Technical Report

UCAM-CL-TR-38
ISSN 1476-2986

Number 38

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Views and imprecise information in databases

Mike Gray

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 November 1982 by the author for the degree of Doctor of Philosophy to the University of Cambridge.

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

Preface

Thanks are due to the following people, who between them have helped me greatly with stimulating discussions and critical advice: my supervisor Ken Moody, Karen Needham, Roger Needham, Charles Jardine, Mike Robson, Ian Braid, Russ Athay, Fred Gray. Thanks also to Mike Gordon and Stephen Todd for their careful reading and helpful comments.
Finally thanks are also due to Shape Data Ltd. for the use of their facilities in the preparation of this text.
This work was supported by the Science Research Council.

CHAPTER 1

INTRODUCTION

Providing user views of a database is an important way of achieving data independence and ease of use of DBMSs. This dissertation discusses one aspect of the problem of supporting views. It is shown that a crucial factor in the support of views is the richness of the data model used, and in particular its ability to represent certain kinds of incomplete information. This dissertation discusses various ways of handling incomplete information, and the operations on views that can be supported. The implementation of an experimental system which supports views on a relational database is described.

The first chapter describes the problem of treating views as first-class objects, that is, allowing all the usual database operations to be performed on data in views. It is shown how this is related to the problem of representing incomplete information in the conceptual schema. The second chapter proposes the use of lattices to represent incomplete information, and shows how this covers various particular kinds of imprecise information. The third chapter reviews other work relating to imprecise information in databases. The fourth chapter discusses certain further implications of representing imprecise information, and makes proposals regarding the interpretation of keys, constraints, and the open-world assumption in this environment. The fifth chapter discusses in detail the relational operations that are appropriate with imprecise data, and proposes modified Join and Group-by operations. The implementation of a system with these features is discussed. Chapter six illustrates some of the points made by considering an example database, and finally chapter seven concludes this dissertation with a summary and examination of further possibilities.

1.1 Data independence and views

Data independence is concerned with enabling the user of a DBMS to refer to his data in terms which are convenient for him, regardless of the organisation of the database, or how that organisation may change. One approach to data independence has been that of the ANSI/SPARC X3 report and others. This provides for several levels of description of the database, called 'schemas'. In the ANSI/SPARC proposal there are three: Internal, Conceptual and External schemas. The Internal schema describes the way the DBMS regards the data as being stored in the system. This will include details of access paths and physical structures. The External schemas give the structure of the data as it is perceived by various users of the system. Many external schemas may exist, containing

differing subsets of the data, and structuring it in different ways. It has been argued [Nijs] that this should extend as far as allowing different users of the DBMS to see their data according to different types of data model, so that one might be using the relational model while another works with the CODASYL model. While this proposal is somewhat radical compared with what is commonly implemented, there is at least widespread agreement that the provision of External Schemas, or 'views', should allow more than simple subsetting of the data.

The Conceptual Schema provides a central level mediating between Internal and External schemas. The Conceptual Schema will describe the whole database in a form which is intended to be free of implementation details, and provides a central reference in terms of which the other schemas can be described.

The idea behind this structure is that the views of the database can be tailored to the different requirements of different users of the system. For example, data which is irrelevant to a given user, or which that user is not allowed to see, will not be contained in the user's view. The structure of the data in the view will reflect the way the user regards the world as being organised, so that the operations he wishes to perform will be simple in terms of the objects in the view, and most of the work will be done by the view support component of the DBMS. The Conceptual Schema provides a unifying implementation-independent description so that alterations to the way the data is stored (the Internal Schema) will not affect the definitions of the views.

To describe the conceptual schema we require some kind of 'high-level' data model -- that is, a model free from details of how the data will be stored. Many such models have been described, the best known being the Relational model. This provides an implementation-free way to describe

data, but does not really allow much of the semantics of the data to be described. Many models have been suggested to describe the semantics of the data, in order to allow the DBMS to behave more intelligently. It should however be emphasised that this is not a problem which can be solved completely. It is possible to go on forever adding more semantic description to databases, but the point up to which this is productive will vary from application to application.

The pure Relational Model, for example, gives almost no way of describing the many constraints which may apply to the data. It is very important to describe these, in order to enable the DBMS both to optimise its operation, and to detect errors in information supplied to it.

It has been suggested in several places that it is a good idea in Relational database systems to implement the concept of domain, although many existing systems do not. One possible approach is described in this dissertation.

1.2 The example schema

The following set of relations, describing a collection of rock samples, will be used in examples throughout the dissertation wherever possible.

The samples are described by the SAMPLE relation:

Sample	Site	Class	Age
S1	T44	D3	59 to 63
S2	T44	D4	59 to 63
S3	T41	D4 or D6	60 to 67
S4	T41	D6	65 to 67
S5	T41	?	58 to 60
S6	T54	D3	60 to 63
..	...	...	...

In this relation, Sample is the key; the Site column gives the site at which the sample was found; the Class column gives a classification of the sample, which may be imprecise; and the Age column gives the age as a real interval (in millions of years). Some imprecise information is present in this relation.

The sites are described by the SITE relation:

Site	Name	Country	Latitude	Longitude
T44	Delmar	USA	34 to 35	268 to 269
T54	Elsworth	England	52.3	0
T41	Branimir	Bulgaria	42.5	25.1
..	...	...	...	...

Here the Site column is the key, and the other information is fairly self-explanatory. The latitude and longitudes are allowed to be imprecise.

The EQUIPMENT relation is as follows:

Inv-no	Site	Descr	Value
I137	T41	Auger	300
I139	T41	Auger	320
I4290	T41	Concentrator	50
I3054	T44	Typewriter	50
..	...	...	...

The EXPERT relation describes a many-many association between experts and sites. It is as follows:

Expert	Site
Mike	T44
Mike	T56
Ken	T56
John	T41
Hiyan	T44
..	...

Those relations which appear in the conceptual schema will be referred to as 'base' relations. These are the relations which we normally think of as being 'stored', but this term will be avoided, since we will make no assumptions about the Internal Schema, which describes what is actually stored. We assume that the system can insert tuples into base relations, retrieve them and delete them, as necessary. The other relations, defined in terms of the base relations, are called 'derived relations' or 'view relations'.

1.3 Views as first-class objects

The reason for providing the view facility in a DBMS is so that different users (which may include not just persons having ad-hoc interactions with the DBMS, but applications programs) can see the data organised in a way that suits them. This can involve such things as changes of units, and also different logical structure. However the value of such a facility is largely removed if users are then restricted in the operations they can perform. If any application program which accesses the database through an external schema is thereby prevented from performing updates to the database, for example, then the three-level schema architecture is not really having its intended effect.

There are, however, difficulties in permitting updates to be performed on views. For example, in a view where certain attributes are not visible, there is no obvious unique way to deal with the addition of a new entity. What values should be added for the attributes not given by the view? If we have a view containing a projection of the Samples relation, in which the age column is missing, and an attempt is made to add a tuple to this view, some value must be inserted for the age in the base relation.

Even worse, suppose we have a relation giving the country of origin of each sample, formed by joining the Sample relation with the Site relation and projecting out the Site column.

SC

Sample	Type	Age	Name	Country	Lat	Long
S1	D3	59 to 63	Delmar	USA	...	...
S2	D4	59 to 63	Delmar	USA	...	...
S3	D4 or D6	60 to 67	Branimir	Bulgaria	...	...
S4	D6	65 to 67	Branimir	Bulgaria	...	...
S5	?	58 to 60	Branimir	Bulgaria	...	...
S6	D3	60 to 63	Elsworth	England	...	...
..	...	...	...	...	...	...

Then an attempt to add a tuple to this relation is very hard to interpret. If we regard it as valid, it will be necessary to find a dummy value for the site to add into the Sample relation, and add a tuple to the Site relation to show that this site is in the right country. Clearly this is going to result in a great deal of complexity.

On the other hand, if an attempt is made to delete a tuple from a relation which is the join of two relations A and B, then there are three ways that the desired result can be achieved. The tuple to be deleted is the result of joining a tuple from A and a tuple from B, so either the former tuple can be deleted, or the latter, or both. Which of these is the correct interpretation of the users wishes cannot be determined from the information available, and depends on the meaning of the relations and the join, and what the user is trying to do.

There are problems with trying to delete information from the database in general. In a system which simply keeps 'records' that contain information new records can be added and existing records modified or deleted at the user's request. However in a system that stores

information in a more general way, an item of information presented to the user may be the result of combining and manipulating several pieces of information on a lower level. If the user asks for the item to be deleted, he is giving the system one clear piece of information, namely that this item is no longer known to be true, but this is in contradiction with the information already in the database, and it is not clear which items in the database need to be deleted (or amended) to 'delete' the unwanted item. The decision as to how to delete a given item from the database will have to be made on the basis of more information supplied by the schema designer.

This illustrates the problems of moving from a simple filing-system frame of mind, where the database is essentially responsible for the safe storage and fast retrieval of a collection of records, to an information-storage frame of mind. In an information-storage system the addition or alteration of information in one place may have effects in other places, because of inference operations and mappings of information that are taking place. Hence it seems too restrictive to insist, for example, that an update to a view of a database should only be able to alter information that is visible in that view.

There have been various proposals for specifying more semantics with the relational model, which would help resolve the problem of identifying meaningful ways to perform deletion from the database. Some of the best known proposals are [SmSm], and Codd's RM/T model [Codd]. Also Sciore has given a proposal which specifically addresses this problem [Scio]; his treatment is based on specifying to the system which combinations of attributes can represent meaningful 'facts'. This kind of approach enables the database to perform more intelligently, although it is necessary to take care, since different users may have different ideas of the semantics of the database and what constitutes a meaningful fact (and

correspondingly different intentions when deleting).

Various existing DBMSs provide support for views on the database, with some facilities for updating them.

IMS is IBM's main hierarchical database system, and provides a subschema facility mainly of the subsetting type. There is some provision for performing update operations on the data in the subschemas.

