an attribute of Y" and "X is a (specialisation of) Y". Use of these abstractions imposes two network structures on what is being modelled. If we have adopted the 'one fact one place' rule when designing a relational schema then these networks express inter-relational structure. It was noted in chapter 1 that one of the main defects of the relational model is its inability to express, and hence for its implementations to maintain, inter-relational structure. In this chapter it is shown how inter-relational references can be specified to the DBMS and represented in the database as part of a centralised data model. Chapter 6 shows how inter-relational references are maintained in a particular DBMS (CODD).

The two abstractions described above were first suggested, in the context of databases, by Smith and Smith [Smith, Smith 77b]. Since that time they have formed a part of every new data model that has been proposed (e.g. DAPLEX [Shipman 81], SHM+ [Ridjanovic, Brodie 82], SDM [Hammer, McLeod 81] and RM/T [Codd 79]). Although in their papers Smith and Smith describe how the two abstractions can be mapped onto the relational model, they do not give any indication of how such a system could be implemented.

Referential constraints have been identified by Date as being an important type of constraint, in that such constraints occur frequently in practice. They give the ability to specify inter-relational links and to ensure that that these links are maintained. Although Date stresses that referential constraints are only a special case of the general constraint problem, it will be shown in this chapter that these constraints can be used as building blocks to specify more complex inter-relational structures. This chapter also describes the relationship between referential constraints and other proposals for doing similar things, giving the advantages and disadvantages of each scheme.

Below I describe Date's basic notion of referential integrity and suggest some extensions. The purpose of the extensions is to increase the usefulness of referential constraints in a natural way by enabling

them to capture more semantic information. Finally I show how sets of referential constraints can be used to construct generalisation hierarchies.

5.2 Definitions

In order to show how the idea of referential integrity has been refined, I start with the definition of referential integrity as proposed by Codd as part of his extended relational model, RM/T. Codd's definition of a referential integrity is:

> Suppose an attribute A of a multi-attribute primary key of a relation R is defined on a primary domain D. Then, at all times, for each value V of A in R, there must exist a base relation (say S) with a simple primary key (say B) such that V occurs as a value of B in S.

In the above definition a primary domain is a domain on which some relation has a single attribute primary key defined. It could be noted in passing that primary domains are very much like the surrogates of Todd [Hall, Owlett, Todd 76]; however, unlike surrogates they are seen by the user and not hidden from him.

Below, 'R' will be termed the referencing relation and 'S' the referenced relation. Note that the definition does not insist that there is only a single referenced relation for a constraint, since different tuples of the referencing relation may reference different relations. Date [Date 81a] has modified Codd's definition for the theoretical and practical reasons set out below.

(a) The definition relies on the notion of primary domain and does not allow references to relations with multi-attribute primary keys. Although it is always possible to invent a single attribute primary key for a relation, this restriction does not really seem to be reasonable.

(b) The definition asserts merely that a referenced relation must exist but does not define which relation it is. In particular, if the referencing attributes are the key of the referencing relation then the constraint is trivially satisfied. However, we may wish to insist that there is a distinct referenced relation.

(c) The definition requires that the referencing attribute is a component of the primary key of the referencing relation. This seems rather too restrictive for a large number of applications, especially when the constraint is being used to define an 'attribute of' relationship between the entities represented by tuples in different relations (see the

'Is managed by' example in section 5.4).

(d) The definition gives no indication as to what action should be taken when tuples in referenced relations are deleted and altered.

For the above reasons, Date produced a modified definition for referential constraints; I have adopted this definition as the basis for the work described in this chapter.

Date's definition of a referential constraint is:

> There is a referential constraint between two relations R1 and R2 (which are not necessarily distinct) 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. Indeed the joins suggested by the referential constraints in the database are those which do not suffer from the connection trap ([Date 81b] p9), i.e. they yield information about what is currently true in the world of the database rather than what might be true.

When a relation references several others as part of the same constraint a quantifier may be specified. This quantifier indicates whether referenced tuples must exist in ALL OF, 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. This part of the constraint definition is in response to objection (d) above to Codd's original scheme. 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.

UPDATE CASCADES seems to be a purely practical concession, in that the key of an object should not change in an ideal world. However, in practice it is sometimes necessary to change the value of the key of a relation and UPDATE CASCADES at least ensures that all references to it are changed as well. For this reason, if any referential constraint for which a relation is the referenced relation has UPDATE CASCADES specified, then all references to the relation should have UPDATE CASCADES specified (otherwise the meaning of updating the relation would be very strange). In all other cases UPDATE RESTRICTED should be specified.

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. It will be shown below how, if we regard relations as representing entities, careful specification of the update and deletion rules for constraints can associate particular semantic interpretations for the entity classes represented by relations.

For the rest of this dissertation the syntax which is described in figure 5.1 will be used for the definition of referential constraints and relations.

All the referential constraints specified for a particular database form a reference graph. This reference graph forms part of the schema of the database and, in keeping with Codd's proposals in RM/T for having the schema data represented in the same way as the 'real' data, Date suggests that the reference graph could be stored in two (normalised) relations. 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 reference graph is recovered by performing the natural join on constraint name between these two relations. Note that there is a referential constraint between these two relations, namely:

```
Ref-graph-con:
   RGY.[Constraint name] ->> RGX.[Constraint name]
                                  DELETION cascades
```

Referential Constraints

<constraint-name> : <referencing-part> ->> <referenced-part>
<deletion-rule> <update-rule>

where:

<referencing-part> ::= <ref>
<referenced-part> ::= <ref>
<referenced-part> ::= <quantifier> (<ref>, ... <ref>)
<ref> ::= <relation-name> . <attribute-set>
<deletion-rule> ::= DELETION <rule>
<update-rule> ::= UPDATE <rule>
<quantifier> is one of ALL OF, SOME OF, EXACTLY ONE OF
<rule> is one of RESTRICTED, CASCADES or NULLIFIES

<attribute-set> denotes an ordered list of column names enclosed in square parentheses, e.g. [col1] or [col1,col2]. The attribute sets in the referencing and referenced parts of the constraint definition must agree elementwise on domain.
<update-rule>, <deletion-rule> and <quantifier> are optional.

Relations

<relation name> : [attribute, ...| attribute, ..., attribute]

The attributes to the left of the '|' are the key of the relation being defined.

Figure 5.1: Definition of Referential Constraints and Relations

---

In Date's proposal all referential constraints are named. This is largely a result of the above representation of the reference graph (which I have adopted). It should however be noted that this naming is not strictly necessary since a system could generate (internal) names for the constraints so that the graph could be stored in 'RGX' and 'RGY'. Further, in some cases the naming of constraints is not even desirable, since it results in a name explosion and a difficulty in choosing names or a need to invent unnecessary names; see the examples in section 5.4.

Referential constraints are tested at the end of transactions, rather than at the time when tuples are inserted. This is because, if the reference graph contains any cycles, it may be necessary to insert several tuples independently in order that the final database state will satisfy the constraints. Therefore, it must be possible for the database to pass through inconsistent states within a transaction. For example, consider the database given below:

```
R1: [A | B ]
R2: [X | Y ]

C1: R1.[B] ->> R2.[X]
C2: R2.[Y] ->> R1.[A]
```

Now, suppose that the tuple (1,2) is to be inserted into R1 but the tuple with key in R2 does not at present exist. If the required tuple in R2 would be (2,1) then it is not possible to insert this tuple into R2 unless then tuple the (1,2) has been inserted into R1, which is what we were trying to do at first!

Therefore, at some point the database will have to pass through an inconsistent state. This means that the constraints cannot be tested at the time at which the update is performed. Hence, referential constraints are checked at the end of transactions after all changes have been made to the database.

At the end of a transaction if any referential constraint is not satisfied then the transaction is aborted and a message is produced to tell the user which constraints were violated by which tuples.

5.3 Checking Constraints

There are three situations which call for constraints to be checked.

(a) Whenever a tuple is inserted into or altered in a relation which is the referencing relation for a referential constraint.

(b) Whenever tuples are deleted from or altered in a relation which is the referenced relation for a constraint which has the deletion or update rule RESTRICTED. The reason for this test is that otherwise it would be possible for the constraint to be violated because the referred to tuple no longer exists.

(c) Whenever a tuple is inserted into or modified in a relation which is the referenced relation for a constraint which has the quantifier EXACTLY ONE OF and for which there is more than one referenced relation. If this check is not made then the constraint could be violated by having a referred to tuple in more than one of the referenced relations

In all three cases the checking of the constraint must be deferred until the end of the transaction, otherwise the the information which was used to make the decision of whether or not the constraint was satisfied may be altered later in the transaction.

5.4 Examples of Referential Constraints

The examples given below give some idea of the usefulness of referential constraints in specifying database structure. They illustrate the use of referential constraints to express several different types of semantic information in a natural way.

Referential constraints may be used to specify that the tuples in one relation must be a subset of the tuples in another (subset constraints).

```
Employees: [Emp-id | ... ]
Managers:  [Emp-id | Department, ... ]
C1 : Managers.[Emp-id] ->> Employees.[Emp-id]
```

This specifies that the class of managers is a subset of the class of employees. Such constraints have referencing attributes which are the key of the referencing relation.

Referential constraints may also express the fact that tuples of a relation represent associations between tuples of other relations.

```
Students: [Name | ... ]
Projects: [Project | .... ]
Assignments: [Student, Project | Start-date ]
Assign1 : Assignments.[Student] ->> Student.[Name]
          DELETION CASCADES
Assign2 : Assignments.[Project] ->> Projects.[Project]
          DELETION CASCADES
```

This example specifies that an assignment is an association between a student and a project. The specific deletion rules which are specified ensure that if either the student or the project are removed from the database then all referencing assignments are deleted as well. Note, however, that the names given to the constraints are contrived. This is an instance of the problem, described above, of insisting that all referential constraints must be named.

Referential constraints may be used to express the fact that an entity represented by a tuple in one relation is an attribute of an entity represented by a tuple in another relation.

```
Employees: [Emp-id | Manager ]
Managers : [Emp-id | ... ]
Is-Managed-By: Employees.[Manager] ->> Managers.[Emp-id]
```

In this example, tuples in 'Managers' are regarded as attributes of tuples in 'Employees'. This relationship is like a functional dependency between two relations, and indeed it might represent a functional dependency which was lost during a normalisation process.

If for some reason the entity class which the constraint needs to reference is partitioned then there will be several referenced relations. Consider for example a library which divides its catalogue into fiction and non-fiction.

```
Loans: [ Bookid | Borrower ]
Fiction : [ Bookid | ... ]
Non-fiction : [ Bookid | ... ]

Is-borrowed :
      Loans.[Bookid] ->> ( Fiction.[Bookid],
                           Non-fiction.[Bookid] )
```

There is of course rather more to the above example since in the full schema something would be said about the relationship between fiction books and non-fiction books (i.e. that the two sub-classes are disjoint).

As was stated above the existence of cyclic reference graphs means that constraints cannot be checked until the end of the transaction. The following example illustrates how such structures might occur.

```
Books : [ Bookid | ... ]
Non-fiction : [ Bookid | ... ]
Fiction : [ Bookid | ... ]

Is-Book1 : Books.[Bookid] ->> EXACTLY ONE OF (
                                 Fiction.[Bookid],
                                 Non-fiction.[Bookid] )
Is-Book2 : Fiction.[Bookid] ->> Books.[Bookid]
Is-Book3 : Non-fiction.[Bookid] ->> Books.[Bookid]
```

Here a new book cannot be added to the database unless it is added to the superclass 'Books' and one of the subclasses 'Fiction' and 'Non-fiction' at the same time, if the constraints are not to be violated at some point during the transaction. The example illustrates the use of referential constraints to specify an IS-A hierarchy. More will be said about this type of structure in section 5.9.

Another example of a cyclic graph is found when information which is represented naturally by an N-ary relation is decomposed into several binary relations, and all of the properties of the entity must be present. For example, consider the relation X : [K | A, B]. 'X' can be decomposed into the two binary relations:

```
X1 : [ K | A ]
X2 : [ K | B ]
```

together with the two constraints

```
C1 : X1.[K] ->> X2.[K] DELETION CASCADES
C2 : X2.[K] ->> X1.[K] DELETION CASCADES
```

5.5 The Form of the Reference Graph

The reference graph which describes the referential constraints in a database can be a general directed graph. However, there is a problem if the shape of the reference graph is not restricted. The potential problem arises with reference graphs which fork and then join again. The problem is that if two streams of cascaded modifications 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. Such a dependency on ordering would be a bad thing. Note that the conflict only occurs if the referencing attributes for the constraints producing the cascaded streams overlap.

This problem is noted in the paper on triggers in SEQUEL [Eswaran 76], where the view taken is that the user must be very careful that the situation does not arise.

Having stated that this problem may occur formally, I have been unable to construct any convincing examples to illustrate it. This suggests that the problem may not arise in practice. Possibly, and this is merely a conjecture, the existence in a schema of the structures leading to this problem is an indication of some basic fault in the schema, i.e. the set of semantically meaningful reference graphs is a subset of the formally possible reference graphs.

Given that the problem relates to graphs which fork and then join again, it is reasonable to ask if such graphs ever occur in practice. The answer to this question is yes; consider the following simple example.

```
Friendship : [ Person1, Person2 | ]
People     : [ Name | ... ]
F1 : Friendship.[Person1] ->> People.[Name]
                  UPDATE CASCADES
F2 : Friendship.[Person2] ->> People.[Name]
                  UPDATE CASCADES
```

In this example the problem does not occur since the referencing attributes for the two constraints do not overlap.

5.6 How is the Cascading of Updates and Deletions Organised?

Date suggests that before any cascaded operations (and he includes in these the testing of RESTRICTED links) are performed, a check is made to determine whether or not the operations would succeed. This approach seems to confuse the two distinct problems of semantic database integrity and the ability to back out from a transaction which fails. Also the scheme suggested by Date would fail on the following example:

```
x: [ a | b ]
Con : x.[b] ->> x.[a] DELETION CASCADES
```

What happens when the tuple (1,1) is deleted?
If we check that the change is valid before doing the deletion then an infinite loop is produced, which continually checks to see if it is valid to delete the tuple (1,1). This could be avoided if we check if the tuple has already been cascaded before entering it into consideration for the next level of cascades. However, this would involve the implementation in quite a lot of housekeeping.

A better (and correct) approach is simply to perform the operations required by the reference graph. If any of these fail in a way which requires the transaction to be aborted then the database integrity system should be allowed to perform the abortion. This is the approach which has been taken in the implementation of referential constraints described in chapter 6.

5.7 Comparison with other work

5.7.1 Aggregation

Smith and Smith [Smith, Smith 77a] define aggregation as an abstraction which allows a relationship between named objects to be thought of as a higher level named object. This is the "X is a subpart of Y" construction. Their main reason for investigating this form of abstraction is a desire to add further semantic structure to the relational model by introducing some inter-relational structure. As an example of aggregation, the relationship between a student and a project may be abstracted as the aggregate object 'assignment'. This is precisely the type of structure which referential constraints allow us to express (see the examples above). Therefore, the enforcement of referential constraints is necessary in a relational system which supports aggregation.

In describing aggregation Smith and Smith also propose a set of rules for governing the insertion, deletion and modification of tuples in relations which represent aggregates. The rules which they suggest are not as extensive as those in the proposal above. In particular, they do not have the important concept of cascaded operations. Instead the referencing tuples must be explicitly deleted or modified as required. This makes transactions for modifying a relation more complex. Referential constraints, as defined above, therefore seem to be more expressive than aggregation, in that more semantic structure can be specified. However, the technique of constructing aggregates is important for building the set of relations on which referential constraints are imposed.

5.7.2 Addis's Extended Relational Analysis

In [Addis 82] Addis describes what he calls 'extended relational analysis' (in his terminology 'relational analysis' is the process by which a relational schema is decomposed into a set of relations in third normal form). The main extension which he proposes is the construction of an implication network which records those (inter-relational) functional dependencies which would otherwise be lost during the normalisation process. The links specified by the implication graph are those which would be specified by referential constraints.

Addis makes the statement that joins which do not correspond to the links in the implication network represent information about what is possible rather than what is a fact in the world being modelled. He uses this to illustrate the connection trap which is documented elsewhere (e.g. [Date 81b] p9). This observation supports the statement that referential constraints defining the meaningful joins in the database.

When considering what should be done to tuples in a referencing relation when tuples which they reference are deleted, Addis considers cascading to be the normal case. However, he admits that cascading is sometimes inappropriate and hence, although he does not consider the problem in detail, admits the need for RESTRICTED referential constraints in some circumstances. He does not provide a discussion of the effects of modification.

Addis also does not allow cycles in his implication graph. In his paper Date explicitly includes these in his proposal, and an example of the need for such structures has been given above. The advantage of not allowing such structures is that it is then possible to define an ordering on the application of updates which keeps the implication graph satisfied and allows the constraints to be checked at the time that tuples are inserted rather than at the end of the transaction. However, Addis's proposal goes further than Date's in some directions, one of which, cardinality constraints, is discussed in section 5.8.2 below.

5.7.3 Triggers and Assertions in SEQUEL

SQL (formerly SEQUEL), which is the query language for System/R [Chamberlin, et al 76] contains the idea of triggers [Eswaran 76], which are arbitrary SQL programs to be executed when a particular trigger condition is raised.

Trigger conditions can be quite general. They may be specified to be raised either before or after an update operation is applied to (some subset of the columns of) a tuple in a relation. Trigger conditions may also be raised after some particular database state occurs, e.g. when the number of employees in a department reaches a certain threshold.

The program associated with a trigger is called the trigger procedure. This program is executed immediately that the trigger condition is raised and the action of the trigger procedure may cause other triggers to be invoked.

Triggers could be used to model the cascade graph needed to maintain referential constraints under deletion and alteration. However, since the trigger procedure is executed immediately that the trigger condition is raised, the basic checking of referential constraints cannot be done using triggers - since the basic checking must be done at the end of the transaction.

However, SQL provides a second mechanism for constraint specification the tests for which can be delayed until the end of the transaction. This mechanism is the use of assertions. Assertions can be used to specify that the correct set theoretic relationship holds between projections of the referencing and referenced relations, hence giving the effect of referential constraints. However, this way of expressing referential constraints does not make the network structure produced by references clear. Therefore I regard this technique as being inferior to the way of expressing referential constraints which was described above.

In some ways triggers are a much more powerful concept than referential constraints, since the actions that may be specified when a trigger condition is raised may be arbitrarily complex. However, referential constraints give a clearer overall view of the semantics which are being enforced, and since referential constraints are restricted in the actions which they can perform when tuples are modified or deleted, they are much more amenable to efficient implementation. Indeed, triggers were dropped from later versions of SQL because they could not be implemented efficiently.

5.7.4 Codd's Extended Relational Model

Referential integrity is an integral part of Codd's RM/T; comparisons with this scheme have already been made in section 5.1.

In RM/T, Codd uses relations to represent his data model. It is interesting to note that although there are a number of referential constraints between the relations which form the data model, these constraints cannot be expressed in terms of Codd's own referential integrity proposal, since they require explicit identification of the referencing and referenced relations.

5.7.5 Functions in DAPLEX

Referential constraints represent functions from the tuples of one relation to the tuples of another relation. This is exactly what is provided by the single-valued, non-optional functions of DAPLEX. Therefore, referential constraints are important if DAPLEX is to be implemented on top of relational systems, since we need to ensure that non-optional functions will return values for new entities of the class to which the function applies.

The syntax used in DAPLEX for implicitly defining referential constraints is better than that proposed by Date, especially since it does not require the naming of the constraints. However, DAPLEX does not have a direct analogue of the update and deletion rules but, like SQL, it has the concept of a general trigger. The comments made about the SQL triggers apply equally to these DAPLEX triggers.

5.7.6 CODASYL

It has been suggested above that referential constraints allow a network structure to be imposed on a set of relations. Therefore, it is natural to ask how the facilities provided by referential constraints compare with those provided in CODASYL systems. The answer is that CODASYL provides most of the facilities of referential constraints, by means of a mixture of membership class, explicit checks and different set types. With these tools simple referential constraints, except those with the combinations (CASCADES, NULLIFIES), (NULLIFIES, NULLIFIES) and (RESTRICTED, NULLIFIES) of deletion and update rules, can be dealt with (see [Date 81a]). However, a more serious drawback is that CODASYL does not provide a way to handle constraints where several relations are (potentially) referenced. This is due to the fact that CODASYL does not allow sets having "alternative owners", which is what this type of constraint would require.

CODASYL also provides a trigger mechanism which predates that described in [Eswaran 76].

5.8 Extensions to the Idea of Referential Constraints

As was stated in the introduction, referential constraints are a special case of constraints. However, there are a number of quite simple ways in which the definition of referential constraints can be extended, allowing more semantic information to be specified. Both extensions described below are, I believe, useful and make the database a little more flexible.

5.8.1 Replacement of DELETION NULLIFIES

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 the query which produces the "default" value yields a single tuple.

Indeed the rule for obtaining the default value is a function which will determine an element of the referenced relation. Note also that there will be a referential constraint between the referenced relation and the relation from which this value is obtained. In the example above the relation giving the senior tutors for colleges will also reference the tutors relation (as does the students relation).

This type of construction is similar to the concept of class-determined properties which is present in Hammer and McLeod's SDM. It is also very like the idea of association suggested by Brodie [Brodie 81]. In the example the name of a senior tutor for a college would be associated with the set of tutors for that college.

In [Date 82] Date suggests that NULL values should not be allowed in databases, but that default values should be specified instead. However, Date's definition of default is simpler than that described above.

5.8.2 References to non-key attributes

The definition of referential constraints 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'; i.e. the function described by the referential constraint is ONTO the set of lecturers.

The need for the extra constraint is a consequence of the fact that referential constraints provide a way of expressing M(>=0) to 1 links. However, what is often required is M(>0) to 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) to 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. The general point is, of course, that I may wish to place constraints on the number of references to a particular tuple in the referenced relation.

This extension is a special case of the cardinality constraints of Addis [Addis 82]. Addis allows, for a relation which references several others, arbitrary predicates to be specified on the cardinality of the links between tuples in the referenced relations and those in the referencing relation. One of his examples is that the number of students in a fencing class must be even and in the range [6,30].

Another example of where cardinality constraints would be useful is in Computer Aided Design (CAD). Consider a geometrical model of a solid object. This is represented by a number of faces which intersect at edges, and there is the constraint that every edge has exactly two faces associated with it. The inability of relational systems to capture such constraints is one reason that it is difficult to use relational DBMSs in CAD applications.

These predicates can clearly be violated at intermediate points in the execution of a transaction. For example, in the example of the fencing class new students have to be inserted in twos; this may be achieved by two separate insertions at different times during the transaction, and until the second tuple is inserted then the cardinality constraint is violated. Therefore, cardinality constraints, like referential constraints, must be checked at the end of the transaction.

Given that Addis has cardinality constraints, he effectively has cyclic implication graphs since the cardinality constraints produce backward implications. It is difficult, therefore, to see why Addis explicitly states in his paper that implication graphs may not contain cycles.

It is interesting to note that these backward references are like inverse many-valued functions in DAPLEX.

Cardinality constraints may also require a more sophisticated failure action than referential constraints, e.g. in the fencing class a single student wishing to join the course should be placed in a 'waiting list' relation which is consulted when other students wish to enrol in the course. Clearly this behaviour can be modelled by triggers as in SQL.

As a consequence of cardinality constraints, Addis is led to introduce forward deletions. I do not believe that these can be done automatically (in general) since a choice needs to be made as to which tuple to delete. One solution to this (which is the solution adopted by Addis) is to perform modifications to the database conversationally. This is similar to the approach used by Sharman in Update-by-Dialogue [Sharman 77].

5.9 Generalisation

Generalisation is the name given by Smith and Smith [Smith, Smith 77b] to the IS-A hierarchy in knowledge representation. It is an important data modelling tool given that it reflects a type of structure which is often found in the real world. Often it is more convenient to regard the abstraction as being specialisation, for example both lecturers and students are clearly specialisations of people. Here the specialisation is with respect to the role that they play in a particular application, namely an educational institution. This may be regarded as a specialisation according to the value of a particular attribute 'person-type', which in effect partitions the class of people according to whether they are lecturers or students. Note that a particular person cannot in this scheme be both a lecturer and a student, which in some cases may be regarded as being quite sad.

Types may be specialised independently in more than one way and each subtype can itself be specialised. Therefore, an acyclic directed graph of types can be constructed.

5.9.1 Generalisation and Referential Constraints

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 define part of a generalisation graph, 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 (e.g. replace EXACTLY ONE OF in 'Gen' by SOME OF). Note that in the Smiths' scheme the subtypes are always disjoint. This is because, in their scheme, the link between the supertype relation and the subtype relations is via the value of the attribute over which the generalisation was performed in the supertype relation. Note also 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, for the reasons given below.

(a) it does not make clear that a particular set of referential constraints represent a generalisation;

(b) 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 naming of the constraints is, as was stated in section 5.2, largely a result of the constraint descriptions being stored in the database as a pair of relations.

(c) There is also the problem that such a representation does not specify the property with respect to which the specialisation has been made. This is an important part of the definition of a specialisation. However, it does not require much additional machinery to solve this problem. One possible solution is to label each edge between the sub- and super-types with the value of the attribute on which the specialisation has been made and to label the constraint node(s) which control the specialisation with the name of this attribute.

However, the fact that the structure implied by generalisation hierarchies can be represented and maintained 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. Further as point (c) above has demonstrated the specification of generalisation hierarchies requires slightly more information than the set of referential constraints themselves provide. Given that referential constraints can be maintained, it is possible to maintain generalisation hierarchies

provided that a suitable reference graph can be constructed from the catalogue relations which represent the generalisations.

A suitable set of catalogue relations is:

Supertype : [Name, Specialisation Attribute |
Partition/Cover]

Subtype : [Name, Supertype, Specialisation Attribute |
Specialisation Attribute Value]

where there is a tuple in 'Supertype' 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, i.e. whether the referential constraint would have the quantifier EXACTLY ONE OF or SOME OF.

From the tuples of these two relations the required reference graph 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.

5.10 Summary

A proposal for the specification of the important class of structural constraints, namely referential constraints, has been presented. It has been shown how these constraints can be used to specify some of the network structure of the data held in a relational database, thus enabling some of the inter-relational semantics to be captured.

Date suggested that referential constraints are an important special case of the constraints which might be enforced. It has been shown that they provide a building block from which other database structures, in particular generalisation hierarchies, can be constructed.

Chapter 6

REFERENTIAL CONSTRAINTS: Implementation

This chapter describes the implementation of referential constraints in CODD. It shows how the pipeline technology of CODD is used to express the structures necessary to maintain these constraints.

The first part of the chapter describes the implementation of the scheme used to maintain the constraints, giving details of the storage and recovery of the reference graph and of the pipeline structures built to maintain the constraints. The second part describes various optimisations which might be made.

6.1 Basic Strategy

The implementation of referential constraints has two parts:

(a) the checking of the constraints when tuples are inserted into (or altered in) a relation; and

(b) the propagation of cascaded deletions and alterations.

The testing for UPDATE and DELETION RESTRICTED is included in (a).

In order to check constraints it is necessary to:

(a) determine the constraints which need to be tested when any relation is modified;

(b) keep a record of the tuples for which the constraints have to be checked.

The constraints which need to be checked and the cascades which have to be performed are determined from the reference graph.

For every relation which is explicitly updated there is a fragment of the reference graph which determines cascaded deletions and alterations. This part of the reference graph is termed the cascade graph. The method used to deal with the propagation of cascaded deletions and alterations is based on the simple observation that the cascade graph can be regarded as a pipeline structure in the same way as a query graph can be regarded as a pipeline structure. The basic idea is to build a

processing node for each relation in the cascade graph. These processing nodes are then connected by pipelines so that the final coroutine structure models the cascade graph. The process just described is easy to do for cascade graphs which are simple trees; however, more care is needed when the cascade graph is more complex.

For constraints which have to be checked, the program which modifies tuples in relations can make a record of tuples which are inserted or altered. This record can then be read at the end of the transaction by the constraint checker to check that the constraints are satisfied.

The following sections deal in detail with the maintenance of referential constraints in CODD. They show how the reference graph is stored and fragments of it recovered, how the coroutine structure to maintain the constraints is built and evaluated and how the constraints are checked.

6.2 Catalogue Representation of the Reference Graph

In order to maintain referential constraints some representation of them is required. In a CODD database the constraints are stored in two relations, the reference graph relations, as proposed by Date. These relations, which were described in section 5.2, are:

```
RGX: [ Constraint name | Referencing relation name,
       Referencing attributes, Quantifier,
       Update rule, Deletion rule ]

RGY: [ Constraint name, Referenced relation name |
       Referenced Attributes ]
```

The constraint name is used as the join link between the two catalogue relations in order to construct the cascade graph. This is a lossless join since there is in fact the referential constraint:

```
Ref-graph-con:
   RGY.[Constraint name] ->> RGX.[Constraint name]
```

between RGX and RGY.

Within the catalogues RGX and RGY have the structure:

RGX:

Constraint name	(Value set id.)	- KEY
Referencing relation name	(Value set id.)	
Referencing attributes	(Value set id.)	
Quantifier	(Integer)	
Update Rule	(Integer)	
Deletion Rule	(Integer)	

RGY:

Constraint name	(Value set id.)	|- KEY
Referenced relation name	(Value set id.)	|
Referenced attributes	(Value set id.)	

The attributes 'Referencing relation name' and 'Referenced relation name' are lists of column names held as value set identifiers.

When constructing the cascade graph, which is an in-store data structure, from the catalogues, it is necessary to access RGX by both 'Constraint name' and 'Referencing relation name', and to access RGY by both 'Constraint name' and 'Referenced relation name'. Therefore, for efficiency of access there are two inversions of each of RGX and RGY. These are sorted with the following columns as the first two fields and have the names shown:

For RGX: (Constraint name, Referencing relation name) - RGXC
(Referencing relation name, Constraint name) - RGXR

For RGY: (Constraint name, Referenced relation name) - RGYC
(Referenced relation name, Constraint name) - RGYR

These inversions permit easy access to the catalogues by both 'Constraint name' and 'Referencing/Referenced relation name'. Both of these are important when recovering the cascade graph for a particular operation on a relation.