Similarly, the CODASYL model of database systems provides for subschemas by subsetting, and data in these can be updated.

System R [SysR] allows a query in SQL, its query language, to be given as the definition of a derived relation. This mechanism is heavily used for control of access and authorisation for update; it is not limited to subsetting and almost any query can be used to define a view. However the only views to which any update operation can be applied are those where the tuples correspond one-to-one with those of a single base relation; hence updatable views cannot be defined using Join, Union or any Project operation that causes elimination of duplicates. This philosophy is based on the discussion in [ChGT].

INGRES, another well-known relational database system, similarly allows the definition of derived relations by queries, and updates on these are supported by 'query modification' on the operation. Insertion and deletion of tuples are not allowed unless the view is a simple restriction of a single existing relation, but operations to **alter** a given tuple can often be translated, even if the tuple is materialised from several base relations. This approach seems quite promising, although for simplicity I have not treated alteration separately, but as a deletion followed by an insertion.

The possibility of performing updates to views of databases has also been considered in a number of papers:

Chamberlin Gray and Traiger, in "Views, authorisation and locking in a relational DBMS" [ChGT], describe the philosophy of views, update and authorisation which was built into System R. Subsequent papers on System R describe essentially the same approach. A view can be defined by any query operation, allowing the user to produce subsets of tables, to join tables together, to perform changes of units, and to produce statistical summaries etc. This is one of the most comprehensive view mechanisms available.

It is used heavily for the control of authorisation for updates to the database. In order to make the view mechanism suitable for controlling update, they specify two rules:

* The uniqueness rule -- an update to a view is permitted only if there is a **unique** operation which can be applied to the base relations that will result in **exactly** the specified changes to the view.

* The rectangle rule -- an update to a view must only affect information visible within the view.

With these rules, it is possible to control a user's update privileges by defining the view which contains all the fields that he is allowed to alter. Thus for example a manager of a department might have a view of the employee relation which contains only those employees in his department, and might have authority to update this view. These rules ensure that the updates that the user performs on his view can have no effects on other information in the database. This is a rather limited

way of controlling authorisation, because it may well be reasonable for a user to perform some action which causes the alteration of certain information as a side-effect, even though he is not allowed to update it directly, or perhaps even to see it. We might well have a situation in which a clerk has the authority to alter an employee's tax code, thereby altering his total pay, although he does not have authority to alter the latter directly. Thus to describe the set of ways in which a given user is allowed to update the database simply by giving a set of fields which he is allowed to alter is not realistic. A more powerful mechanism for authorisation would be desirable, based perhaps on giving a list of the transactions which a user is allowed to invoke. The effect of the above rules is to limit very severely the cases in which users who have views of the database will be able to perform any updates, thus defeating the purpose of the view mechanism.

[FuSS] proposes a rather more liberal approach to permitting updates to views in a relational database. It is slightly harder to follow because it is described in terms of a 'quotient algebra' relational system, where the relations are divided into blocks. This proposal regards neither the rectangle rule nor the uniqueness rule as necessary, observing that updates may well need to have effects on data not actually in the view, and that where there is more than one possible translation of an update the user can supply more information. One rule they do stick to, however, is that the effect of the update on the view relation itself must be exactly the same as if it had been a stored relation. (This will be called the correct-effect rule). Thus an update is forbidden if it would have a side-effect on the view as well as the one originally requested. This seems rather arbitrary once the step has been taken by the authors of discarding the rectangle rule and permitting side-effects on data **not** in the view. It is based on the wish, apparent in several papers on the subject, to make each relation in the system, including

derived relations, behave like a flat file which can be updated in any way. Once we have constraints in the system, this will not even be true of base relations.

[FuSS] goes on to discuss how updates can actually be propagated through relational operations, and points out that insertion through a Project operation can be performed by adding nulls in the missing positions. However the implications of using nulls in this way are not really covered. Simple rules are given for each relational operation. In the next section it is pointed out that these rules do not in fact map the information in all the cases that they should, and to remedy this some further special cases are discussed where a better method can be applied, such as a Join followed by a Project, or a Select followed by a Project. Patching up a few more cases like this by no means ensures that the system will always behave correctly, and will make the system more complex and harder to understand. What is required is a general system for passing updates through relational expressions.

Bancilhon [Banc] defines yet another rule characterising the correctness of view update: this is that any sequence of operations on the view which leaves it unchanged, such as inserting a tuple and then deleting it, will also leave the whole database unchanged. The motivation for this ingenious rule is that when taken in combination with the rectangle rule, it allows the view update to be described entirely by the 'constant complement' of the view. This is another subset of the database which is specified as being unchanged by any operations on the view. The constant complement is also used to characterise view updates in [Spyr].

Klug [Klug] argues very well the case for full support of views, including updates to them and constraints defined on them. He insists that rules such as the rectangle and uniqueness rules are far too restrictive. He seems to argue that the meaning of an update on a view should be specified by the user when he defines the view. Next he develops a mathematical formulation of the question of whether the view definition is 'correct', that is whether constraints and updates are mapped properly. He then investigates whether this correctness can be checked automatically by a schema compiler. Unfortunately, the answer seems often to be 'no', and in any case to depend very closely on the precise features of the system in use; for example the mathematical treatment seems to need to be largely reworked when he adds one new kind of constraint, and the decidability depends crucially on which relational (or other) operations are available.

What is being investigated, then, is support for extra semantic modelling facilities over the relational model, such as constraints of various kinds and richly structured domains, and in particular support for views as first-class objects.

It is debatable to what extent the information in any view of the database can be considered updatable. In a 'statistical' view, for example, containing only information such as the total number of samples, and the average age, but no references to individual samples, it would seem unreasonable to allow an update to the total number of samples. Although an attempt to increase this number by one could be construed as adding a new sample to the database, there is in general no way to map statements in the view back into statements in the base relations, and so such a statistical view cannot really be considered updatable.

It might, however, be remarked that for certain purposes the user might wish to update even such a view as this. If a relation formed by a summarising operation of some kind is subjected to no further processing, then there would be little point in updating it, but in a complex schema there might be many more levels of calculation above it. The user might then want to perform some kind of 'what-if' operation; 'What if the number of samples were to be 5000 and the average time to process them 3.7 hours?' so that he would want the change made in some way to the 'statistical' view, just as if it were a base relation, in order to see the effects on other computations involving it. To take another example, in a database being used for designing something, we might wish to store in advance some information about a part, for example the mass or volume, which when the design is complete will be **computed** from the detailed design. We will not be considering the treatment of such 'hypothetical' operations on views, but will look only at information which can definitely be mapped back to the base relations. It will not be possible in my system to place any information into a view except insofar as it can be made to appear there by updating the base relations.

The problem is to devise a system of sufficient expressive power for the base relations so that updates to views can be translated. In order to be able to translate any update to any view, we would need a system that could store any statement -- this would mean at least a system which could store any assertion in first-order predicate calculus. Such systems have many attractions, but are hard to implement in such a way that they go as fast as conventional database systems. What is sought is some system of less expressive power which will still be reasonably complete, will allow the translation of most updates to views where it seems sensible to do so, and which can be implemented fairly efficiently. This is clearly a compromise, and it will not be possible to translate any view-updating operation perfectly.

The approach to propagating updates through views adopted here is as follows. Rules such as the rectangle rule, the uniqueness rule, the correct-effect rule and the identity-operation rule are too restrictive. The reasons for this have been argued above. When information is added to the system, deductions may occur which cause side-effects to appear in other places. It is therefore essential that the user should understand what he is doing, and that the system should also understand what he is doing. Rather than thinking of his operations as acting on a set of physical records, he must have semantically meaningful operations; enough semantics must be specified to the system that the updates can be correctly propagated.

When translating updates on derived relations the system should not introduce any extraneous (incorrect) information, and should try always to preserve as much of the given information as possible. It is often possible to map all the information present from the view to the base relations, but there are some cases where this cannot directly be done, for one of several reasons. The information implied by the view may not be directly representable in the data model used at all, it may not be representable by the conceptual schema in use, or the translation system in use may not be able to find a translation of the update which has the desired effect. We would like to design the translation system to eliminate the third possibility, and enrich the data model to minimise the first. The second possibility is in the hands of the schema designer. There are many papers on how to design schemas so that the right sets of statements are representable.

Consider the cases for which [FuSS] gives special rules. These essentially fall into two kinds. The first is where a tuple containing a null (missing) value has to be inserted into a view which is the result of a Select operation, where the selection condition restricts the

possible range of the null or missing value. The second is where a tuple to be inserted into a Join has a null value in the joining column(s).

In the first case the selection condition gives us some information about the range of possible values of the 'null'. We would like to be able to record this information, and so we need to be able, for any select expression, to put an entry in a tuple which represents the set of values that the select expression accepts. For example, if we have a view relation defined to consist of all those samples with age between 65 and 70 million years, with the age column projected out,

Sample	Site	Class
S4	T41	D6
..	...	...

and we insert a new sample into it, this would cause the new sample to be added to the selection with null age; we would like the age field in the tuple inserted in the base relation to convey the information that we know the age of this sample to be in the range 65 to 70 million years. Unless we do this, the sample will not appear in the view relation into which we asked to insert it. Thus what we want is some way of representing, in the base relation, the imprecise value (65 to 70 million years) which is implied by the selection condition. This places a requirement on the data model itself, rather than on merely the schema design or the translation algorithm. In practice we will not be able to represent perfectly the information implied by any predicate, but we would like to be able to describe ranges of values such as in the example above.

The kind of selection predicates we are likely to meet include restricting an attribute to a particular value, or to a given range, as above, or to one of some set of values; also selecting those values which satisfy some arithmetical property, or which when looked up in some other relation produce a given value -- these also give rise to a set of possible values. We will consider predicates restricting values to sets or to ranges later as particular cases of our scheme for representing imprecision.