It should be noted that this is not the only possible way of storing the reference graph; it is merely a convenient way to do it. The alternative would be to have some special purpose scheme for storing graphs, but this was rejected because the work involved in building such a subsystem would have been considerable.

6.3 Recovery of Cascade Graph from the Catalogues

For each relation which appears in an INSERT, DELETE or ALTER command it is necessary recover the part of the reference graph in order to maintain any referential constraints applying to the relation. It is also clear that, if possible, this fragment should be pruned in such a

way that it contains only that information needed to maintain the constraints applying to this relation. The most striking example of the need for pruning is if the relation being modified is not itself referenced but references other relations which in turn reference other relations which in turn reference other relations ...; clearly in this case it is not desirable to recover the whole hierarchy, the first level alone is sufficient.

The information in RGX and RGY is used to build an in-store representation of the reference graph applying to a relation. The data structure which is built 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; this includes information on which inversions are available.

(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 the relevant constraint node

  (2) referencing attributes

(e) for each constraint for which this relation is the referenced relation a pointer to the relevant constraint node.

The structure of a constraint node is:

(a) the name of the constraint

(b) pointer to the referencing relation node

(c) number of referenced relation nodes

(d) For each referenced relation node:

  (1) pointer to the referenced relation node

  (2) correspondence between the referencing attributes and the key of the referenced relation

(e) update rule

(f) deletion rule

(g) quantifier.

The cascade graph is used in two ways; firstly it is used as a template for the coroutine structure which will be used to control the cascading of alterations and deletions. Secondly it is used by the constraint checker to determine the relations, attributes and quantifiers involved for constraints which have to be tested.

The construction of the graph representing the cascades which are to be performed is terminated either by UPDATE/DELETION RESTRICTED or when the referencing relation is not itself referenced. 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.

The cascade graph for a relation is different depending on whether the operation being performed on the relation is INSERT or DELETE or ALTER; the basic algorithms used to construct the cascade graph in each case are given in section 6.11. Note that while the cascade graph is being constructed, the constraint nodes are marked according to whether:

(a) they need to be tested when tuples in the referencing relation are modified (TO-BE-TESTED);

(b) they will be maintained by cascding but need to be tested the first time this constraint node is processed (CASCADE-AND-TEST);

(c) they will be dealt with as part of the cascade structure (BY-CASCADE).

Note that (c) includes constraints which are included in the graph because there is a RESTRICTED link between the referenced and referencing relations. Case (b) occurs when a relation, which is a referenced relation of a constraint which has quantifier EXACTLY ONE OF and update rule CASCADES, is ALTERed. Note is taken on the marking of constraint nodes when constructing the coroutine structure which will execute the operation.

When constructing the cascade graph it is important to recognise cycles in the graph. There are two reasons for this.

(a) If there are cycles which are not noticed then the algorithms which recover the graph fragment will loop forever.

(b) The knowledge of where the cycles occur is important when evaluating cyclic cascade structures.

For these reasons a list of the places at which cuts must be made in the graph in order to avoid cycles is constructed, at the same time as the fragment is being constructed. This list is termed the CUTLIST.

The rules for the construction of the CUTLIST are only of interest when tuples are being ALTERed or DELETEd, since INSERT does not produce cascades. Cycles are detected by associating with each relation node along a cascade path the list of its predecessors. Places where cuts need to be made are then easily detected. Before adding a relation node to the graph a check is made to see whether the relation is in the predecessor list for the current relation. If it is then a cycle is being constructed. The following actions are then taken:

(a) add the new constraint node, connecting it to the existing referencing and referenced relation nodes as normal;

(b) do not add the referencing relation node to the list of relations to be processed, even if this would otherwise have been required;

(c) add to the CUTLIST an entry for the two relation nodes and the constraint node.

In the case of ALTER slightly more care has to be taken since the constraint node connecting the two relations in a cycle may already be present as a constraint which had to be tested. A further point is that constraints of this type require a third type of mark for constraint nodes; this mark indicates that for the first invocation of the cycle they are to be tested, but for subsequent invocations they will be maintained automatically.

There is a special case for both DELETE and ALTER which it is important to get correct. This is when a relation references itself.

Care must also be taken with graphs which fork and then join again. These can be recognised when the referencing relation node is already present in the graph fragment but when the edge to be added will not produce a cycle (remember that the cascade graph is a directed graph). In this case the constraint node is also connected to the existing referencing relation node, and we must check that the substructure associated with this node has not already been added before proceeding to add it.

Although the construction of the cascade graph for ALTER, INSERT and DELETE have been treated separately, in the implementation advantage is taken of their common features in order to provide a single function which will recover the cascade graph for a relation.

The structure which is returned to represent the cascade graph therefore has two components:

(a) the graph itself, which consists of a collection of linked relation and constraint nodes;

(b) a list of cuts (usually empty) which indicates where the cycles in the cascade graph are.

This graph is used as a template for the coroutine structure which will perform any cascading and testing of RESTRICTED links together with the checking of constraints. How the CUTLIST is used to control the evaluation of cyclic cascade structures is described in section 6.6.

The relations for which cascade graphs have to be constructed are determined by analysis of the syntax trees describing the transaction being processed. A list of locks for relations which are accessed by the complete transaction is also constructed. This list is used to control the updating of the RNAMES catalogue at the end of the transaction and to discover the database addresses of the inversions of relations. As cascade graphs are created new locks may be added to the lock list and the access which is required to relations described by existing locks may change from READ to READWRITE, if the relation was to be read anyway is now to be modified.

6.4 Checking the Constraints

Section 5.3 described the situations in which constraints need to be checked. The three cases are:

(a) Whenever a tuple is inserted into or altered in a relation which is the referencing relation for a referential constraint.

(b) Whenever tuples are deleted from or altered in a relation which is the referenced relation for a constraint which has the deletion or update rule RESTRICTED.

(c) Whenever a tuple is inserted into or modified in a relation which is the referenced relation for a constraint which has the quantifier EXACTLY ONE OF and for which there is more than one referenced relation.

In each case information required to check the constraint is obtained from the cascade graph. This information, together with the modified tuple, is used to check the constraint. The tests which need to be made are:

(1) For (a). The value referencing attributes of the tuple inserted or the new values of the tuple modified must be used to try and find a tuple with matching key in the referenced relation(s) and the quantifier specified in the constraint node must be applied.

(2) For (b). The key of the deleted tuple or old key of the altered tuple must be used to try and find a tuple with that key in the referencing relation.

(3) For (c). If a referencing tuple exists for the inserted tuple or new value of the altered tuple then the constraint must be tested. It is not possible merely to check that the new value is already referenced since the referencing tuple may have been inserted in this transaction.

Constraints of type (1) are easy to test since they only require access to relations by primary key and this is quick. However, in order to check the constraints described in (2) and (3) it is necessary to have a secondary index structure. It would be prohibitively expensive to search the referencing relation every time that a link from referenced relation to referencing relation needs to be followed. The secondary index structures required are provided by inversions. The inversions allow tuples with given referencing attribute values to be located quickly. The information in the constraint node is used to determine which inversion should be used to access the referencing relation. The identity of this inversion can then be passed to the program which checks the constraints.

The definition of a referential constraint causes the automatic creation of an inversion sorted on the referencing attributes of the constraint, if such an inversion does not already exist.

Although inversions confer the advantage of quick access to relations by the referencing attributes of constraints, inversions incur overheads.

(a) The space taken in the database is increased. Currently the space taken is increased by the amount of space occupied by a single copy of the data in the relation for every constraint for which a relation is the referencing relation. However, there is no additional copy made if the referencing attributes are the key of the relation or if the referencing attributes are the same as those of another constraint.

For relations of large degree which have few fields which participate in referential constraints the current form of inversions is wasteful on disc storage. For such cases inversions of the form:

<referencing attributes, primary key value>

would be more efficient in the use of disc space. The existing scheme is used because it could be implemented easily in the experimental system.

(b) The time taken to modify a relation is increased. For a relation the increase is linear in the number of inversions maintained for the relation.

The constraint checker takes as input a list of tuples and a list of constraints to be tested against the tuples. The constraints are identified by a constraint node together with a tag which indicates the type of constraint to be checked. This list of tuples to be checked is produced by the UPDATE node which modified the relation.

Although in theory the constraints cannot be checked until the end of the transaction, in practice it is often possible to check constraints earlier. To understand this consider how a transaction is processed.

A transaction is an ordered sequence of operations which are committed or aborted as a unit. In CODD a transaction is presented as a sequence of operations which the DBMS is told to evaluate; for example:

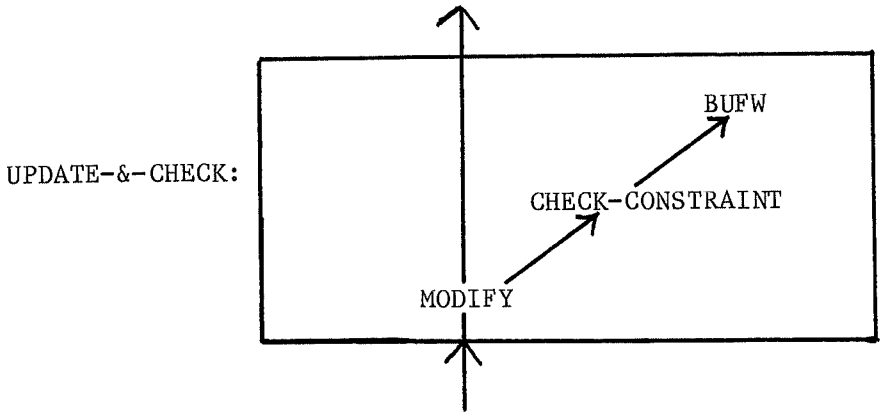
```
insert ... into RelationA
delete ... from RelationB
insert ... into RelationC
DoIt
```

For each operation in the transaction it is possible to determine which relations the operation can modify. This can be done by analysing the cascade graph for the operations. This information can be used to determine whether relations needed in an earlier operation for checking

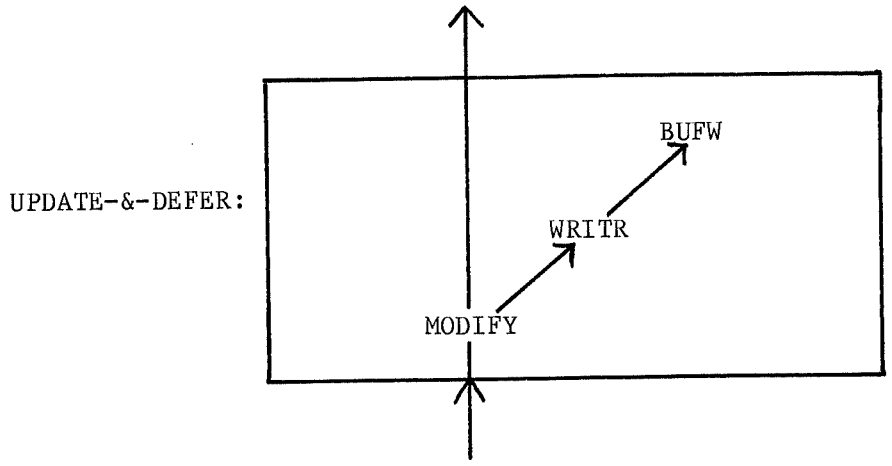
constraints could be modified by subsequent operations in the transaction. If a relation which needs to be read to test a constraint could not possibly be modified by the current operation or by a later operation, then the constraint can be tested immediately, rather than waiting to the end of the transaction. The simplest example in which this optimisation is useful is where the transaction consists of inserting tuples into a single relation.

Therefore, UPDATE may be considered to have two alternative internal structures, one of which is used when constraints can be tested immediately and the other of which is used when checking has to be deferred.

For constraints which can be checked immediately the structure is:



For constraints for which checking must be deferred the structure is:



When the checking of the constraint has to be deferred a record is kept of which constraints have to be checked against the tuples written by the WRITR. The file containing these tuples is deleted after the constraints have been checked.

If constraints are violated then error messages of the form:

- o Insertion of [...] into RelA failed since it would violate Con1
- o Deletion of [...] from RelA failed since it is still referenced by [...] in RelB and Con2 specifies DELETION RESTRICTED
- o Insertion of [...] into RelA failed this would cause [...] in RelB to reference more than one tuple which is forbidden by Con3

can be produced. Note that there is sufficient information in a constraint node and its associated relation nodes to spell out the constraint in full rather than just giving its name.

The precise action which is taken when a constraint is violated depends on whether or not the checking had been deferred. For a constraint which is checked immediately then the transaction is aborted immediately since it is going to fail anyway. However, if constraints are being checked at the end of a transaction the transaction is not aborted as soon as an error is found. Instead all of the constraints are checked and errors reported before the transaction is finally aborted. This action is taken since at this stage not to check all of the tuples would waste some of the work already done and the cost of checking a tuple against a constraint is small.

6.5 Construction of the Coroutine Structures

The cascade graph for a relation which is to be updated is used as the template from which to build the STAGE structure for the operation. The STAGE structure is a template for the coroutine graph which will execute the operation.

Before the way in which the STAGE structure is built is described, the types of processing which are required by nodes in the cascade graph will be considered. There are two types of node in the cascade graph; those that require modifications to be performed and those which require constraints to be checked. Nodes causing relations to be modified may require one of the following five operations to be performed.

(a) Insert - Each input tuple is inserted into the relation being modified by the UPDATE. The tuples must not exist already in the relation otherwise an error is raised.

(b) Delete - Each input tuple is deleted from the relation being modified. The tuples must exist in the relation otherwise an error is raised. This operation is used for the first stage of a deletion cascade.

(c) Delete-Set - Each input tuple gives a value of the referencing attributes for a constraint. Every tuple in the relation being modified with referencing attributes which match the input value are deleted. This operation is used for the later stages of a deletion cascade.

(d) Alter - Each input tuple gives the old and new values of the tuple to be modified. The old value must exist in the relation. This operation is used for the first stage of an update cascade.

(e) Alter-Set - Each input tuple gives an old and new value of the referencing attributes for a constraint. Every tuple in the relation being modified with referencing attributes which match the old value are modified. This operation is used for the later stages of an update cascade.

One of these operations is specified to every UPDATE which is built.

Whether a constraint is tested immediately or not is determined by looking at the relations modified by this operation and subsequent operations in the transaction. If the test can be performed immediately then UPDATE is initialised in its UPDATE-&-CHECK form. Otherwise the UPDATE is initialised in its UPDATE-&-DEFER form and a record kept for the constraint checker of where the tuples to be tested have been written and what the constraints to be tested are.

When an UPDATE-&-DEFER node is initialised a record is created which contains:

(a) the identifier of a file containing the list of tuples which which will have to be tested; this is written by the WRITR component of UPDATE-&-DEFER;

(b) a list of the constraints which are to be tested against the tuples in the file.

This record is shared between the UPDATE-&-DEFER and the constraint checker which is invoked at the end of the transaction.

The translation of the cascade graph into a STAGE structure proceeds by way of an ink-blot algorithm, starting from the relation which is to be modified.

The processing begins as follows.