The other class of problem is that of inserting a tuple into a Join, where the entry in the Join column is not precise. This gives rise to two tuples, one in each of the relations joined, with a copy of the imprecise value in each, plus the extra knowledge that these two entries represent the **same** unknown real-world value. We would like to be able to store two separate null entries in the two underlying relations, flagged to indicate that although we know neither precisely, we know that they are equal. One solution to this is the brute-force method: generate entries flagged in precisely this way. This has attractions, and is discussed at some length later. There are some serious objections to using this method. The other possibility is to discard the information that the two values are equal, and this is what has been done in constructing my experimental system, in order to see how far it is a problem. The justification for this is that the Join is regarded as applying to the tuples in one relation a function represented by the other relation, which we can regard for the purposes of this operation as fixed.

When we have a well-controlled system for update with some set of transactions that can be invoked by users, we can regard these transactions as representing the dynamic constraints on the database. Updates will then have some clear conceptual meaning; the operation of

adding a new sample is quite distinct from that of adding a new site. If we wish to add a new sample that was found at a new site, it will be necessary to add the new site first, and then the sample, rather than just adding the sample and thereby introducing the new site. This is why we will often be able to assume that only one relation in a Join is being updated, rather than both at once. This is discussed more fully in the section that describes my Join operation.

Suppose, for example, we wish to add a tuple to relation SC saying that sample S10 was found in country Britain. This will result in an attempt to add a tuple to the join of SAMPLES and SITE with a null in the column on which they are joined. We can attempt to resolve this by finding out from the SITE relation which sites the unknown site could be, in other words inverting the function represented by the Join. If we discover that there are two sites in Britain, then we can store the Sample tuple with the information that S10 was found at one of those two sites (making the assumption that a new site is not also being introduced). This will ensure that the desired tuple appears in the view SC. While this does not represent the given information perfectly, it is the best that can be done given the schema in use, and we need not regard this as a serious problem, since if we wished to be able to record exactly which country a sample came from without knowing the site, we could use a different schema.

We might hope to find that users would only expect to be able to update such a view when the attribute they have in their view is a candidate key; in other words the function they are applying is one-to-one. This is precisely the case in which the above trick will give exactly the right answer.

1.4 Summary

What is proposed is to have a representation for certain kinds of incomplete information. It is clear that this will be useful in many applications anyway, but the motivation here is to enable us to translate statements from views to base relations. The extent to which we can carry out this translation depends on the richness of the model for representing incomplete information. Two kinds naturally arise: incomplete values corresponding to selection predicates, and missing entries which represent the same real-world value. These will be discussed in more detail later. As mentioned above, it was found practical in the experimental system to handle some values of the former kind, but not the latter.

CHAPTER 2

REPRESENTING MISSING INFORMATION

As was pointed out in the previous section, problems arise in the translation of operations on views to base relations when there is information in the base schema that is missing in the view. The approach proposed is to have a mechanism for representing missing information in relations, and this section discusses various ways of doing this, and how it affects various functions in the DBMS. A complete solution to this problem would allow us to

(a) store an entry representing any partial information about a real-world value that can be deduced from a selection predicate

(b) record in some way the information that two entries in relations represent the **same** partially-known real-world value.

This section first describes a simple approach to missing information, the so-called 'null value', which is essentially standard, except for the section on 'merging information'. Similar treatments are described in, for example, Codd [Codd], Date [Dat2], and others. It is seen that catering even for this has many implications. Then an extension to this is discussed which allows us partly to carry out (a) above. In the next chapter the possibility of catering for (b) is discussed, but my conclusion is that it is more practical not to attempt this.

It is generally desirable to have some means of representing missing information in a database system anyway. Most applications will have some area where missing information is likely to arise. When a new entity is added to the database some of its attributes may at that stage not yet be known. A rock sample could be added to the Sample relation before the age has been measured.

The usual method suggested to deal with this is to store a 'null' value for the attribute, which stands for 'value not (yet) known'. There is a distinction between this and another common use of null, to stand for 'attribute not applicable'. This 'inapplicable' type of null might occur if for example there were a column in the Samples relation called 'Species', giving the species for any sample which was in fact a fossil, and null for all other samples. This latter type of 'inapplicable' null does not represent any missing information, and must be clearly distinguished from the other type of null, as the techniques appropriate for handling it may be quite different. It will not concern us further in this dissertation.

The null which stands for 'this value has not yet been made known to the DBMS' is often required in real databases and indeed in other programs, and if not provided by the system is likely to be fudged by the user, by reserving some value which cannot otherwise occur to stand for it. The user of the sample database might decide to use an AGE entry of -1 to stand for 'age unknown'. However this has undesirable side-effects; if the system is asked for the average age of samples, the result will be wrong, for example. In fact many parts of the DBMS need to be designed with missing information in mind, and the null value must be integrated into the system at a deep level.

2.1 Comparisons and three-valued logic

There are many places in the system where values are compared with one another. What should the result be if a value is compared with null? For example, what should the result be if a test 'AGE > 50,000,000' is applied to a sample with null age? The answer cannot be TRUE, as we do not know that the age is greater than 50 million, but equally it cannot be FALSE, since that would assert that the age was less than or equal to 50 million, which is not known to be true either. The answer is that we do not know whether the age is greater than 50 million, and so an unknown truth-value must be the result; a null in the domain of truth-values. There are thus three truth-values: True, False, and 'Maybe'. These three truth-values can be operated on with all the usual functions, such as AND, OR and NOT, with definitions extended so that for example the truth-table for AND is:

AND	T	?	F
T	T	?	F
?	?	?	F
F	F	F	F

If the database system is designed to work with domains that contain an 'unknown' value anyway, then the set of truth-values will fit into this scheme.

2.2 Outer operations

When there is provision in the data model for representing 'unknown', certain modifications to some of the relational operations become desirable. For example, suppose we are joining the Sample relation with the Site relation to get a list of all the samples with their countries of origin. Then if for some sample in the Sample relation, the site number does not appear in the Site relation, there will be no record of that sample in the result at all. This is unsatisfactory; since we have a way of saying 'unknown', it would be preferable for the Join operation to produce a tuple for that sample, with 'unknown' as the country. This modified Join operation is called 'outer Join' and there are outer extensions of various other operations. The precise type of outer Join which I propose is described fully later.

2.3 Merging information in tuples

Normally in a DBMS if an attempt is made to add a new tuple to a relation and it is found that a tuple with the same key already exists in the relation, it is an error and the attempt is rejected. The word ADD here refers to inserting a **new** tuple, rather than modifying an existing one, so there is no intention to discard or replace any existing tuple that has the same key. If a tuple exists already for a given key, an attempt to insert a new (different) tuple for that key will be an error. However when null values can be present in the database, it may no longer be appropriate to reject insertions that match existing keys.

It may be possible to merge the information in the new tuple with that already in the database. For example, in the Sample relation, if there exists a tuple for sample 309 giving the age as 30,000,000 and the site unknown, and an attempt is made to add a new tuple to the relation with the same key, namely SAMPLE = 309, giving the age unknown and the site as D37, then the information in the two can be combined so that the relation finally contains a tuple giving age as 30 million and site as D37. When we merge information in this way we still end up with only one tuple and so the rule that there can be only one tuple with a given key still applies.

The request for insertion of the tuple would only have to be rejected if the information in it could not be merged with that in the existing tuple with the same key, for example if it gave the age as 40 million.

For databases with no unknown values this way of dealing with insertion of new tuples reduces to the rule that a tuple with the same key as an existing one must be rejected unless it is identical in every column with the existing tuple. This is because two tuples with no nulls would have to contradict one another if they were not identical.

2.4 Other kinds of imprecise information

There are various other kinds of imprecise information which we might want to store in a database which have features very similar to those discussed above for unknown values. Two examples are discussed below.

Real numbers with tolerances

Suppose we wished to manipulate real numbers with tolerances attached to them, or, equivalently, ranges of real numbers. For example in the Sample relation the age of a sample might be given by some measuring technique as being in the range 35 to 37 million years. If the system did not provide for storing a value of this kind, the user might be able to build one up for himself, as with null values. For example the upper and lower bounds of the range could be packed together in some way, and the resulting words stored as the age. The user would then have to provide himself with the necessary routines for doing interval arithmetic on values in this format.

However as with null values there are various parts of the database system that would need to understand this representation. For example an attempt to compare the ages of two samples for equality in the system would just compare the patterns bitwise, whereas what is really wanted is a more intelligent comparison that yields a result in three-valued logic. If two samples have ages 35 to 37 million years, and 36 to 38 million years, then an uninformed system would simply regard these as different, whereas the actual ages of the samples could well be the same. Hence, as with the simple null value, it is necessary for the database system to understand the representation of imprecise information in use.

The facility to store and manipulate imprecise real numbers might well be useful in many applications, and also enables us to store the information implied by certain kinds of selection predicate, namely those which restrict the value of some real-valued attribute to a given range.

P-domains

Another kind of incomplete information that it might be useful to be able to store is discussed at some length in [Lips]. Lipski suggests that it would be useful to have a system in which our knowledge about the value of an attribute takes the form of knowing that its value lies in some given set. For example, the type of a sample in the Sample relation could be given as '39 or 40' if the classification were uncertain. The attribute could be given precisely, or specified as being known to belong to some finite list of values, or not known at all. In this case, again, the whole system should know about values of this kind. The user should be able make requests such as

'List all samples that may be of type 39'

or 'List all samples definitely of type 39'

Lipski's example is of recording the blood groups of patients in a medical database, where the information can be recorded that a patient's blood group has been established as being either A or O. We will refer to such a domain as a P-domain, where P might stand for 'possibility'.

This facility also allows us to represent the information represented by some more selection predicates. Any predicate that effectively restricts the value of an attribute to lie in some finite set can be represented.

2.5 Lattice Domains

The above sections describe three kinds of domain with incomplete information structure:

(a) Ordinary domain of values plus 'unknown'

(b) Real number intervals

(c) P-domains

Many of the considerations that apply when handling these domains are very similar; for example, the need for three-valued logic. The treatment of them can be unified by describing them all in the terminology of lattices. This is a very general mathematical structure of which they are all particular cases. The theory of lattices was used by Scott and Strachey in the description of programming languages, and in that area some powerful theorems about lattices could be applied to produce the results they wanted. In this case, the use of lattices merely provides a convenient terminology to talk about these domains. No results are produced from the theory of lattices, and in fact no mathematics is being done in this dissertation. However the language of lattices has been used before [Vass] in discussing nulls in databases (although he did not apply it to the other two kinds of domain) and so it seems worthwhile to use it.

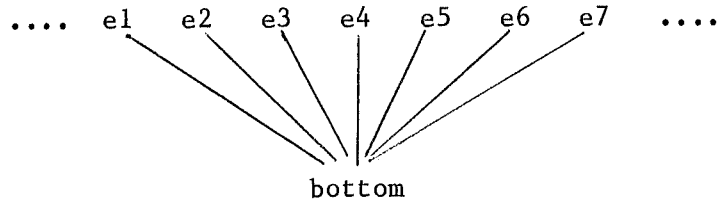
For the purposes of this dissertation we define a lattice domain as a set of elements with a partial ordering, which in this application represents information content. Thus some elements are regarded as giving more information than others. It also has the property that if two elements do not contradict, we can form a third element which contains all the information in both of them. This is called the 'least upper bound'. There exists a 'bottom' element which contains no information, and this

can be regarded as the 'unknown' value in the domain. See for example [Stoy].

The usage here differs from the conventional usage there in that my lattices have no 'top' element, and if the least upper bound of two elements would be 'top' we shall say they are not 'lub-compatible', or that they contradict one another.

We say that an element A approximates another element B if B is a more precisely specified version of A. In other words, B contains all the information in A, plus some more.

In the case (a) of a domain of values with 'unknown', the lattice contains all the values and 'unknown'. 'Unknown' is the bottom element and approximates every other element, but all the other elements contradict one another and so no other element approximates any element. This is called a 'flat lattice'.



In case (b), the real intervals, the lattice consists of every finite real interval, plus the 'interval' (- inf, + inf) which is the bottom element. This element conveys no information about the value of a real number. An interval approximates all those intervals that are subsets of it. Thus for example the element (3.0 to 5.0) approximates the smaller interval (4.0 to 4.1), as the latter is compatible with it but gives a smaller range. This element itself approximates the precise element 4.0 (i.e. the interval (4.0 to 4.0)) which does not approximate anything.

The lattice structure would also be able to model the domain of real intervals including semi-infinite intervals, as these form a lattice. These correspond naturally to selection predicates of the form (Attr > Value) or (Attr < Value).

In case (c), the elements of the lattice are all the possible subsets of the base domain other than the empty set. The approximation ordering is by set inclusion, and the bottom element is the set of every element in the domain. Thus for example over the integers, the subset (39, 42) representing the information that the true value is either 39 or 42 approximates the precise value (39) and also the precise value (42).

Thus in this use of lattices we always have some 'precise' elements which correspond to the 'true' values in the real world, and we have other values which approximate these, representing possible states of incomplete knowledge. The bottom element represents a completely unknown value, and approximates all the other elements of the lattice domain.

It could also be said that the precise values are those which would be acceptable in the primary key of a relation, and indeed this would be the translation of the usual 'relational rule of integrity'. Some discussion of preciseness of keys follows later.

Comparisons in lattices

When comparing two values from a lattice domain for equality, the result will be in three-valued logic, as was explained earlier for the flat lattice. Thus the result will be false, unknown or true depending on whether the precise values represented by the elements cannot be equal, may be equal or must be equal. The true result can only be given when comparing two precise elements. Similar remarks will apply to the other comparisons such as greater than, less than etc.

Arithmetic in lattices

For domains based upon real number or integer values, it is possible to extend the definition of arithmetic operators to cover all elements. Any operation involving the bottom element will give a bottom result, and otherwise the result is the g.l.b. of the possible results of applying the operation to the precise values approximated by the operands. In the case of imprecise real numbers, for example, this gives the usual 'interval arithmetic'.

Merging of information in lattices

When an attempt is made to add a new tuple to a relation whose domains are lattices, suppose an already existing tuple is found with the same key. Then as was explained earlier, an attempt can be made to merge the new tuple with the existing one, by forming the least upper bound, column by column. If in any column the two tuples are not compatible, then the insertion must be rejected.

For example, if there is a tuple giving the age of a rock as being in the range 30 to 35 million years, and an attempt is made to insert a new tuple keyed to the same sample with age field 32 to 40 million years, the system should store the merged value 32 to 35 million years.

2.6 Domains in relational database systems

What is proposed, then, is that the domains in a relational DBMS can be based on lattices. How this is implemented in my system is described later, but it is appropriate here to say something about the role of domains in relational systems. They function in a way very like that of data-types in programming languages, providing extra semantic information which tells us what operations make sense. For example, a relational system with domains should not allow us to compare two values from columns over different domains, or Join on two such columns.

Two papers discussing the definition of domains for relational database systems are [Mcle] and [Robs]. The first of these, 'High level domain definition' by Dennis McLeod, suggests a syntax for a domain-defining language which gives, for each domain, a description of the component elements, specification of any sort-order on them, and details of actions to be taken if errors are encountered. The elements of a domain can be numbers, strings or enumerated sets, or a union of these, and can be restricted by a selection clause. These domains all contain a null value. He goes on to conjecture that it might be possible to have a hierarchic system where domains are defined on top of other domains, and that domains may have their own operations associated with them. He draws the comparison with abstract data types, and points out that his own approach is rather representation-orientated, and that the domains are better characterised by the set of operations available on them.

In [Robs], Mike Robson describes the domain definition features available in the system CODD. There, domains may indeed be defined hierarchically on one another; however the only operations available for forming new domains are 'filters', which restrict the values available. The basic domains are again numbers, strings and enumerated sets, but this time null values are not catered for. The domains are described in terms of routines to recognise values of the domain, and to compare values for ordering. Provision is not made for operations to be defined over domains.

Certainly we will want to provide for primitive domains the sets of operations that apply to them, including their own routines for recognising and printing their values, comparison, arithmetic if applicable, validity testing, and so on. We will also want operations for defining new domains hierarchically on top of other domains.

Questions analogous to those for programming language data-types therefore arise: for example, what types are there, and what operations to form new types? The usual base types for programming languages will carry over: Integers, Real numbers, Character strings, Truth Values; but these will all have to be lattices, so each will (at least) have to contain a null element.

In this section the operations for forming new lattices (i.e. types) will be described. These are of two kinds: operations for constructing compound types, as found in programming languages such as Algol 68, and operations for forming 'imprecise' versions of types, according to a set of 'imprecision operations'.

The usual operations in programming languages to create new types are: forming structures containing several objects of possibly different types, forming arrays or sets of objects of a given type, and forming the union of several types. Ideally operations like these should be provided by the data model in use, and most data models provide at least some of them. The relational model specifies that the objects stored in relations should be atomic, and so forming structures and arrays are not really allowed; these operations can be carried out in terms of relations anyway, for example rather than have a column where the entries would be a pair of real numbers, we should have two columns. The other operations are not so directly provided by the simple relational model, but would be available in more sophisticated semantic data models, such as RM/T or SDM.

These domain-forming operations do, however, need to be considered in the light of the fact that the underlying domains are to be lattices. For example, in a 'united' domain formed from two other domains, what is the bottom element? Each of the domains united will have its own bottom element. The operations have to be considered as lattice-operations, and certainly the theory of lattices gives us some operations for forming new lattices which we can use. These are as follows:

Cross-product of lattices

This forms a lattice where the elements are n-tuples of elements from the underlying lattices. Hence this gives the 'structuring' operation we require (and really the array-forming operation as well.) In the cross-product lattice things such as the bottom element and the l.u.b. of two elements are simply formed column-by-column. Hence if the data-model supplies these operations, as most do, then they will work correctly anyway.

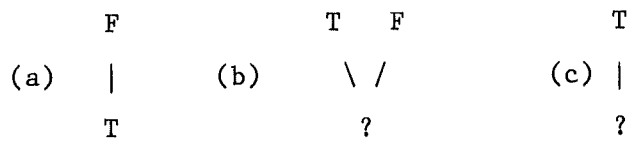
Sum of two lattices

The sum of two lattices is effectively their union, and provides the united domain we want. The theory of lattices offers us two ways of forming the union so as to preserve the lattice structure. We can either unite the two bottom elements of the domains, or keep them separate and create a new bottom element below both of them. These are called the 'coalesced' and 'separated' sums. Thus for example if we are creating a domain which is the union of strings and integers, we can either have a domain which contains all the strings, all the integers, and one 'unknown' null, or we can have a domain containing the strings, the integers, a 'null' standing for 'unknown integer', one for 'unknown string', and one for 'completely unknown'. In either case, the l.u.b. and other operations are performed within the sub-domains, and elements from different sub-domains are not lub-compatible.

Domain of subsets of a lattice

We can form a lattice consisting of elements which are subsets of a given lattice; in other words it corresponds to it in the same way as a powerset does to a set. This gives us the function necessary for the user to form 'set' domains, but some care is necessary. We can define the 'set' domain formally as the lattice of functions from the base lattice into the set membership lattice, and we have to choose the latter with care, in order to get the desired behaviour. The usual choice in lattice theory is the lattice (a) of two elements, True and False, where True is the bottom element, and False is above it. However for this purpose we could use our usual lattice of truth-values (b), with three truth-values. In fact three truth-values are usually an embarrassment here, and we generally use the lattice (c), giving only 'true' and 'maybe'.