(a) An UPDATE node is built for the relation which is to be modified initially. The operation requested for the UPDATE node is 'Insert', 'Delete' or 'Alter' one tuple per input tuple, as appropriate. The argument to the UPDATE node contains a list of those constraint nodes which are attached to

this relation node, and which were marked TO-BE-TESTED when the cascade graph was built or which reference this relation and have the update/deletion rule RESTRICTED if the tuples are to be altered or deleted. The UPDATE node is informed whether it can test the constraints immediately or whether it needs to defer testing until the end of the transaction. This process gives the structure:

Producing structure --> UPDATE(Operation)

The bracketed term after the UPDATE is the operation which the UPDATE is to perform on its input.

(b) If the relation node is not referenced or if it is only referenced by a RESTRICTED link then the STAGE structure is terminated by adding a BUFW node to the output of the UPDATE node. This gives the structure:

Producing structure --> UPDATE(Operation) --> BUFW

If (b) has been performed, the STAGE structure needed to perform the update is now complete. Otherwise, a set of links consisting of:

- o Referencing relation node
- o Constraint node
- o Referenced relation node
- o Producing stage
- o Output number of producing stage

is constructed, one for each reference to the current relation. The 'Producing stage' is either the UPDATE node constructed for the current relation or a COPY node. It may be a COPY node since, if a node requires several outputs then an appropriate number of interconnected COPY nodes are added to its output, in order to generate the required number of outputs. The outputs of this tree of COPY nodes are then used as producers for the consumers of the original UPDATE node.

The list of links is then processed as follows.

(a) If the link is a cut then attach the structure:

--> WRITR --> BUFW

to the producing stage. This pair of nodes is, in fact, one half of a flexible buffer, see section 2.7.5. A record is

kept, for use by the SCHEDULER, of the cut and the LOCK associated with the WRITR node. The LOCK both identifies where the cascaded tuples have been written and contains a count of the number of tuples written. This LOCK will be passed to the

BUFR --> READR

pipeline structure which will be constructed to consume the tuples written by the WRITR. The reading structure will not be constructed if no tuples have been written by the WRITR, since there is no point in building executable structure which will do no work. The cut is recorded to identify the relation node from which new structure is to be built if any tuples are cascaded.

(b) If the update or deletion rule (as appropriate) associated with the constraint node in the link is RESTRICTED, then the UPDATE node is passed the information needed either to test the constraint immediately or to defer the checking, as appropriate.

(c) If the update or deletion rule is CASCADES then the structure:

--> UPDATE(Alter-Set, or Delete-Set) -->

is built. Any constraint nodes which are referenced from the relation being modified and which are marked 'TO-BE-TESTED' are passed as part of the argument to the UPDATE node. Links are produced for any references to this relation and are appended to the list of links.

Steps (a) to (c) are repeated until the list of links is empty. CODD does not support NULL values and therefore NULLIFIES is not supported. However, it would be processed similarly to CASCADES.

If the cascade graph forks and joins again then sometimes a STAGE will have already been built to deal with a relation node when another of its producers is being processed. Therefore, when a relation node has been processed a record is kept of the STAGE associated with it. Before a node is added for a relation, a check is made to see if a STAGE already exists for that relation. If it does then the STAGE representing the node is modified so that it has an extra input. At this time a check is made that the operations requested for each input are the same. If they are not then the result of evaluating the graph may depend on the order in which the inputs to this STAGE are processed (see section 5.5); therefore, an error message is produced if this structure is encountered.

When structure is being built above a cut the list of links consists initially of the single link containing:

- o Referencing relation node of cut
- o Constraint node of cut
- o Referenced relation of cut
- o READR node of the BUFR --> READR pipeline used to read the previously cascaded tuples
- o 0 - the only output of the READR

The algorithm expressed in (a) to (c) is then used, starting with this list.

Cascaded operations use the inversions which are present for checking constraints. The inversions allow the tuples with given referencing attribute values to be located quickly. The information in the constraint node is used to calculate which inversion UPDATE node should use to check its input tuples. The identity of this inversion is part of the argument passed to the UPDATE node.

Examples of the coroutine graphs built for a variety of operations on an example database are presented in section 7.5.

6.5.1 An extension

The scheme described above does not make use of the ability to build pipelines dynamically. However, it is much easier to implement than the more sophisticated scheme in which the structure built below a cut is:

--> COUNT --> WRITR --> BUFW

As soon as a tuple passes through this COUNT the SCHEDULER will be invoked. The SCHEDULER will construct the next level of cascade structure above the BUFW giving

---> COUNT --> WRITR --> BUFW --> BUFR --> READR -->

When the single tuple written by the WRITR has been consumed by the READR then the set of nodes illustrated above can be removed from the executing structure. This scheme, which has not been implemented, has the side effect that several UPDATE nodes may be operating on the same relation at the same time. Therefore, it is important that they share the same cursors on the relation and that the relation is not closed until the last of the UPDATE nodes has completed, otherwise some of the modifications would be lost.

6.6 Evaluation of Cascade Structures

Exactly how the coroutine structure which organises the cascading executes depends on whether the cascade graph is acyclic or cyclic.

6.6.1 Acyclic graphs

This case includes simple trees and graphs which fork and then join again. However, if the graph forks and joins the graph 'optimiser' in CODD will insert flexible buffers along the forked arcs so that the synchronisation problems discussed in section 2.7.5 are avoided.

To execute these cascade structures the coroutines are built according to the template provided by the STAGE structure described in section 6.5. The coroutine graph is then simply initialised and runs to completion, performing all of the cascading required.

6.6.2 Cyclic graphs

In this case only the coroutines corresponding to the first cycle of the cascade can be built initially. When this structure has completed control is returned to the SCHEDULER which checks to see if any cascaded tuples have been produced from the cycle. This is done by looking at the cardinality in the locks associated with the WRITR nodes which are constructed for a set of cuts. If the number of tuples is zero for every cut then the computation terminates successfully. Otherwise, coroutine structure is built for the next cycle of every cut with a non-zero count. This results in a new set of cuts being produced, which will be tested when all of the new structure has been executed.

6.7 The Definition of Constraints

6.7.1 Definition of Constraints to the DBMS

The syntax used to specify referential constraints to CODD is that used for the examples in this thesis. When a constraint is defined it is checked for consistency 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.

6.7.2 Dynamic Definition of Referential Constraints

It has been tacitly assumed above that the data model is a static object. This will, in general, not be true since it must be possible to create new constraints and delete old ones as the data model and the database evolve.

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. For example, given the database:

```
Projects : [ Project | ... ]
Students : [ Student | ... ]
Assignments : [ Student, Project | ... ]

Assign1 : Assignments.[Project] ->> Projects.[Project]
                                    DELETION CASCADES
Assign2 : Assignments.[Student] ->> Students.[Student]
                                    DELETION CASCADES
```

the definition of the constraint 'Assign1' would cause an inversion of 'Assignments' sorted on 'Project' to be created. This involves sorting the file being inverted and writing the resulting file sequentially. This process is achieved by setting up a suitable pipeline structure and using the standard CODD sort function to perform the sort. The inverted file is then used as input to the constraint checker which checks the constraints in the usual way. The constraint checker is told not to delete the file after checking the constraints; this is different from the normal process of constraint checking.

This way of defining a new constraint and creating any inversions required is safe because the creation of the inversions and the modifications to the catalogues to define the new constraint and the inversions are all committed in the same checkpoint.

6.8 Implementation of the Extensions Suggested in Section 5.8

The following sections describe how the extensions described in section 5.8 could be implemented.

6.8.1 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 may be in the form of a relational algebra expression. Such expressions can be represented by prefixed polish strings. These can then be stored in the value set, 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. When the value which this query yields is required, it can be evaluated in the same way as any other query. The value which is produced can then be assigned to the relevant attributes of the tuple being updated.

If the rule to obtain the value is simple, e.g. a selection of a tuple from another relation, then rather than evaluate the relational expression (which would require a coroutine graph to be built and executed), it would be possible just to perform a simple lookup.

6.8.2 References to non-key attributes

The solution to this problem can be illustrated by considering the constraint 'Exists-Lecturer' in the database:

```
Courses: [ Name | Lecturer, ... ]
Lecturers: [ Name | Department, Date appointed, ... ]

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

Suppose that there is also the constraint that each lecturer must give at least one course.

The test which is required in order to maintain this constraint is that if a tuple is deleted from 'Courses', then we must check at the end of the transaction that the lecturer for that course still teaches some course. The check is performed at the end of the transaction for the same reason that referential constraints are checked at the end of a transaction, that is that the final course for a lecturer may be deleted and a new one created in the same transaction. The constraint checker can be easily modified to deal with this type of constraint, since the inversion required to perform the existence test already exists to facilitate the cascading of tuples and the testing of RESTRICTED links.

In order to represent this constraint in the data model the catalogues which define the reference graph will have to be extended so that RGY contains a field which states whether all the tuples in the referenced relation need to be referenced or not.

It is easy to see how to extend this scheme so that it can maintain arbitrary constraints on the cardinality of a simple link. General cardinality constraints as proposed by Addis [Addis 82] would, however, increase the complexity of constraint checking significantly.

6.9 A Different Way of Maintaining Inversions

The streams between UPDATE nodes are currently unsorted. For small numbers of tuples in the streams this does not matter, especially since the same UPDATE node maintains all of the inversions of a relation. If the number of tuples involved is large, however, it is probably worth sorting the streams so that the modifications to each inversion are ordered. However, in order to exploit the sorted streams it is necessary to maintain each inversion of a relation by a separate UPDATE node. This can be regarded as an extension of the cascade mechanism which has been described above. In this scheme only the UPDATE node which was maintaining the inversion keyed on the referencing attributes would be connected to the rest of the cascade structure.

The maintenance of inversions by separate UPDATE nodes would also be a great advantage for initial data load, since then each inversion can be loaded by simply writing it sequentially, hence reducing disc transfers. Here there is a tradeoff between the cost of sorting the input and then writing the file sequentially as opposed to performing random insertions into the file. For large files the latter method will result in increased disc traffic if the whole file cannot be held in store.

6.10 Conclusions

An interesting feature of this implementation is the way in which the coroutine graphs are used to interpret the data model. The users of the PRTV discovered that pipelining is a good strategy for evaluating queries against a database. The scheme described in this chapter for maintaining referential constraints extends the use of data pipelining techniques into the area of updating the database.

6.11 Algorithms for constructing cascade graphs

This section describes the algorithms used to recover cascade graphs. Detailed understanding of these algorithms is not necessary to follow the dissertation, but they are presented here for the sake of completeness.

The algorithms are presented as routines in pidgin BCPL. '//' introduces a comment and the comments at the head of the first routine define the environment in which the programs operate.

6.11.1 Recovery of Reference Graph Fragment for INSERT

```
//
//  add-relation( Relation )
//       - Adds a relation node for 'Relation' to the cascade
//         graph (if such a node does not already exist)
//
//  add-constraint( Constraint, How-maintained )
//       - Adds a constraint node for 'Constraint' to the cascade graph.
//         'How-maintained' indicates whether the constraint
//         is to be tested (TO-BE-TESTED), or whether it will be
//         maintained by cascading (BY-CASCADE), or whether
//         it will be maintained partly by cascading but also
//         needs to be tested (CASCADE-AND-TEST).
// add-referencing-link( Relation, Constraint )
//       - Adds a link from a referencing relation to a constraint
// add-referenced-link( Constraint, Relation )
//       - Adds a link from a constraint to a referenced relation
//
//  The following routines obtain information from the catalogues:
//
//  referencing-constraints( Relation )
//       - Returns the set of constraints which reference
//         'Relation' (i.e. constraints for which 'Relation'
//         is the referenced relation)
//  referenced-constraints( Relation )
//       - Returns the set of constraints which are referenced by
//         'Relation' (i.e. constraints for which 'Relation' is the
//         referencing relation)
//  referencing-relation( Constraint )
//       - Returns the referencing relation for 'Constraint'
//  referenced-relations( Constraint )
//       - Returns the set of referenced relations for 'Constraint'
//

LET process-insert( relation ) BE
$( add-relation( relation )
   FOR EACH Con IN referenced-constraints( relation ) DO
   $( add-constraint( Con, TO-BE-TESTED )
      add-referencing-link( relation, Con )
      FOR EACH Rel IN referenced-relations( Con )
      $( add relation( Rel )
         add-referenced-link( Con, Rel )
      $)
   $)
//
//     Deal with the case where constraints need to be checked -
//     see section 5.3.
//
   FOR EACH Con IN referencing-constraints( relation )
   $( IF ( quantifier( Con ) = EXACTLY ONE OF ) &
         ( number-of-referenced-relations( Con ) > 1 ) THEN
      $( LET referencing-rel = referencing-relation( Con )
         add-constraint( Con, BY-CASCADE )
         add-relation( referencing-rel )
         add-referencing-link( referencing-rel, Con )
         add-referenced-link( Con, relation )
      $)
   $)
$)
```

6.11.2 Recovery of Reference Graph Fragment for ALTER

```
LET process-alter( relation ) BE
$( LET to-be-processed = NULL
   add-relation( relation )
   FOR EACH Con IN referenced-constraints( relation ) DO
   $( add-constraint( Con, TO-BE-TESTED )
      add-referencing-link( relation, Con )
      FOR EACH Rel IN referenced-relations( Con )
      $( add relation( Rel )
         add-referenced-link( Con, Rel )
      $)
   $)
   APPEND-TO( to-be-processed, relation )
   UNTIL to-be-processed = NULL DO
   $( LET Rel = HEAD-OF( to-be-processed )
      FOR EACH Con IN referencing-constraints( Rel )
      $( LET referencing-rel = referencing-relation( Con )
         add-constraint( Con, BY-CASCADE )
         add-relation( referencing-rel )
         add-referencing-link( referencing-rel, Con )
         add-referenced-link( Con, Rel )

         UNLESS update-rule( Con ) = RESTRICTED ) THEN
         $( IF intersect( referencing-attributes( Con ),
                          key( referencing-rel ) ) THEN
              APPEND-TO( to-be-processed, referencing-rel )
            FOR EACH C IN
             referenced-constraints( referencing-rel )
            TEST C = Con THEN
            $( IF ( quantifier( C ) = EXACTLY ONE OF ) &
                  ( number-of-referenced-relations( C ) > 1 )
               THEN
               $( change-marker( C, CASCADE-AND-TEST )
                  FOR EACH R IN referenced-relations( Con )
                  $( add-relation( R )
                     add-referenced-link( Con, R )
                  $)
               $)
            $)
            ELSE IF intersect( referencing-attributes( Con ),
                               referencing-attributes( C ) ) THEN
            $( add-constraint( C, TO-BE-TESTED )
               add-referencing-link( referencing-rel, C )
               FOR EACH R IN referenced-relations( C )
               $( add-relation( R )
                  add-referenced-link( C, R )
               $)
            $)
         $)
      $)
   $)
$)
```

6.11.3 Recovery of Reference Graph Fragment for DELETE