With the choice of (b) or (c), we get a lattice where the bottom element is a set where every element has membership 'unknown'; the least upper bound is then formed by taking the l.u.b. element-by-element of the membership function. This models what we want in an imprecisely known set. The distinction between them is that with (b) we can specify those elements which are certainly not in the set.



We must consider what we want the set-forming operation to be. Some languages (such as Pascal) and some high-level data models support the notion of sets of values. In a set-domain, each element is a set of values from the underlying domain; in the example concerning blood groups, a patient might have an attribute 'blood groups accepted' which lies in the domain of sets of blood groups. There are two obvious ways to implement this: as a list of the members of the set, or, where the base domain is small (e.g. blood groups) as a bit-map giving the membership function for each element of the underlying domain.

What about databases that support incomplete information? Date, for example, asks 'the question of whether sets are allowed to contain null values' [Dat2]. Clearly we could decide to support only precisely-known sets. The members of a real-world set are real-world values, i.e. precise values, and hence a precise set will give a function from the precise elements of the underlying domain to the truth-value set {T, F}. It will be implemented as a list of precise values, with the understanding that any values not in the list are definitely not in the set, or as a bitmap giving true or false for each precise element of the underlying domain. (We must additionally provide a bottom element, to

make this a lattice.)

If we decide to allow imprecisely known sets, there is a plethora of ways in which we can weaken our knowledge; there are many convenient ways of allowing a partial description of a set to be given. Hence the immediate answer to a question such as 'If there is a null in a set, does it stand for one unknown value or for possibly many?' is that there are many possible schemes. The information must assign one of the three truth-values to each element of the underlying domain, but could also carry additional information such as 'the cardinality of the set is definitely N' or 'exactly one of the following elements is in the set' and so on.

Considering what we can actually implement reasonably, we consider the two ways mentioned above of representing sets. If we use a bit-map over a small domain, then we can use it to give the truth-values true and false, in which case we are just representing precise sets, or the truth-values true and maybe, in which case we have sets where there may be other members than those known. This corresponds to lattice (c) above. We could use two bit-maps, to permit the assignment of any of the three truth-values to each element, as for lattice (b), but this seems slightly clumsy. Better is to use just one bit-map, plus a 'switch' to allow it to be interpreted in either of the above ways.

If we decide to store a list of the precise values in the set, we have similar arguments: we can have two lists, giving elements that are definitely in the set and those that may be, or just one list and a switch. However it might also be possible to extend the expressive power by allowing imprecise elements to be given in the list. We might have a set of reals where the membership function is given precisely as true or false, but the values are subject to slight imprecision. This raises the question mentioned above, of whether each imprecise value stands for just

one real-world value or can stand for many. If each value stands for one real-world value, then we know the cardinality of the set; this means our list may contain duplicates, which must represent **different** real-world values, which is a slight nuisance. If we permit each imprecise element to stand for zero or more values then we can remove duplicates, but have rather less information. Unfortunately, neither of the above constructions gives rise to a lattice, since it is not always possible to form the l.u.b. of two such sets. This is not merely a mathematical difficulty, but corresponds to a practical problem, which is that when comparing two such sets we cannot tell which element corresponds to which. In order to have a lattice, it is necessary to insist that all the values in each list are 'incompatible' with one another. This means that they cannot stand for the same real-world value, so that they can definitely be distinguished. With this adjustment, either of the above schemes becomes tenable.

Date [Dat2] proposed a scheme called 'N-sets', which are sets that contain precise values, and may also contain one 'null' i.e. bottom element. If the null is present, it may stand for an arbitrary number of other values. Hence this scheme is in fact identical to the one mentioned above, where there is a list (or bit-map) of precise values only, plus a switch to say whether other values may be present. In Date's scheme, the switch is represented by the presence or absence of the null in the list.

Storing an imprecisely known set may seem a rather unreasonable thing to do, and indeed it is not being suggested here that this should be done; the point is rather that **if** an advanced data-model is provided which supports sets for the user, and imprecise information is to be supported, the choice of how to allow sets to be approximated needs to be made carefully with an understanding of the principles involved.

The discussion above mentioned the lattice (a) with false above true, for forming 'set' domains. This may seem strange, since true is not normally an approximation to false. What we get when we form the lattice of functions into the lattice (a) is a lattice where the bottom element is the set containing every value, and the least upper bound of two subsets is their intersection. This models exactly what we want for the P-domains described earlier. This is where an entry stands not for a set of values in the real world, but for one value which is known imprecisely in such a way that its value is restricted to be in a given set. Thus if we know nothing about it, the set of values it can take on is the whole domain, and if we have two items of information describing it, we merge them by taking their intersection. This distinction must be made clear: if we refer to Lispki's example of this kind of domain, we might have a record for a patient in a medical database giving his blood-group as a set of possibilities over the enumerated set (O, A, B, AB); for example a given patient might have had a test establishing that his blood group was either O or A, so the system could record (O or A) for him. This stands for **one** unknown value. On the other hand, he might also have an attribute 'set of blood groups accepted', for example (O, A, AB), and this attribute describes a value which is a **set** in the real world -- this set might itself be imprecisely known.

This brings us to the subject of another reason for forming new lattices from old -- that is in order to represent certain kinds of imprecision. Just as we can create new lattices for operations corresponding to programming-language type-operations such as structuring and uniting, we can also form lattices describing imprecise knowledge of values over 'precise' (i.e. flat) lattices. One such is the operation described above, for forming the P-domain over a flat lattice; the other useful one, corresponding to the case of ranges of real numbers described earlier, is to form the lattice of intervals over an ordered flat

lattice. Both of these operations take a flat lattice and give us a more richly structured lattice with many approximations to each precise element.

Thus our base types are flat lattices, containing precise values plus null. The data model may provide structural operations on these, to create more types such as set or array types, and we also have domain operations which introduce the representation of forms of imprecision into the system, such as representing P-domains, or representing intervals.

2.7 Summary

Domains for a relational database system to contain imprecise values can be modelled on lattices; this will cover both the well-known "null value" and other forms of imprecise information. We can provide the same kind of data-structuring facilities for lattice domains as are found in typed programming languages, but this level of structuring is subordinate to the facilities provided by the data model itself. Although in the relational model these may be few, in a more semantic data model there are likely to be found facilities of aggregation and generalisation (see [SmSm]) which themselves play the part of 'structured' and 'united' modes in an Algol-like language. Hence in this case it is not necessary to provide these structuring operations on domains as well, but rather the implementation of the semantic data model must have regard for the fact that the domains are lattices. The theory of lattices supplies us with the necessary operations. We can also use operations on the lattices to introduce the various forms of imprecision that we wish to store in the system.

CHAPTER 3

OTHER APPROACHES TO MISSING INFORMATION

Codd's paper [Codd] describes a simple scheme for handling nulls, roughly equivalent to the treatment of flat lattices here.

Vassiliou [Vass] considers 'imperfect information' in database systems, and attempts to treat both missing information and inconsistent (self-contradictory) information in the same system, adopting an approach based on flat lattices of values. It is true that inconsistent values can be fitted nicely into the lattice framework, but it is not clear whether it is useful to try to maintain in this way a self-contradictory database. At any rate, it is not a treatment of missing information, so will not be discussed further here. Although he uses flat lattices, he does not go on to consider other possible lattices of imprecise information. His work relates mainly to the definition of functional dependency, and the

effect of nulls on the universal relation assumption and database design.

Vassiliou does also consider query evaluation, but rejects the usual three-valued logic approach on the grounds that not all the tautologies of two-valued logic are tautologies of three-valued logic, and instead provides a more complicated algorithm which will return the correct answer more often. It might be useful to include this algorithm in a system supporting nulls (although I have not done this in my system) especially if the processing, which is of exponential order, could be done once when 'compiling' a query that might then be used many times. It should however be noted that no such system can always give the correct answer, in the general case. A simple example of the problem is a select-expression such as

select SAMPLE **where** (AGE>40.0) **or** (AGE<50.0)

which is tautologically true. If the age of a sample is unknown, the simple truth-functional algorithm will evaluate the selection condition as (maybe or maybe) which gives a 'maybe' result although the result should be 'true'. The algorithm given by Vassiliou, and similar algorithms elsewhere, will detect that this condition is tautologically true. It is not however possible in principle to do this for any condition: if we have a condition on three integer-valued attributes I,J,K of the form

(I>0) & (J>0) & (K>0) & (I**3 = J**3 + K**3)

and each of I,J, and K is unknown, it may well be that this condition is tautologically false (if Fermat's Conjecture holds) but no simplification algorithm will be able to arrive at this result.

Lipski [Lips] also avoids the truth-functional three-valued logic approach to nulls, on the same grounds, and gives a complicated algorithm to get better answers. He considers what we have called P-domains; that is where an imprecise value can be completely unknown, or restricted to a

finite set of possible values. He treats this case fairly fully, but does not use lattice terminology, or consider other lattices. He describes two possible interpretations of queries on incomplete databases, which he calls the 'external interpretation' and the 'internal interpretation'. The difference between these is that in the former, queries are understood to be asking about the real world, whereas in the latter they are about the state of knowledge of the database. He gives the opinion that the former, being more natural, is more suited to naive users, whereas the latter, being more powerful, is better for sophisticated users. An example might be: if the user asks 'what is the total population of all cities', the result in the external model might be 'not known', meaning that the system does not believe it knows about all cities, whereas in the internal model it might be '123,000', where this is the total of all cities that the database does know about. Clearly neither of these answers is at all satisfactory without any explanation. My belief is that queries should be understood in the external model unless operations are used which explicitly request information about the state of the database; my experimental program would reply to the above query to the effect that it does not know about all cities (unless it has been told that it does), but it would be possible to specify that the set of all **known** cities was intended. Blurring the distinction between these interpretations gives rise to many of the 'anomalies' of incomplete information.