```
LET process-delete( relation ) BE
$( LET to-be-processed = NULL
   add-relation( relation )
   APPEND-TO( to-be-processed, relation )
   UNTIL to-be-processed = NULL DO
   $( LET Rel = HEAD-OF( to-be-processed )
      FOR EACH Con IN referencing-constraints( rel )
      $( LET referencing-rel = referencing-relation( Con )
         add-constraint( Con, BY-CASCADE )
         add-relation( referencing-rel )
         add-referencing-link( referencing-rel, Con )
         add-referenced-link( Con, Rel )
         UNLESS deletion-rule( Con ) = RESTRICTED THEN
           IF intersect( referencing-attributes( Con ),
                         key( referencing-rel ) ) THEN
              APPEND-TO( to-be-processed, referencing-rel )
      $)
   $)
$)
```

Chapter 7

AN EXAMPLE DATABASE: The University Computing Service

In database research it is always nice to have a real database on which to test one's system. The example to hand was the administrative database of the University of Cambridge Computing Service. This is not a large database since it contains only a few megabytes of data; however, neither is it a toy database either in terms of size or of structure. There is sufficient variety of structure present for the data model to provide a reasonable test of the system described in this dissertation.

There are two reasons for having this example.

(a) The first reason is to demonstrate the part played by referential constraints in the description of a database.

(b) The second reason is to show that the system works on a serious database. However, it was not intended nor was there time available to perform a detailed performance evaluation or a comparison with the existing DBMS used by the Computing Service.

The first part of the chapter describes the structure of the database, gives details of the sizes of the various relations and gives the schema produced for the database. It will be seen that referential constraints are the major modelling tool, with domains and intra-tuple constraints playing a more minor role. It would be nice to be able to state that there is nothing in the database that it was not possible to model. Unfortunately this is not the case; the failings of the model are pointed out and suggestions made as to how these might be remedied. However, even allowing for these deficiencies a better model has been produced than was possible with the current database system used by the Computing Service. This model would in no way be possible with a relational system which did not support referential constraints.

The second part of the chapter gives some examples, taken from this database, of how referential constraints are maintained.

7.1 The World of the Database

The database deals with the allocation and accounting of computing resources within the University. Authorisation to use the computer is controlled by authorisations between users and projects. Projects are, for accounting purposes, grouped into accounts which usually correspond to entities like university departments. Accounts in turn are collected into account groups which are organised into a tree which models the administrative hierarchy of the University. The root of the tree of account groups is the Computing Service (a fact which some may regard as an anomaly).

As well as accounting information, the database also contains information about users and the resources allocated to them independently of the projects which they are authorised to use. Such resources include magnetic tapes and disc filespace allocations. This part of the database is linked to the accounting structure via the authorisations of users of projects.

The above description indicates that there is quite a lot of structural information in the database schema. This means that the data model will contain a large number of referential constraints. The following sections describe the data model in detail. Note that since my primary interest is in the structure of the database many of the data fields which do not imply links to other relations have been omitted from the description. These data fields were also omitted from the database on which tests were carried out. The size of data in this database was about 1.5 megabytes.

7.2 The Accounting Structure

The base for this part of the database is the tree of account groups. This is modelled by the relation:

```
Account_groups : [ Name | Father ]
```

together with the referential constraint:

```
Account_group_tree :
   Account_groups.[Father] ->> Account_groups.[Name]
                          DELETION CASCADES UPDATE CASCADES
```

The convention is that for the root of the tree, since CODD does not support NULL values, 'Name' = 'Father'. Note that the referential constraint forces the deletion of all the sons of an account group when that account group is deleted.

Unfortunately there are two faults with the model as suggested.

(a) It is possible to create orphaned account group trees (i.e. trees not attached to the main accounting tree) by inserting tuples into 'Account_groups'.

(b) It is possible to create detached looped structures by inserting more than one tuple into 'Account_groups' in the same transaction, e.g. by inserting the tuples ("Loop1","Loop2") and ("Loop2","Loop1").

There are two ways of solving problem (a). Firstly the constraint that

```
( Account_groups.[Name] NE Account_groups.[Father] ) OR
( Account_groups.[Name] = "cserv" )
```

could be introduced. This would prevent the creation of more than one root node but has the disadvantage of binding the name of the root of the tree into the data model so that it is impossible to change it without causing some constraint to be violated. The second mechanism is to introduce a special relation

```
Root_of_account_tree : [Name]
```

together with the modified referential constraint

```
Account_group_tree :
  Account_groups.[Father] ->> (
                        Account.groups.[Name],
                        Root_of_account_tree.[Name] )
                  DELETION CASCADES UPDATE CASCADES
```

which is restricted to having a cardinality of one by having an intra-tuple constraint which binds the name of the root into the schema (this is an unfortunate side effect). If CODD supported cardinality constraints on references then the number of references to the root tuple could be limited to one.

The binding of the name of the root into the schema is not as serious in the second solution as in the first solution. This is because in the second scheme the root tuple may be regarded merely as a hook off which the world hangs with the effective root of the accounting system being the 'Account_groups' tuple which refers to it.

Problem (b) is more difficult to cure. A possible solution is to restrict to one the number of tuples which may be inserted into 'Account_groups' in any one transaction. However, this cannot be enforced by CODD.

Associated with each account group are one or more accounts. These are defined by the relation

```
Accounts : [ Name | Group ]
```

together with the constraint

```
Account_in_group :
   Accounts.[Group] ->> Account_groups.[Name]
                        DELETION CASCADES UPDATE CASCADES
```

Again the semantics of deletion are that if an account group is deleted then all of the accounts associated with that group are also deleted.

The last part of the accounting structure is 'Projects'. These represent accountable resources which users use. Projects are grouped into accounts and are defined by the relation

```
Projects : [ Project_no | Account, Number_of_shares ]
```

together with the now familiar constraint

```
Project_in_account :
   Projects.[Account] ->> Accounts.[Name]
                      DELETION CASCADES UPDATE CASCADES
```

7.3 Users and Resources Associated with Users

Users of the Computing Service are typically people and are defined by the relation

```
Users : [ User_id | Name ]
```

Among the resources owned by users are magnetic tapes which are defined by

```
Tapes : [ Name | Owner ]
```

and these are linked to their owners by

```
Owner_of_magnetic_tape :
   Tapes.[Owner] ->> Users.[User_id]
         DELETION RESTRICTED
```

Note that the referential constraint ensures that a user cannot be deleted until all of his tapes have been either deleted or transferred to another user.

Tapes represent the first example of generalisation in the database since they are divided into two groups; those stored in racks and those stored in some external place. This situation is described by the following relations and constraints.

```
Tapes_in_racks : [ Tape | Rack ]
Tapes_not_in_racks: [ Tape | Location ]

Is_tape1 : Tapes.[Name] ->> ( Tapes_in_racks.[Tape],
                              Tapes_not_in_racks.[Tape] )
                        DELETION CASCADES UPDATE CASCADES
Is_tape2 : Tapes_in_racks.[Tape] ->> Tapes.[Name]
                        DELETION CASCADES UPDATE CASCADES
Is_tape3 : Tapes_not_in.racks.[Tape] ->> Tapes.[Name]
                      DELETION CASCADES UPDATE CASCADES
```

The racks which are available for the storage of tapes are represented by the relation:

```
Racks : [ Name ]
```

which is linked to 'Tapes_in_racks' by

```
Exists_rack : Tapes_in_racks.[Rack] ->> Racks.[Name]
                      DELETION RESTRICTED UPDATE CASCADES
```

Between 'Tapes_in_racks' and 'Racks' there is also the constraint that there can only be one tape in any rack. This relationship can be modelled by a cardinality constraint. An alternative way of enforcing this constraint would be for both 'Tape' and 'Rack' to be considered to be keys for 'Tapes_in_racks'. However, the fact that the rack attribute behaves like a key is in some ways an accident, in that the constraint could just as easily have been that 'a rack can contain up to ten tapes'. Therefore, the cardinality constraint is the better way to model the constraint.

7.4 Authorisations: The link between users and the accounting tree

So far I have discussed two separate facets of the database. These are linked via the authorisation of users to make use of projects. A single user may be authorised to use several projects and each project may have several authorised users. Therefore the relationship between users and projects is many-to-many. The relationship has the semantics of an association in that if either the user or the project participating in an authorisation is deleted, then the authorisation itself must be deleted. This part of the database is represented by the relation

```
Authorisations : [ User, Project | ]
```

and the two referential constraints

```
Auth1 : Authorisations.[User] ->> Users.[User_id]
        DELETION CASCADES UPDATE CASCADES
Auth2 : Authorisations.[Project] ->> Projects.[Project_no]
        DELETION CASCADES UPDATE CASCADES
```

This concludes the description of the database. The above sections show that the modelling exercise was reasonably successful. Further, the effort involved in specifying the data model and creating a test database was small (it took about a day). This shows that the techniques which have been described provide a powerful data definition language. The example illustrates the considerable amount of structure which can be captured by the types of constraints, particularly referential constraints, treated in this dissertation.

For the sake of clarity the complete schema together with the cardinalities of the relations in the example database is given in figure 7.1. A graphical description of the schema is given in figure 7.2.

Relations and Intra-tuple constraints (Cardinalities in Brackets)

```
Root_of_account_tree : [Name]                                          (1)
Account_groups : [ Name | Father ]                                     (107)
   ( Account_groups.[Name] NE Account_groups.[Father] ) OR
   ( Account_groups.[Name] = "cserv" )
Accounts : [ Name | Group ]                                            (503)
Projects : [ Project_no | Account, Number_of_shares ]                  (4455)
Authorisations : [ User, Project | ]                                   (5807)
Users : [ User_id | Name ]                                             (5228)
Tapes : [ Name | Owner ]                                               (11216)
Tapes_in_racks : [ Tape | Rack ]                                       (7579)
Tapes_not_in_racks: [ Tape | Location ]                                (3637)
Racks : [ Name ]                                                       (8932)
```

Referential Constraints

```
Account_group_tree :
  Account_groups.[Father] ->> (
                        Account.groups.[Name],
                        Root_of_account_tree.[Name] )
                      DELETION CASCADES UPDATE CASCADES
Account_in_group : Accounts.[Group] ->> Account_groups.[Name]
                      DELETION CASCADES UPDATE CASCADES
Project_in_account : Projects.[Account] ->> Accounts.[Name]
                      DELETION CASCADES UPDATE CASCADES
Auth1 : Authorisations.[User] ->> Users.[User_id]
                      DELETION CASCADES UPDATE CASCADES
Auth2 : Authorisations.[Project] ->> Projects.[Project_no]
                      DELETION CASCADES UPDATE CASCADES
Owner_of_magnetic_tape : Tapes.[Owner] ->> Users.[User_id]
                      DELETION RESTRICTED
Is_tape1 : Tapes.[Name] ->> ( Tapes_in_racks.[Tape],
                              Tapes_not_in_racks.[Tape] )
                      DELETION CASCADES UPDATE CASCADES
Is_tape2 : Tapes_in_racks.[Tape] ->> Tapes.[Name]
                      DELETION CASCADES UPDATE CASCADES
Is_tape3 : Tapes_not_in.racks.[Tape] ->> Tapes.[Name]
                      DELETION CASCADES UPDATE CASCADES
Exists_rack : Tapes_in_racks.[Rack] ->> Racks.[Name]
                      DELETION RESTRICTED UPDATE CASCADES
```

Figure 7.1: The Complete Database Schema

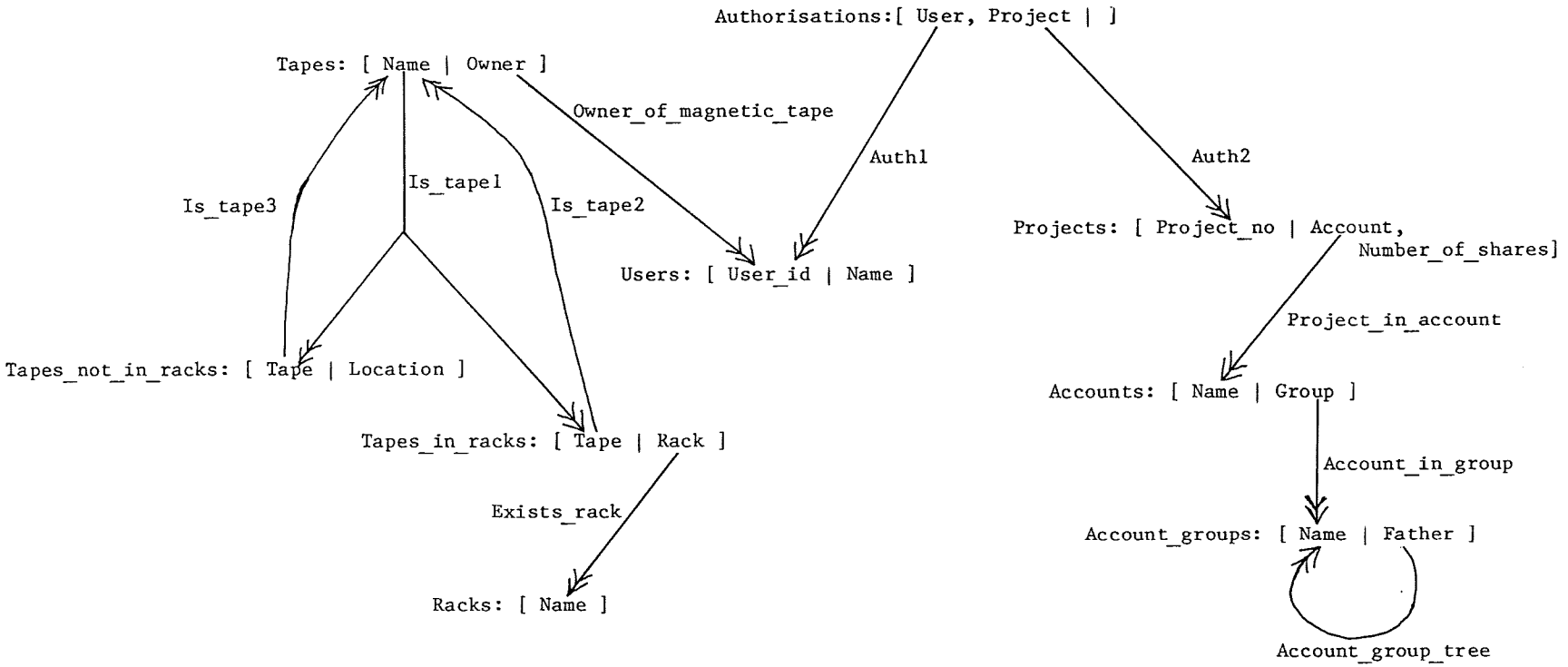


Figure 7.2: A Graphical Representation of the Database Schema

7.5 Examples of the Maintenance of Referential Constraints

Chapter 6 described the techniques used to maintain referential constraints in CODD. This section takes some example modifications to the Computing Service Database and shows the structures which CODD builds to maintain referential constraints. The examples which will be presented are:

(a) insertion of a single tuple into 'Authorisations';

(b) insertion of new project and an authorisation for an existing user to use that project;

(c) updating the account with which a project is associated;

(d) deletion of a tuple from 'Authorisations';

(e) deletion of a tuple from 'Users';

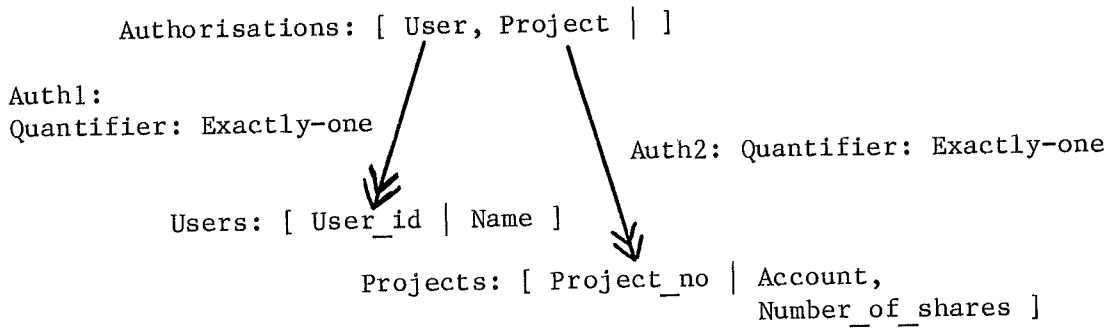
(f) deletion of the root of the account-group tree.

The examples are each illustrated by figures showing the dependency graph fragment recovered for each and the coroutine structure which is built to perform the operation.

7.5.1 Insertion of a single tuple into Authorisations

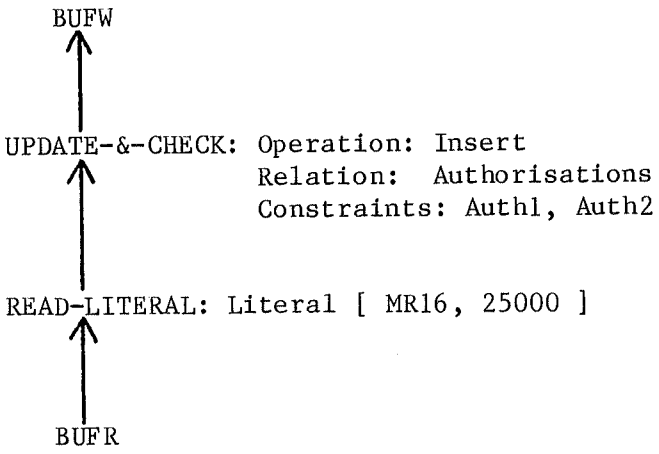
Consider the insertion of an authorisation for the user "MR16" to use the project 25000.

The dependency graph recovered for this operation is:



This shows that constraints 'Auth1' and 'Auth2' need to be checked.

The coroutine graph which is built to perform the requested operation is:



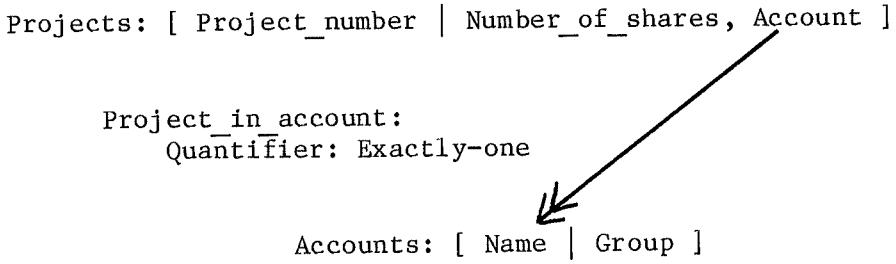
7.5.2 Insertion of a new project and an authorisation to use it

Consider the transaction which involves creating a new project with project number 99000, with 20 shares and in the account "cs-research".

This transaction involves two separate operations

- o Insertion of a tuple into Projects
- o Insertion of a tuple into Authorisations

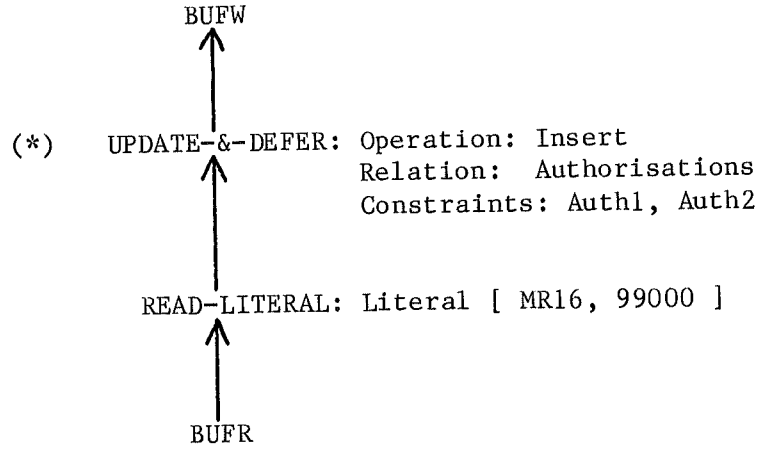
Two dependency graphs are recovered in this example; one for the modification to ´Projects´ and one for the modification to ´Authorisations´. The dependency graph for ´Authorisations´ is the same in the previous example. The dependency graph for ´Projects´ is:



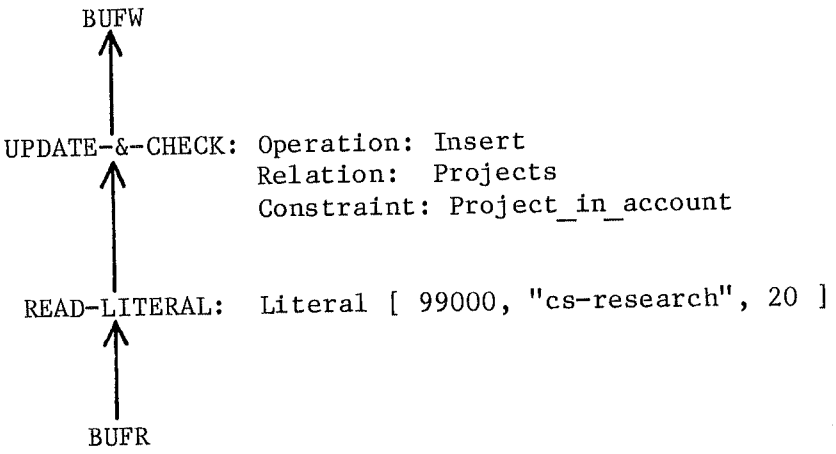
CODD performs the operations in the order they are specified to it. The system is not clever enough to reorder the operations. The point at which the constraint ´Auth2´ is checked depends on whether the insertion into ´Projects´ occurs first or not.

If the new project is inserted first then when the authorisation is created the constraint 'Auth2' can be checked immediately. In this case a coroutine graph similar to that in section 7.5.1 executed to insert the tuple into 'Projects' followed the graph described in section 7.5.1 to insert the tuple into 'Authorisations'.

If the authorisation is created first then the checking of the constraint 'Auth2' is deferred until the end of the transaction, using the graph



to insert the tuple into 'Authorisations' and the graph



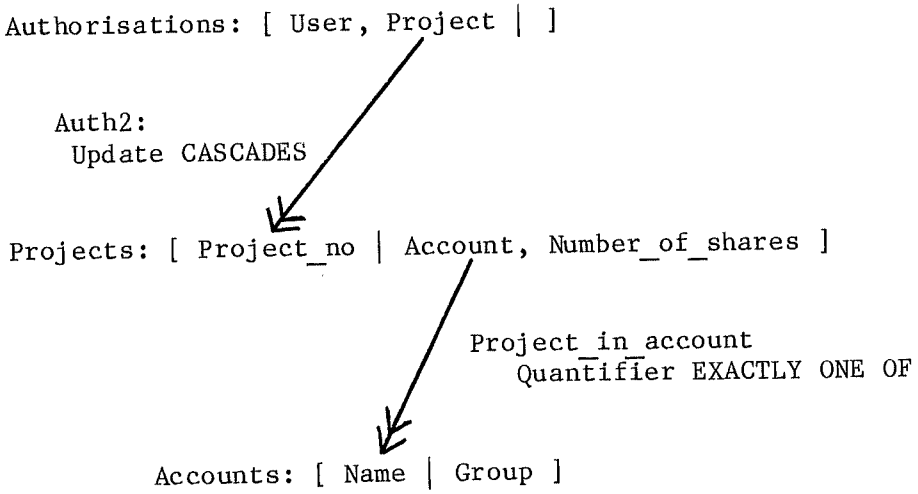
to insert the tuple into 'Projects'.

The tuples written by the UPDATE-&-DEFER (*) are checked under 'Auth2' after the insertion into 'Projects' has taken place.

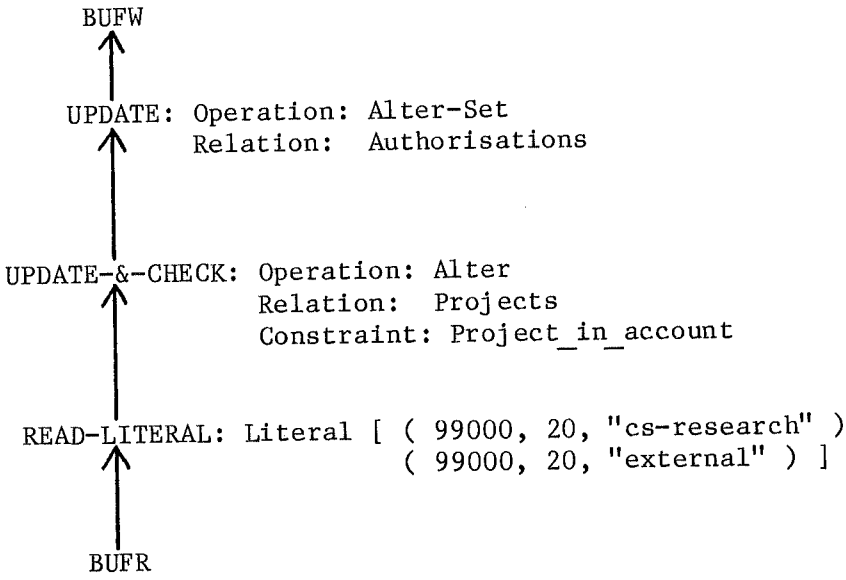
7.5.3 Moving a project from one account to another

Consider moving the project 99000 with 20 shares from "cs-research" to "external".

The dependency graph fragment recovered for this example is:



The coroutine structure which is built is:



Note that when UPDATE is performing an alteration then it does not produce a cascaded tuple when the key of the tuple has not been modified because for a CASCADEd constraint link it would be pointless.

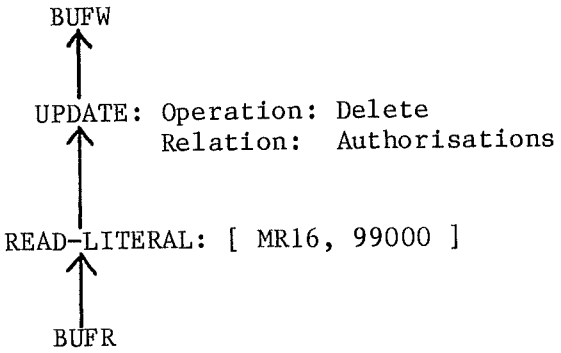
7.5.4 Deletion of a tuple from Authorisations

Consider the deletion of the authorisation of the user MR16 to use the project 99000.

Authorisations is not the referenced relation for any constraints and deletion does not cause any constraints to be checked. This operation is therefore very simple; the dependency graph consists solely of:

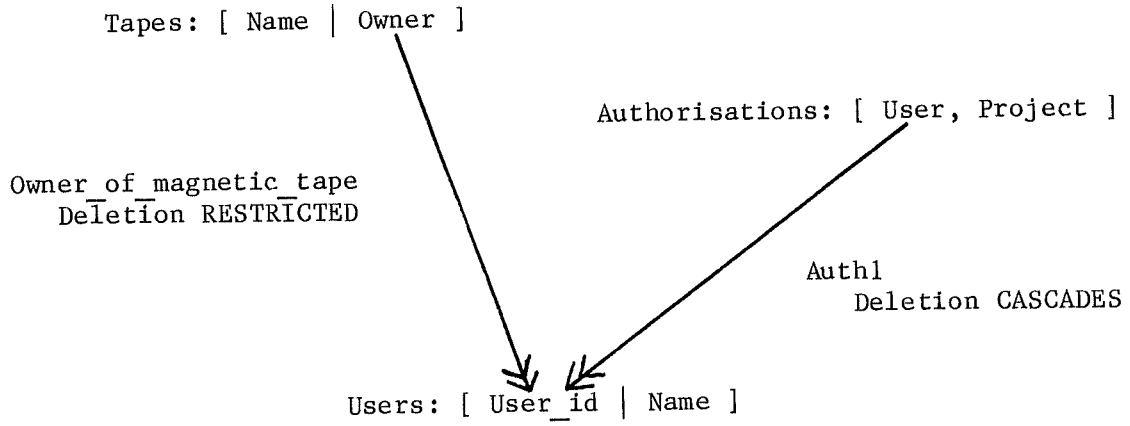
```
Authorisations: [ Project, User | ]
```

and the coroutine program is:



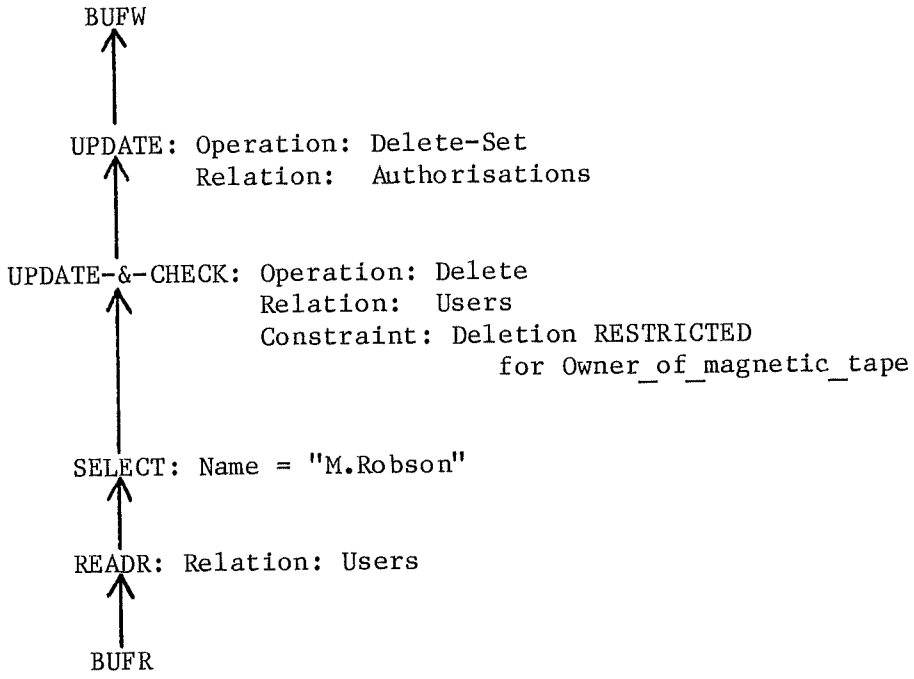
7.5.5 Deletion of a tuple from Users

Consider the deletion of the user(s) called "M.Robson" from the database. The dependency graph recovered is:



Note that the semantics of this graph are that it a user who is still the owner of some tapes. The CASCADES link in 'Auth1' causes any authorisations for the user to be deleted.

The coroutine graph built for this operation is:

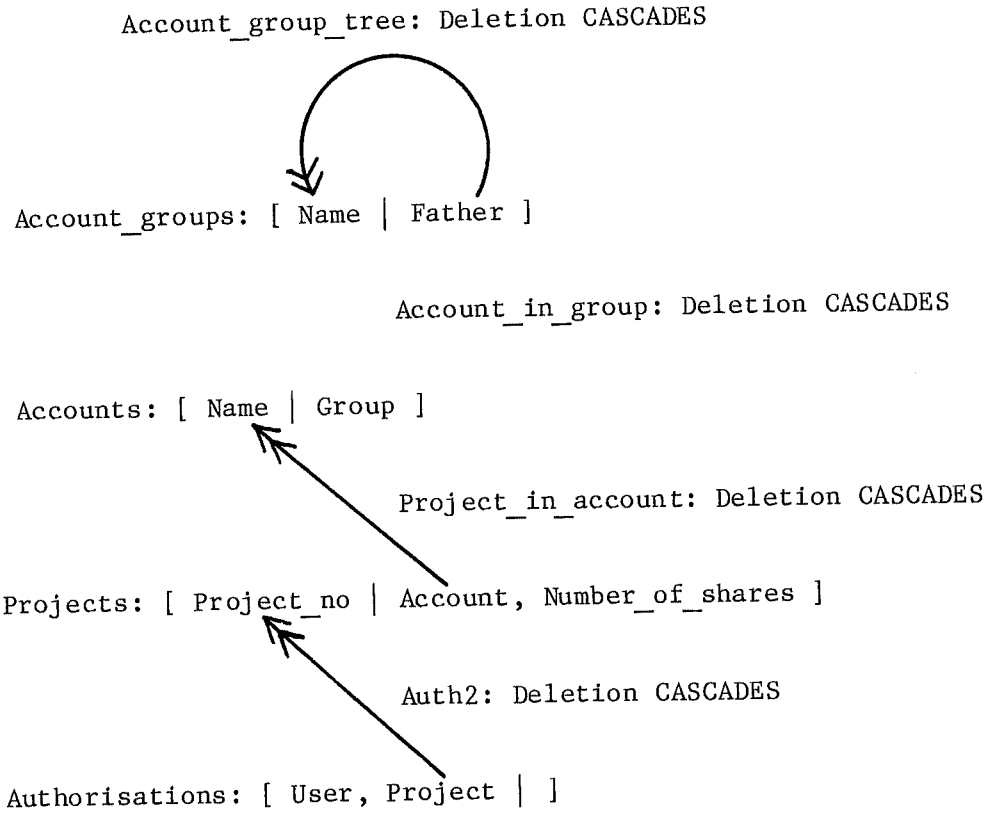


This example illustrates the use of a coroutine normally used in queries (SELECT) in an UPDATE transaction.

7.5.6 Deletion of the root of the account-group tree

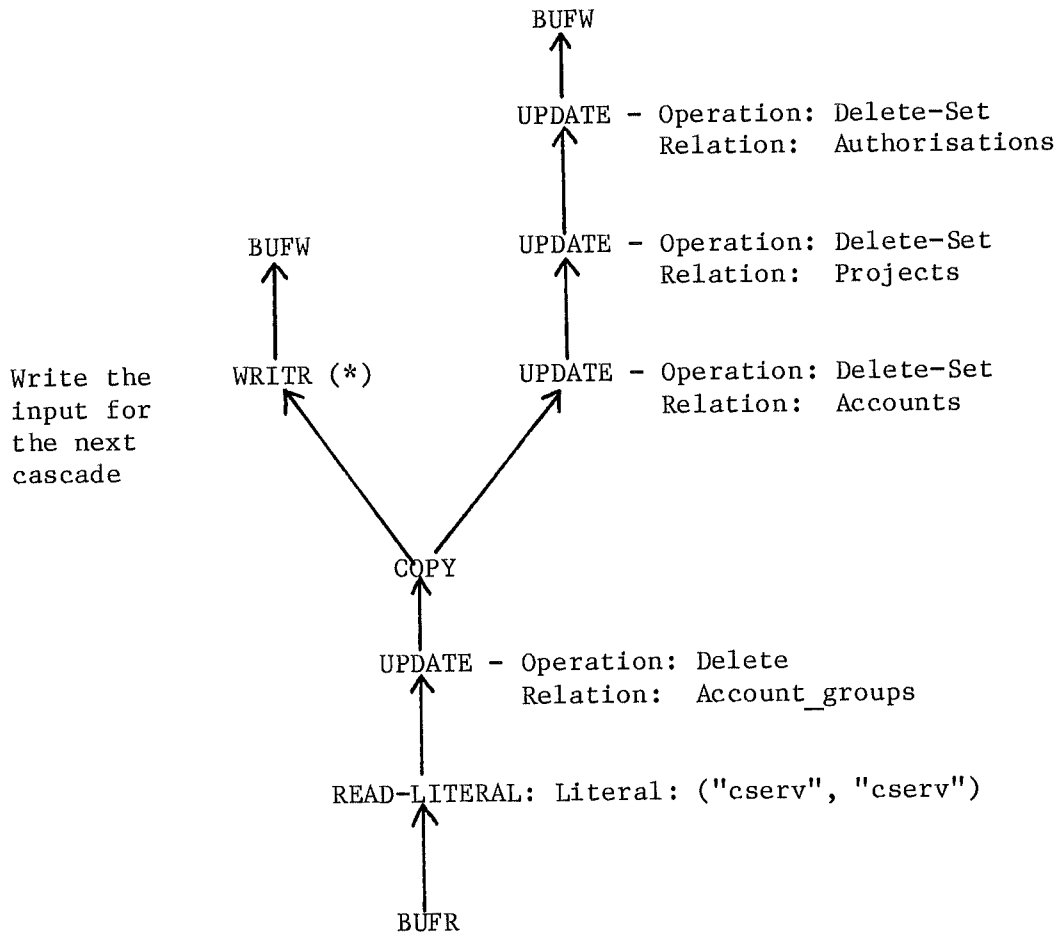
The root of the accounting tree is the tuple ["cserv", "cserv"] in ´Account_groups´. This example considers the effect of deleting this tuple.

The dependency graph fragment recovered for this operation is:

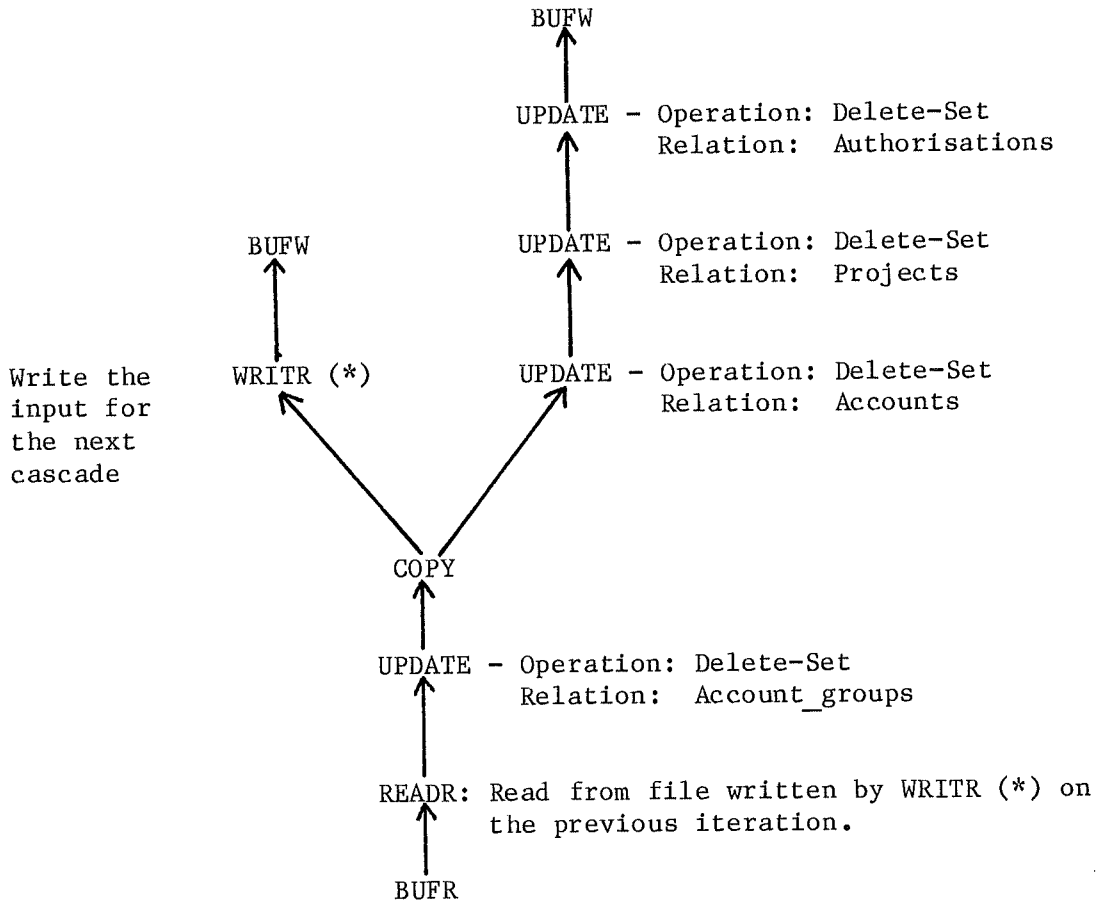


Since this graph contains a cycle the CUTLIST will not be empty; it will contain an entry for the cycle produced by 'Account_group_tree'.

For the first iteration of the operation the coroutine graph which is built is.



When this structure completes execution the list of cuts is examined to determine the number of tuples written by the WRITR (*). If (*) has written any tuples then the information associated with the cut is used to build the structure shown below in which the READR reads the tuple written by (*).



This structure executes, completes and is rebuilt until no tuples are written by WRITR (*). For the example database the loop containing Account_groups is traversed seven times.

This transaction should (and does) result in the deletion of all of the account groups, accounts, projects and authorisations. For this operation the times taken, both in terms of CPU time and disc channel time, are approximately two orders of magnitude less than those taken by the Computing Service's existing DBMS to perform a similar operation (naturally the Computing Service would not wish to perform the operation which has been described; indeed it would be sensible to specify DELETION RESTRICTED for 'Account_group_tree' and 'Account_in_group' in order to avoid a disaster).

Deleting the root of the accounting tree is a complex operation which involves several levels of cascade.

7.6 Experience with this Database

The main reason for looking at the Computing Service Database was to discover whether a sensible data model could be produced rather than to do detailed performance tests. However, from the small number of tests which were performed the observation given below can be made.

(a) The cost of operations which make small modifications to the database is relatively high, e.g. insertion, deletion and alteration of single tuples which do not cause many CASCADES operations. This is largely because the cost of accessing the catalogues and building the graph and coroutine structures necessary to execute the transaction greatly outweighs the cost of doing the actual modification. It is noticeable that inserting several tuples into the same relation at the same time is not much more expensive than inserting a single tuple.

(b) Complex deletions or alterations which involve many cascaded tuples and many levels of cascade execute quickly. The most striking example of this operation is the transaction which deletes the root of the accounting tree.

(c) Experience with the initial loading of the database suggests that the optimisation involving the sorting of updates would in fact be of considerable value. This conclusion was reached by simulating the effect of using this strategy by loading the database before defining the referential constraints and then defining the constraints. Done in this way the loading of the data, together with the checking of the constraints, is cheap compared with defining the schema completely and then loading the database. This is because the inversions are in the first case loaded in sorted order whereas in the second case they are not.

7.7 Summary

Using the tools described in the earlier chapters of the dissertation, much of the structure of a real database has been modelled successfully. Further, although CODD is not used by the Computing Service, CODD was more successful in modelling the database than a number of the available DBMSs which they have considered using. For the particular database considered the cost of modifying the database was certainly not prohibitively expensive, which is satisfying. Therefore, it seems that the techniques which have been suggested are useful for dealing with real database problems.

Chapter 8

CONCLUSIONS

8.1 Review

In the introduction it was shown that the connections between the classes of objects in a database are important and that the description of these connections is an integral part of a number of data models. The absence of a way of describing inter-class connections in the Relational Model was argued to be a serious defect.