Biskup [Bisk] describes null values in a mathematical formalism. The approach is essentially the same as in most other papers on this, except that he includes 'universally quantified' nulls as well as the usual 'existentially quantified' type, which means that the tuple containing the null is true for **every** possible value of this null symbol, rather than for some unknown value of it. While this may be a useful shorthand, allowing one tuple to stand for many, it is not a representation of

missing information and so will not concern us further here. He also describes briefly another system which he calls 'indexed nulls', a system where there is an unlimited supply of different null symbols that can be used to stand for different unknown values. In such a system it can be recognised when two null entries represent the same real-world value, which solves a number of problems (this is discussed further later). He criticises this approach on various grounds; firstly, that it will be difficult to check and enforce global consistency, and indeed that any global operations on the database will be very difficult when these 'cross-referenced' values are present; secondly, that there is a danger that with certain kinds of operations, semantic problems can arise, for example an indexed null which is supposed to stand for one distinct unknown value in the real world, might end up playing the role of many. This approach was considered for the experimental system, and various problems were found tending to confirm Biskup's criticisms, of which the following is an example: If we attempt to add a tuple to the join of the samples and experts relations,

SAMPLE	SITE	EXPERT
101	?	Mike

then using this method, we might create a new unknown site 'alpha' and add (101, alpha) to Samples and (alpha, Mike) to experts. This will cause exactly the desired tuple to arise in the Join. However the symbol 'alpha' is intended to stand for some unknown site, and there is no site for which Mike is the only expert; in other words this **cannot** consistently be the Join, for any existing site alpha. If we think we know about all sites this is a problem, as it forces the assumption that 'alpha' is some new site for which Mike is the only expert. Of course, what we probably meant when we inserted the tuple is that Mike is an

expert on that site, and we would be pleased if the system could fill in for itself who the others are or may be. To put it another way, there is a multi-valued dependence of expert upon site, and unless we are careful we can create an indexed null which cannot satisfy it.

Imielinski and Lispki [ImLi] give a mature survey of some different ways of representing incomplete information. They state as their criterion of correctness that a system should not allow any incorrect conclusion to be derived, and that every correct conclusion can be derived. These conditions depend critically on the set of operations provided. The latter condition, which seems desirable, is almost impossible to satisfy in a realistic system, as is explained below.

They consider first a system with one null symbol, equivalent to the flat lattice case discussed above. They show that this is correct for the operations of project and select and union. They point out, however, that when the Join operation is included some correct statements may not be derivable. This is because there may be two nulls which in fact represent the 'same' unknown value, and thus should be matched in the join, but will not be matched because the system simply considers them as two nulls.

They therefore go on to consider indexed nulls. In this system, it can be recognised when two different occurrences of the null symbol actually refer to the same unknown value, and so the above problem does not arise. Such a system is a reasonable candidate for a real implementation. It is correct for the operations of Project, Join, Union and positive Selection. (Some expressions involving selection with negative conditions cannot be produced). It has the advantages that the symbols can be manipulated just like regular values, and are easy to interpret intuitively. The problem of generating new null symbols as required

might not be insuperable; a relational system might have the entries in relations as 32-bit identifiers pointing to the actual values in some kind of value-set storage system, in which case perhaps all those with the top bit as '1' could be reserved for nulls, allowing 2**31 different nulls to be generated and 2**31 real symbols. There might be valid solutions to the problems raised above by Biskup concerning this kind of system; however working with a null value which generalises towards partly-known values, such as ranges of integers and real numbers with tolerances, rather than towards 'distinguished nulls', seems more likely to be useful in practice. An system to allow both kinds of null value would be theoretically possible, but would seem insupportably complicated.

They go on to consider a third possibility, which is mainly for theoretical argument, although it could be implemented. This is a system of indexed nulls where each tuple in the database also has associated with it a predicate making a statement about the values in the tuple. Manipulating such a system would be rather more expensive, but it is correct for all the operations they consider: Project, Select, Union and Join.

One of the most important recent papers on the subject of null values in database systems is [Dat2]. In this paper Date describes a method of handling 'null' in database systems which is again similar to the particular case of the flat lattice in my treatment above. He states that this 'is as reasonable as any that has been given in the literature.' He then goes on to describe some of the problems that arise in following this approach, and concludes that it is not a good idea to support nulls. He proposes instead a scheme which he calls 'default values'. Since Date concludes that the accumulation of the problems he describes is so great as to make the implementation of any scheme to

handle nulls inadvisable, we will consider his arguments in some detail.

He states that his objections to null values are caused largely by ramifications of the fact that null values cannot be assumed to be equal, and so the tautology 'X=X' is no longer necessarily true if X 'takes the value Null'. He proposes a scheme which he calls 'default' values, which act rather like nulls except that they are regular values, and so do compare equal. He describes this as 'less ambitious but more straightforward'. It is argued here that this approach, by refusing to get entangled in the difficulties surrounding nulls, forces the user to do this for himself instead, with much greater chance of disaster.

In his scheme there is a default value associated with each domain; this value is inserted by the system whenever a value is missing. Subsequently it is processed just like any other value. The word 'default' suggests choosing a 'sensible' value, perhaps corresponding to the most typical or likely value for an attribute from that domain. However this is not what the user will usually have to do, as he will need to be able to recognise the values that have been put in on his behalf, in order to be able specifically to take them out again. Date gives the example of computing an average, with the query

```
select AVG (SP.QTY)
 from SP
 where QTY NOT EQUAL TO DEFAULT(SP.QTY)
```

in order not to have the average perturbed by the default values. Clearly this will only work if the default chosen is a value that can never arise in practice. Again, in the definition of functional dependency that Date suggests in his scheme, the default value is specifically excepted and this could cause real problems if this value could ever occur naturally. Hence the user is left with the task of

choosing some value that can never occur. If he is so unlucky that there is no such value, then there is little he can do. Date criticises nulls on the grounds that the system has to decide between implementing them by finding an impossible value or by a hidden field to indicate null; at least this choice is open to the system, whereas if the task is thrust upon the user, he has only the former option.

Since the default value functions as a real value, the user must be aware that test for equality will succeed. Hence for example in a Join of two relations, tuples may appear which were matched because both contained the default value.

Date points out that there are some problems with the definition of Functional Dependency when nulls are present. As he mentions, there are several formal papers which cover this problem; the definition of functional dependency needs to be altered slightly (and indeed it is altered in the same way in his default scheme.)

One real problem however is pointed out, in the procedure for normalising a relational schema. This is that in the world of relations without nulls, when we find certain functional dependencies in relations, we can decompose them into smaller relations with exactly the same expressive power; this is normalisation. However if nulls can be present then the decomposed schema has slightly less expressive power. The decomposition may still be desirable, but it is no longer quite the same process mathematically, as the schema when decomposed is not equivalent to the original one; it will be necessary to decide in some other way whether this decomposition should be made or not. For example, in the SAMPLES relation, suppose we had a column giving the country in which each sample was found. Then we could store the statement 'sample 109 was found at some site in Britain' but Country is functionally dependent on Site, and

so we have split off this attribute into the SITE relation; now we cannot store the above statement. This is the problem that indexed nulls solve; with them we could have in the SAMPLES relation

'sample 109 was found at site alpha'

and in the SITE relation

'site alpha is in Britain'

where 'alpha' represents some unknown site.

We simply have to decide whether we wish to be able to store the above information or not; whether it is a 'meaningful fact' in the terms of Sciore and others. If not, then the decomposition can be carried out.

Date also objects to null values on the grounds that tautologies of two-valued logic are not preserved in three-valued logic. For example, if we ask for a list of all samples of age less than or equal to 50 million, and another list of all samples of age more than 50 million, we might expect to cover all the samples; however samples of indeterminate age will not appear. This problem, which is really the problem of what to do with the 'maybe' truth-value, can be approached in a number of ways, none of them entirely satisfactory, but between them probably sufficient in practice. First of all, expressions which should really be tautologies can often be recognised as such. [Vass] gives an algorithm for this, as mentioned above. Then such constructions as

IF (x GT 50) or (x LE 50) THEN....

will not give 'maybe' for unknown x but 'true'. Also, the user can be given the facility to request that if a test is applied to a truth-value, as above, and it IS maybe, then a warning can be given. Finally, the problem of Select operations where some tuples 'maybe' satisfy the selection criterion can, if desired, be addressed by extending the membership of tuples in relations to three-valued; a computed relation might then contain some 'definite' tuples and some 'maybe' tuples. This possibility is discussed in a later section.

Date points out that when we ask for a list of all the tuples in a relation sorted in order of some attribute, for example asking for employees in order of salary, then it will be necessary to sort the null values into some position in the listing for convenience, and this appears to contradict the principle that we cannot obtain a non-null result when performing an order comparison such as X > Y with a null. However as he admits, sorting the tuples into order for processing and comparing values are distinct operations, and there is no reason why the null values cannot be sorted into some position for processing. If the user really asks for employees in salary order, the system can give an error or warning message if it finds a null salary. A rather worse version of the same problem occurs in his scheme, namely that all the employees with default salary (zero) will appear at the top of the listing, implying that they are the lowest paid, when this is not the case. Since he is treating the default value as a regular value, this could not even be trapped, as the system will not know it is not the true salary. Hence if for example the user were to be looking at a view giving employees in salary order, but with the actual salaries not shown (projected out) he really could be misled.

He states that we need additional operators, including in particular the outer equi-join, and goes on to show that this operation is no longer properly associative as he defines it, which is a nuisance. Also the outer natural join is not, as one might expect, a projection of it. It can however be argued that this is a consequence of the fact that the 'outer equi-join' he defines is a very odd operation, and not very useful. These objections do not arise if we stick to the outer natural join. Also note that if we do want the outer equi-join, the same problems arise using his 'default' mechanism.

He says that since we do not accept the identity 'null=null', the elimination of duplicate tuples from relations is hard to justify intuitively (although it is clear formally). It seems intuitively clear enough to me that there is no point (although perhaps little harm) in storing two copies of the same statement.