A detailed investigation of a proposal for incorporating inter-relational connections into the Relational Model, namely referential constraints, has been presented. This proposal has been compared both with other proposals and with the facilities provided in other data models. An implementation of referential constraints for the relational DBMS CODD has been described. The implementation makes considerable use of data pipelines implemented using coroutines, a novel feature of CODD.

Two further types of constraint, domain and intra-tuple constraints, have been considered in the dissertation. Whereas referential constraints deal with structures which may span more than one relation, domain and intra-tuple constraints deal with structure within single relations and provide provide useful checks when data is entered into the database. The techniques for maintaining domain and intra-tuple constraints are similar but the two types of constraint provide different semantics and for this reason they were treated separately.

The descriptions of all of the constraints are stored in the database. Such a centralised database model is important since it is far too easy for information about the semantics of the database to become embedded in a set of application programs; when this happens the semantic information which the programs contain becomes hidden from the user.

8.2 Conclusions

The previous section reviewed the work which has been done; this section presents the major conclusions.

(a) Referential constraints are an important addition to the Relational Model.

The University Computing Service example illustrates how great a part they play in the description of a database; it would not be possible to model much of the structure of this database using just the basic Relational Model. It is necessary that both the user and the system are aware of the connections between relations. The user needs the knowledge in order to update the database sensibly. The system needs the knowledge both to ensure that the user makes only permissible modifications and to provide information to other programs which require knowledge of inter-relational connections; for example, a program providing a natural language interface or one which explains the database structure to the user. Referential constraints provide a description of the inter-relational connections which is accessible to both the user and the DBMS.

(b) The evaluation of update transactions by use of coroutine graphs is a useful technique.

This is not only because the pipelining which the technique provides can produce gains in performance but also because the technique allows complex database operations to be composed from a small number of simple processing elements, the coroutines which implement the individual operations.

The information about inter-relational connections is represented in the data model as a network. A nice feature of the use of coroutines to maintain referential constraints is the way in which the coroutine graph reflects directly the network structure represented in the data model.

The way in which coroutines are used in CODD gives:

- a uniform mechanism and computational model for both query and update;
- a uniform mechanism for dealing with both simple and complex operations on the data.

One of the aims of this work was to demonstrate that pipelined evaluation could be useful when performing database modifications; it is satisfying that this existing technology could indeed be adapted to a task other than that for which it

was designed initially. [T. King 79] demonstrated that data pipelining is a good technique for evaluating complex database queries; the work in this dissertation demonstrates that pipelining is also useful when modifying the database.

(c) The data model is not just a static description of how the database should be; it also contains rules to ensure that the contents of the database satisfy the description.

- o The implementation of domains and intra-tuple constraints stores procedural information in the database.
- o The implementation of referential constraints uses the data model to generate programs, executable coroutine graphs, to maintain the constraints.

The data model is also used to generate automatically coroutine graphs which check that new constraints are satisfied by the current contents of the database before they are added to the data model.

In some ways the path which has been followed in this work is the opposite of that pursued by the advocates of persistent data in programming languages [Atkinson 78]. They add persistent data objects to a programming language whereas this work is a step in the direction of adding algorithms to the data.

(d) The ability to enforce the constraints which have been investigated forms the base on which DBMSs which use other data models as their means of describing the database can be constructed.

The facilities which have been provided supply the structures, particularly inter-class connections, required by other data models. It has been shown that the ability to maintain referential constraints can be used as a building block to construct other network structures in the database. Only experimentation will prove the truth or falsehood of this conclusion.

(e) Although the implementation of domains which has been described is an adequate means of maintaining the constraints implied by domains, the implementation suffers from the defect that it does not provide definitions of the operations which are allowed on domains. Hence it may prove difficult to integrate with the query language. This issue would have to be investigated if a uniform user interface were being produced.

Another problem with the techniques used to implement domains is that it is too powerful. The domain checkers can be very flexible; however, since they are arbitrary BCPL programs, the DBMS has no control over their action; it has to trust them. In retrospect a better approach would be to have produced a small number of primitive checkers and a language, known to the DBMS, in which more complex checkers could be implemented using the primitive facilities.

8.3 Critique

Although the implementation which has been produced has a number of strong features, which have been described above, there are a number of areas in which it could be improved. The following sections outline these areas and suggest how the problems could be approached.

8.3.1 The Maintenance of Referential Constraints

When cascaded operations which involve large numbers of tuples are performed, pipelining can produce considerable gains; the example of deleting the root of the accounting tree in the University Computing Service example illustrated this. On the other hand insertion or deletion of single tuples with few dependent tuples involves considerable fixed overheads, which include the catalogue accesses and the optimisation of the executable coroutine graph. The same problem arises for queries, where the fixed overheads make simple queries relatively expensive and where the data pipelining allows complex queries to be evaluated at a reasonable cost. In a system where insertions of single tuples are done frequently the fixed overhead would be unacceptably great. It is, however, possible to construct a system where the coroutine graph for an operation is retained when the operation completes and where it is reset to a state ready to receive further input. In such a system a frequently used operation would be activated simply by providing it with some input. Initialisation of frequently invoked operations could then become part of the initialisation of CODD.

The problem of checking referential constraints was simplified by ensuring that all of the inversions which were required for constraint checking were always present. This is expensive on disc space. Therefore, a topic for further work would be to investigate the problems, costs and advantages that would arise if some of the inversions were not present. Some topics of interest are:

- Which inversions are worth maintaining?

- Is it possible to implement an adaptive scheme in which access paths available are tuned automatically to the usage of the database?

- Connected with the problem of deciding which inversions to maintain is the problem of how the inversions should be maintained. Currently CODD uses a single UPDATE node to maintain all of the inversions of a relation. A possible optimisation is to provide one UPDATE node per inversion.

8.3.2 Bulk Update

Bulk update of the database is a long sequence of small transactions which are carried out together for reasons of efficiency. From the point of view of success or failure of the transaction bulk modification should, therefore, be treated as many small transactions. At present CODD treats a bulk update as a single transaction which is committed or rolled back as a whole. This would be unacceptable in a commercial system.

8.3.3 User Interface

A criticism of the current system is that it does not have a uniform user interface; the existing interface is an ad hoc collection of ways of describing particular parts of the data model together with a query language which has evolved rather than been designed. A more uniform user interface must be constructed if CODD is to be developed further or used in real applications. One possibility would be to provide a DAPLEX interface to the DBMS. This would involve investigation of the mapping between DAPLEX schemas and relational schemas; this should not prove too difficult.

Hand in hand with the need to provide a uniform user interface goes the need to provide better facilities for querying the data model. It should be possible to ask questions like "what might be affected if I alter tuples in relation Y?" and "what constraints apply to relation Y?", instead of merely being able to ask what the constraints are. It has been asserted that the schema ought to be a documentation aid; therefore, it should be possible to query the schema in a flexible way.

8.3.4 Use in a Real Application

A great deal of effort (both on the part of the author and of a number of other people) has gone into the development of CODD. However, apart from King's original project on historical records, it has not been used for a real application. It would be interesting to test the facilities which CODD now provides in a real application. It would then be possible to determine how well, or badly, it performs in serious use. Sadly this poses the question of who will be willing to use it until it is proven?