He goes on to describe 'implementation anomalies', all of which arise from traumas suffered with System R. The only one which is likely to be a problem in systems in general is that programming languages do not include the notion of unknown values, and hence the translation of the desired concepts into, in his case, PL/1 is ghastly. The solution to this would have to be the extension of the programming language in use to contain lattice domains, which would be helpful in itself anyway. At the moment the database operations in the language are handled by passing the source through a pre-processor which translates the embedded database-language statements into valid PL/1. This could be extended to cover translating operations which refer to variables from lattice domains into corresponding standard PL/1, if it were desired to keep this kind of interface.

He also describes some problems with the aggregate operations such as SUM, AVERAGE and COUNT in System R. The definitions of these operations are simply wrong; they can be correctly defined so that no anomalies such as he describes arise (see section 5.5). However, in his 'default' scheme there are going to be more problems with aggregates. He points out that in this case the user would have to take the trouble specifically to exclude tuples with default values from aggregates, since otherwise incorrect values would be obtained, and the system could not recognise them for him. If the user failed to realise this, he could find that the SUM value given for salaries was rather less than the actual wage-bill, because of the 'default' zero entries present.

CHAPTER 4

VARIOUS SEMANTIC ISSUES

We have suggested a style of 'information management system' that involves support for complex views of the database, constraints and updates on those views, and treatment of partial information. In the light of these requirements, certain aspects of our data modelling must be looked at again carefully.

4.1 The negative information problem

A problem which arises even in databases that have no concept of null is that of negative information: how to interpret the absence of certain information from the database. In the relational model, for example, how do we interpret the absence of a tuple from a relation? Or to put it another way, what is the truth-value assigned to the statement that it represents? Sometimes, no information is implied about that statement, and so it might be true or false; in fact the truth-value is 'maybe'. This is called the open-world assumption. For example, a relation giving the populations of cities might not be exhaustive, and so if a city did not appear, it would not imply that that city did not exist. More often, however, we have complete information about the enterprise that the database describes, and so we have the closed-world assumption, that any tuple not in the relation is definitely false.

Closed-world relations are more useful: for example, we must be careful in performing any kind of aggregate operation on an open-world relation, since there may be true tuples that we do not have stored. We cannot sensibly ask for the total population of all cities, or even the total number of cities in the world, from our open-world relation giving the populations of cities. For the system to reply to this query by giving the total for the cities that it happens to know about would violate the principle that the answer given should be an approximation to the true answer. If the user has explicitly asked for the total of the cities **in the database** then it might reasonably be claimed that that is what he wants, but as we move to higher-level query languages, the query will be more like:

"What is the total population of all cities?"

and the user's thoughts should be moving in the direction of asking questions about the real world, rather than about the database. Therefore, because this is an open-world relation, a warning must at least be given that the question asked is not the one being answered. He can reasonably ask for the total number of samples that we have, since the Samples relation will be closed-world (we know about all samples).

Also, we cannot answer negative questions from an open-world relation, because we cannot in general tell that a given tuple is not true. This means we cannot, for example, compute the set-difference operation if the relation to be subtracted is open.

It is not a facility generally provided in database systems to allow the user to specify whether a given relation is to be regarded as closed- or open-world. Enen in a system without nulls, it would be necessary for the system to know whether the open or closed-world assumption is in force, as it can affect the definition of some relational operators, such as set difference, and it would be a useful piece of semantic information to prevent misunderstandings such as performing aggregate operations on open-world relations.

In a system that represents unknown values, more care is necessary. The definition of closed-world is that any statement that is not known from the relation to be true, must be false. If a relation is to be closed-world in that sense, every true tuple must be present in the relation. In particular, no imprecise values can be present in the relation; each true tuple must be represented precisely, and those are the only tuples present. Thus the system should know which relations are to be closed-world, and forbid **any** imprecise values in those. From here on, this type

of relation will be referred to as 'fully closed', because it is useful to define a slightly more relaxed version of the closed-world assumption.

The most important point about the closed-world assumption is that we have a tuple for each entity in the universe of discourse; for example in a closed-world employee relation, we Know that there is a tuple for each employee. It is unnecessarily restrictive to insist that there should be no imprecision at all in the relation in order to achieve this. We can define our 'closed-world' relation to mean that there is a tuple for each entity, but some of the non-key columns may not be precise. What this means is that any tuple representing a true statement will be present in the relation, but perhaps with some entries 'weakened'. Any tuple that is not approximated by a tuple in the relation is known to be false. Since we will not permit imprecise values in key columns in this kind of relation, we are assured of a one-to-one correspondence between our tuples and the 'true' tuples representing a state of perfect information; we can therefore meaningfully perform operations such as counting and summation, and form set differences.

This kind of relation semantics, in which we can have imprecise values in non-key columns, but still preserve the useful one-to-one correspondence between the entities in the Universe of Discourse and the tuples in the relation, seems to satisfy all our requirements. Why then will we ever want to have open-world relations? The answer is that apart from the fact that we may occasionally want to store information about something without covering it exhaustively, as with cities in the world, we will also generate open-world relations whenever we perform operations in the database. If we perform Joining, for example, on two relations, then even if they are both closed-world, as defined above, the result will be open-world if there are any missing values in the join columns, because matching cannot be guaranteed. Also, when a Select is performed, if the

result of testing the selection criterion on a tuple is 'maybe', it will not be included in the result, as it is not known to be correct. Hence some tuples that are entitled to be selected may be rejected, and thus the result relation is open-world. All the tuples in it are known to be correct, but some that are not in it may also be correct. Hence some of the computed relations that appear in views, or that occur as intermediate results in the evaluation of more complicated relational expressions, will be open-world.

One possible way of improving this situation slightly is to allow computed relations to consist of two sets of tuples, those that are definitely in the relation and those that 'may' be in the relation. What this really means is converting the membership function of relations to be three-valued instead of two-valued. Then when evaluating expressions such as the above, no tuples need to be discarded, and a certain amount of 'negative information' can be retained; any tuple not found in the sets for 'true' or 'maybe' can definitely be said to be false, approximate values of aggregate functions can be calculated, and so on.

What this would mean would be a rather different kind of interface for specifying queries on the imprecise database. Instead of building relational expressions out of operations such as maybe-join or true-join, maybe-select or true-select, the query is expressed in terms of Join, Select etc, and the result is a relation containing maybe-tuples and true-tuples. Then at the end the user can request to see just the definite tuples, or the maybe-tuples as well. This gives a rather more high-level interface, where the user can ask his question about his Universe of Discourse, rather than about the state of the database; he describes the information he would like, assuming perfect knowledge, and the system describes to him how good an approximation it can give him to what he asked for. In such a system, for example, we would not name two

separate derived relations for (a) samples definitely over 50 million years old and (b) samples which may be over 50 million years old; we would simply name the set of samples over 50 million years old, and the database could answer questions about which samples definitely are, or may be, in this set.

Such a system would preclude some operations that might otherwise be possible, such as selecting on two conditions which have different levels of certainty (e.g. all samples that have site definitely equal to T44, and age maybe equal to 50 million). Such an operation would have to be requested in terms of more explicit operators making it clear that tuples were being promoted from 'maybe' to 'true', or vice-versa.

It might also be felt that for many purposes the maybe-tuples are merely an encumbrance, and not worth computing unless specifically asked for. In this case the above approach might be discarded on grounds of efficiency.

4.2 Open-world relations and keys

In an open-world relation, the tuples describe some subset of the entities in the Universe of Discourse; the essential one-to-one correspondence between tuples and entities is not present. In this case it can be argued that there is no reason to insist any longer on the rule of 'relational integrity', that the key fields of each tuple must be precise. We could consider permitting tuples with key columns that are imprecisely specified.

What would such a tuple signify? The meaning would have to correspond to an 'existentially quantified statement'; that is, it would be of the form

'There exists some entity with these properties'.

One argument for allowing tuples with nulls in key fields is that they arise naturally anyway in connection with Project operations that remove all or part of the key of a relation. This is the kind of projection which causes elimination of redundant tuples. The resulting derived relation will have its own key; if the base relation was in third normal form, this will be the whole of the result relation. When we perform such a projection some of the entries in the key columns of the result relation may well be null. Ways that this has been dealt with in the past include ignoring the fact that computed relations have keys (which any relation must), or saying that the integrity rule does not apply to them. Tuples with null values in key fields could be discarded, but it is very hard to find any rationale for doing this. Hence, we end up with a view relation that has nulls in some of its key fields.

Nulls can also be made to appear in the key of the **base** relation, if we permit insertions of tuples into the view. This operation is commonly regarded as dubious in the extreme, as we are updating a view in which the original key is no longer present. However, the view relation presumably has meaning to the user, in virtue of the fact that it has a key of its own. The entities in it are in some sense 'aggregate' entities or 'associative' entities; hence it is arguable that it is reasonable to allow the view to be updated.

For example, consider the EQUIPMENT relation. This is classified as open-world in this example, since it is not clear that every piece of apparatus or equipment will always be recorded -- it is not even clear exactly what constitutes a 'piece of equipment'. The key of this relation is 'inventory-number', a key which has been assigned arbitrarily so that this relation can be used to keep track of certain pieces of gear, to know where they are at any time. To somebody in charge of this task, the descriptive information may be of no interest. However somebody who needs to allocate or use the facilities of certain kinds of equipment might simply want to know which facilities are available at which sites. They might well then use a view of this relation with the inventory number and the value projected out. This will incidentally remove duplicate tuples if there is more than one of the given item at a given site. Each tuple will mean 'there is an X available at site Y'. This user might then wish to record the availability of certain facilities at certain sites without worrying about assigning (or without having the authority to assign, or knowing the system for assigning) an inventory number for the item. He would simply wish to insert a statement such as the above into his view; we will have to propagate this to the underlying relation.