8.4 Further Work

The previous section made some very specific suggestion on ways in which the work which has been done can be extended. This section suggests some broader areas of investigation.

(a) The coroutines in the graphs evaluated by CODD are not autonomous processing elements since the flow of control during the execution of a graph is single threaded. However, there is no reason why the nodes in a pipeline graph should not be autonomous processes and consequently be executed on different machines. In recent years there has been interest in the development of parallel architectures in general and data-flow machines in particular; the processing facilities embodied in CODD could form the basis of the implementation of a hardware pipelined machine.

(b) In all of its development CODD has not addressed the problems provided by the presence of NULL values in a database. How best to deal with NULL values remains an open question and one which is a suitable topic for further work. For example, can the ideas suggested by [Gray 83] be incorporated into a real DBMS?

(c) It was remarked above that knowledge of the connections between facts in a database is necessary for the construction of a natural language interface to a database. In what directions would the facilities provided by CODD have to be extended in order to provide such support? One facility which would need to be provided is better access to the data model; in particular it is desirable to have a procedural interface to the data model which the natural language processing system could call. Further, it would be necessary for the DBMS and the natural language processing system to be able to communicate in similar terms; this is likely to involve the DBMS maintaining information about the meaning as well as the structure of the data in the database, i.e. a more 'semantic' data model; the Semantic Relation and Network Models described in [Borkin 80b] may provide a suitable starting point.

REFERENCES

[Abrial 74]
Abrial J.R.
Data Semantics
In: Database Management, ed. Klimbie and Koffeman, pp1-59, North-Holland (1974)

[Addis 82]
Addis T.R.
A Relation-Based Language Interpreter for a Content Addressable File Store
ACM-TODS Vol. 7 No. 2 pp125-163 (June 1982)

[Astrahan, et al 76]
Astrahan M., et al
SYSTEM/R: A Relational Approach to Database Management
ACM-TODS: Vol. 1 No. 2 pp97-137 (June 1976)

[Atkinson 78]
Atkinson M.P.
Programming Languages and Databases
Proc. VLDB 1978, Berlin, West Germany

[Atkinson, Chisholm, Cockshott 82]
Atkinson M.P., Chisholm K.J. and Cockshott W.P.
PS-Algol: An Algol with a Persistent Heap
ACM SIGPLAN Notices: Vol. 17 No. 7 pp24-31 (July 1982)

[Atkinson, Kulkarni 83]
Atkinson M.P. and Kulkarni K.G.
Experimenting with the Functional Data Model
Persistent Programming Research Report No. 5, University of Edinburgh, Department of Computer Science (September 1983)

[Boguraev, Sparck Jones 83]
Boguraev B.K. and Sparck Jones K.
Final Report on SERC Grant GR/B27159:
Natural Language Query Processor for Database Access (November 1983)

[Borkin 80a]
Borkin S.A.
The Semantic Relation Model: Foundation for a User Interface
Proc. International Conference on Databases, ed. Deen and Hammersley, pp47-64, Heyden (1980)

[Borkin 80b]
Borkin S.A.
Data Models: A Semantic Approach to Database Systems
MIT Press (1980)

[Brachman 79]
Brachman R.J.
On the Epistemological Status of Semantic Networks
In: Associative Networks: Representation and Use of Knowledge by Computers, Findler N.V. (ed), Academic Press (1979)

[Buneman, Frankel, Nikhil 82]
Buneman P., Frankel R.E. and Nikhil R.
An Implementation Technique for Database Query Languages
ACM-TODS Vol. 7 No. 2 pp164-186 (June 1982)

[Brodie 81]
Brodie M.L.
Association: A Database Abstraction for Semantic Modelling
Proc. 2nd. International Entity-Relationship Conference, 1981

[Brodie, Zilles 80]
Brodie M.L. and Zilles S.N. (eds.)
Proc. of the Workshop on Data Abstraction, Databases and Conceptual Modelling, Pingree Park, Colorado 1980
ACM SIGMOD Record Vol. 11 No. 2

[Chamberlin, et al 76]
Chamberlin D.D., et al
SEQUEL2: A Unified Approach to Data Definition, Manipulation and Control
IBM Journal of Research and Development Vol. 20 No. 6
pp560-575 (November 1976)

[Chen 76]
Chen P.
The Entity-Relationship Model: Toward a Unified View of Data
ACM-TODS Vol. 1 No. 1 pp9-36 (March 1976)

[Codd 70]
Codd
A Relational Model of Data for Large Shared Data Banks
Comm. ACM Vol. 13 No.6 pp377-387 (June 1970)

[Codd 79]
Codd E.F.
Extending the Relational Model to Capture More Meaning
ACM-TODS Vol. 4 No. 4 pp397-434 (December 1979)

[Date 81a]
Date C.J.
Referential Integrity
Proc. VLDB 81, Cannes, France pp2-12 (1981)

[Date 81b]
Date C.J.
An Introduction to Database Systems (3rd Edition)
Addison Wesley (1981)

[Date 82]
Date C.J.
Null Values in Database Management
Proc. Second British National Conference on Databases, Bristol, England (1982), ed. Deen and Hammersley, pp147-166 (to be published by John Wiley)

[Dewar, McCann 77]
Dewar R.B.K. and McCann A.P.
MACRO SPITBOL: A SNOBOL-4 Compiler
SOFTWARE: Practice and Experience Vol. 7 pp95-113 (1977)

[Eswaran 76]
Eswaran K.P.
Aspects of a Trigger Subsystem in an Integrated Database System
Proc. 2nd. International Conference on Software Engineering pp243-250 (1976)

[Eswaran, Chamberlin 75]
Eswaran K.P. and Chamberlin D.D
Functional Specifications of a Subsystem for Database Integrity
Proc. VLDB 1975 pp48-68

[Glauert 81]
Glauert J.R.W.
The Protocol for Pipeline Communication in CODD
University of Cambridge Computer Laboratory, Database Research Group Note (January 1981)

[Gray 83]
Gray M.A.
Views and Imprecise Information in Databases
Technical Report No. 38 (Ph.D. Dissertation), Univerity of Cambridge, Computer Laboratory (1983)

[Gray 81]
Gray P.D.M.
Use of Automatic Programming and Simulation to Facilitate Operations on a CODASYL Database
In: Database - Infotech State of the Art Report, Series 9 No. 8, pp345-369 ed. Atkinson M.P. (Permagon-Infotech 1981)

[Hall, Owlett, Todd 76]
Hall P., Owlett J. and Todd S.J.P.
Relations and Entities
In: Modelling in Database Management Systems, ed. Nijssen, pp201-220 North-Holland (1976)

[Hammer, McLeod 76]
Hammer M. and McLeod D.J.
A Framework for Database Semantic Integrity
Proc. 2nd. International Conference on Software Engineering, pp498-504

[Hammer, McLeod 80]
Hammer M. and McLeod D.J.
On Database System Architecture
In: Data Design - Invited Papers - Infotech State of the Art Report, Series 8 No. 4, ed. Atkinson M.P., pp177-201 (Infotech 1980)

[Hammer, McLeod 81]
Hammer M. and McLeod D.J.
Database Description with SDM: A Semantic Database Model
ACM-TODS Vol. 6 No. 3 pp351-386 (September 1981)

[Hepp 83]
Hepp P.E.
A DBS Architecture Supporting Coexisting User Interfaces: Description and Examples
Persistent Programming Research Report No. 6, University of Edinburgh, Department of Computer Science (August 1983)

[ISO 81]
ISO TC97/SC5/WG3 Preliminary Report
Concepts and Terminology for the Conceptual Schema
ed. van Griethuysen J.J., et al (1981)

[P. King 80]
King P.J.H.
A Critical Review of Aspects of Database Theory
Distributed Databases, ed. Draffan and Poole, pp33-56, Cambridge University Press (1980)

[T. King 79]
King T.J.
A Relational Database System for Historical Records
Ph.D. Thesis University of Cambridge (1979)

[King, Moody 83]
King T.J. and Moody J.K.M.
The Design and Implementation of CODD.
SOFTWARE: Practice and Experience Vol. 13 pp66-78 (1983)

[Larson 78]
Larson P.A.
Dynamic Hashing
BIT Vol. 18 pp184-201 (1978)

[McLeod 76]
McLeod D.J.
High Level Domain Definition
ACM: SIGPLAN Notices Vol. 11 Special Issue, Proc. Conference on Data: Abstraction, Definition and Structure pp47-57 (1976)

[Moody, Richards 80]
Moody J.K.M. and Richards M.
A Coroutine Mechanism for BCPL
SOFTWARE: Practice and Experience Vol. 11 pp765-771 (1980)

[Richards, Whitby-Strevens 79]
Richards M. and Whitby-Strevens C.
BCPL: The Language and its Compiler
Cambridge University Press (1979)

[Ridjanovic, Brodie 82]
Ridjanovic D. and Brodie M.L.
Semantic Data Model Driven Design, Specification and Verification of Interactive Database Transactions
Department of Computer Science, University of Maryland (April 1982)

[Robson 82a]
Robson M.
The Implementation of Domains and Referential Constraints in CODD
Proc. Second British National Conference on Databases, Bristol, England (1982), ed. Deen and Hammersley, pp203-217 (to be published by John Wiley)

[Robson 82b]
Robson M.
Constraints in CODD
Technical Report No. 22, University of Cambridge, Computer Laboratory (1982)

[Robson, King, Glauert 81]
Robson M., King T.J. and Glauert J.R.W.
A Relational Database for Minicomputers
In: Databases: Proc. First British National Conference on Databases, ed. Deen and Hammersley, Pentech Press (1981)

[Roussopoulos 77]
Roussopoulos N.D.
A Semantic Network Model of Databases
Ph.D. Thesis University of Toronto, Department of Computer Science, Technical Report No. 104 (1977)

[Sharman 77]
Sharman G.C.H.
Update-by-Dialogue: An Interactive Approach to Database Modification
IBM UK Laboratories Technical Report TR.12.164 (June 1977)

[Shipman 81]
Shipman D.W.
The Functional Data Model and the Data Language DAPLEX
ACM-TODS Vol. 6 No. 1 pp140-173 (March 1981)

[Smith, Smith 77a]
Smith J.M. and Smith D.C.P.
Database Abstractions: Aggregation
Comm. ACM Vol. 20 no. 6 pp405-413 (June 1977)

[Smith, Smith 77b]
Smith J.M. and Smith D.C.P.
Database Abstractions: Aggregation and Generalisation
ACM-TODS Vol. 2 No. 2 pp105-133 (June 1977)

[Stonebraker 75]
Stonebraker M.
Implementation of Integrity Constraints by Query Modification
Proc. ACM SIGMOD Conference 1975, pp65-78

[Stonebraker 80]
Stonebraker M.
Retrospection on a Database System
ACM-TODS: Vol. 5 No. 2 pp225-240 (June 1980)

[Todd 76]
Todd S.J.P.
The Peterlee Relational Test Vehicle: A system overview
IBM Systems Journal Vol. 15 No. 4, pp285-308 (1976)

[Tsichritzis, Lochovsky 82]
Tsichritzis D.C. and Lochovsky F.H.
Data Models
Prentice-Hall (1982)

[Weber 76]
Weber H.
A Semantic Model of Integrity Constraints on a Relational Database
In: Modelling in Database Management Systems, ed. Nijssen, pp269-292, North-Holland (1976)

[Weber, Stucky, Karszt 83]
Weber W., Stucky W. and Karszt J.
Integrity Checking in Database Systems
Information Systems Vol. 8 No. 2 pp 125-136 (1983)