If we wish to do so, then we can easily map the statement back to the base relation, as it corresponds exactly in meaning to a tuple with an imprecise value in the key position. The rule that missing columns in projections are filled in with nulls will then apply uniformly, even if the missing column is a key column; no different action is required. The above example could alternatively be handled by allowing the database system to assign a new inventory-number to any item added in the view where inventory numbers are not seen, but this may not be convenient; the inventory-number may have to be assigned in some way that the DBMS cannot carry out; in any case, this is not necessary, since creating a

tuple with a null in the inventory-number key field does exactly what is required. Permitting imprecise values in key fields in this way seems to offer an extension to the expressive power of the relations entirely free, as no extra complexity is required if we decide to permit it.

4.3 Checking constraints

One of the more important things that a database system needs to be able to do is to store and maintain constraints on the data. These constraints give extra semantic information to the system, and are used in two ways: the database system must ensure that changes to the database do not violate the constraints, and the system must also **use** the constraints in organising itself; for example a constraint on the range of values an item can take might be used in deciding how to store it.

How do we check constraints on a database that contains imprecise information? A constraint is a function on the database that evaluates to a truth-value, saying whether the constraint is satisfied or not. We can evaluate such an expression when imprecise values are involved, and the result will be in three-valued logic. For example, if there is a constraint 0 < AGE < 100 million, then if a tuple has age=(50 to 55 million) this gives result TRUE, whereas a tuple with age=(90 to 110 million) will give result MAYBE.

Clearly, if the result of checking the constraint is TRUE, then there is no problem; similarly if the result is FALSE, the database is clearly in an invalid state, and action will need to be taken, just as in any other system. What, however, if the result is MAYBE?

What this signifies is that the stored data is imprecise, in such a way that it describes some possible states that satisfy the constraint and some that do not. Ideally, what the system should do is to eliminate the possibilities that describe invalid states, by increasing the precision of the stored data according to the constraint rule. Thus in the example above, when there is a constraint on samples that the age must be between 0 and 100 million, if the system finds a sample with age attribute (90 to 110 million), it should change this to (90 to 100 million).

This may seem a rather dangerous step, altering the database to make it satisfy the constraint. We might say, 'if the constraint evaluates to MAYBE, does that not mean that we don't know whether the constraint is satisfied?' The answer to this is that although the 'true' values in the real world are not precisely known, we **do** know that they satisfy the constraint, because this must always be true -- that is what a constraint is. The values stored in the database, on the other hand, are at least perfectly known to us. Hence there is no uncertainty about the status of the constraint. We can use it to refine the stored data, just as constraints are used in determining other aspects of the behaviour of the database. What the MAYBE result to constraint evaluation tells us is that the stored values are compatible with the constraint, but represent certain possibilities that can be discarded.

A distinction must be made clear here between this use of the word 'constraint', meaning something which is always true in the universe of discourse, and another possible interpretation, meaning a rule which the system is intended to enforce. The latter could be dealt with by something like a 'trigger', as available in some systems. For example, in an airline booking system, there might be a rule that the total weight of all packages booked on a freight aeroplane must not exceed the capacity of the plane. This is not a constraint in my sense, since there

is no implication that the total weight booked **cannot** exceed the limit; it is a trigger, to take some action if it **does** exceed the limit. In this case if some package has imprecisely known weight, the desired action would be to issue a warning, rather than to make the 'deduction' that the package must be sufficiently light 'because the constraint says so'.

Given the above distinction, what is proposed is that when constraints are checked, a MAYBE result should initiate some action which will refine the stored values to make the constraint TRUE. How this can be done will of course depend on what form the constraint takes. The constraint-enforcement component of my experimental program will be described later.

CHAPTER 5

IMPLEMENTATION

This chapter describes the implementation of an experimental program which allows the user to define base relations over lattice domains, define derived relations in terms of these, and apply update and constraint operations to all relations.

The program consists of about 4000 lines of BCPL. As well as the standard run-time facilities it uses a set of indexed access-method routines for operations on stored relations.

The program allows the user to define a set of base relations, specifying the domain of each column and giving the key of each relation. Derived relations can then be defined in terms of relations already defined, using relational operations: Project, Select, Join, Group-by, and

Computed Column operations are supported. These are described in detail later in this chapter. The user can then add tuples to any defined relation, or delete tuples, and operations on derived relations are propagated to the base relations. The contents of relations can be displayed.

The user can also specify a constraint on the database in the form of a predicate applied to any relation. Since the relation can be a derived relation, this can express constraints which link several base relations. The constraint is stored in the form given, that is, expressed in terms of the derived relation; it is not translated into a constraint on the base relations, since this would in general be much more complex. Thus the derived relation mechanism is used to express complex constraints in a simple way. How the constraints are enforced is described later.

The program consists of the following sections:

* a simple value-set system for storing and retrieving variable-length data items in the database by pointers

* routines for maintaining and indexing stored relations over lattice domains, described below

* simple cataloguing routines

* routines for handling predicates: reading, writing, intersecting etc. A description of predicates is given later

* routines describing the relational operations available: how each transforms predicates, and in terms of this, how to evaluate it and transform updates through it

* routines to read in, print out, and evaluate arithmetic expressions over lattice domains

* routines to read, analyse and carry out commands

5.1 Managing lattices

To store relations containing entries which are elements of lattices, we must be able to sort these so as to perform retrieval operations on them efficiently. In my system, each element has a one-word representation, which could be either some packing of it, or a pointer into the value-set storage mechanism. Sorting the tuples in order of this representation is unlikely to be helpful with retrieval, except for the simplest query 'find all tuples containing the value X'. In fact this problem will arise with any system using a value-set mechanism, quite independently of the question of lattice domains. Where lattice domains affect the issue, however, is in what kinds of query are likely to occur.

In a system with no imprecise values, the kinds of retrieval request likely to occur are

(a) most commonly: find all tuples where attr=X

(b) for an attribute in an ordered domain: find all tuples with attr in range (X < attr < Y)

The former can be satisfied by any well-defined sort order; for the latter, we want the target tuples to be clustered together, so the tuples

need to be sorted in the ordering in the domain of the attribute in question. This may well not be the same as the ordering of the representation; for example, if we are storing variable-length strings in a hashed value-set store, the lexical order of the strings will probably not be preserved in the hash keys stored in the relation. Hence we must either give up the power to perform retrieval (b), or refer to the value-set store for the real value whenever we are searching for a value or searching for the place to insert one.

Essentially the same considerations apply when considering the sorting of elements from lattice domains. The retrieval requests that can arise are, however, more varied. We can request records with attribute X such that:

(a) X is approximated by some lattice element A

(b) X is lub-compatible with some element A

where A can be a precise or imprecise element of the lattice.

If A represents a precise value then these operations have the effect of retrieving tuples where:

(a) X is definitely equal to A

(b) X may be equal to A

There is however another possible way of using these two requests, which is to perform retrieval of tuples where attribute X falls in a given range (where the domain of X has some intrinsic sort order). We can do this by using the value A to describe the range, which is a slight twisting of the use of the lattice. Thus if we want all samples where the age lies in the range 30 to 40 million then we can use the lattice element (30 to 40 million), which normally represents imprecise

information, to describe this range. Using the value A in this way, our two requests now have the effects of retrieving tuples where:

(a) X definitely lies in the range given by A

(b) X may lie in the range given by A

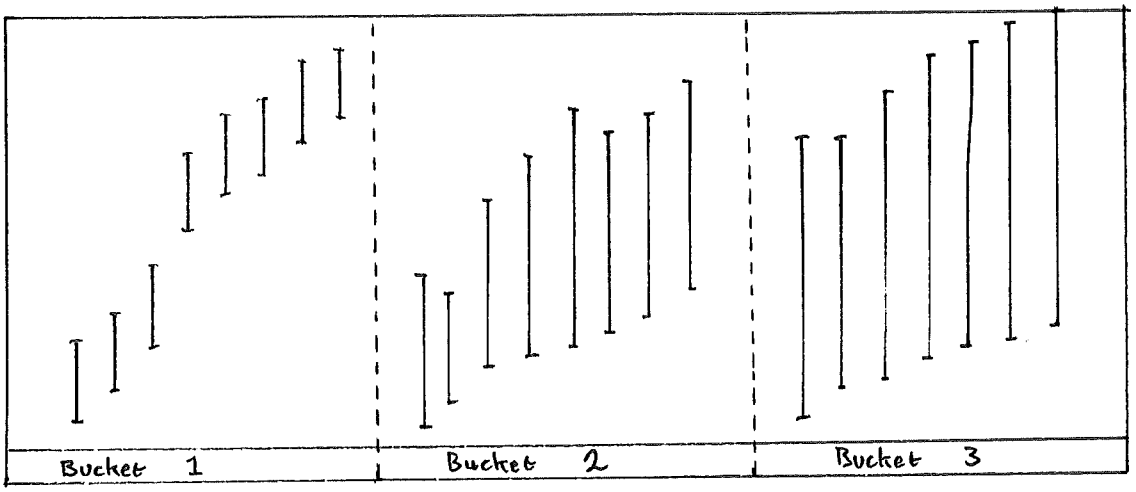
Thus these two lattice operations can be used to describe all the simple types of request that we can expect our sorting to deal with. What kind of sorting order, then, will cluster the tuples which satisfy these requests?

If we have a flat lattice, then we can sort the proper values in the normal way, and sort the null values to some fixed place, say the end. The null attribute will never satisfy a query of type (a), but will always satisfy one of type (b).

Suppose we are dealing with real intervals. If they are all of fairly similar size, then we could sort them roughly into order in order of the lower bound, say, although there is no natural ordering. This will enable us to retrieve on the two query types efficiently.

In the general case, there is no natural ordering for lattice elements. If we have, for example, real intervals which vary widely in size, or a P-domain, how do we sort them? The method used by my program is a generalisation of the two methods mentioned above: the elements are sorted into buckets by how imprecise they are, and within buckets can be roughly sorted to enable fairly efficient retrieval. Real intervals, for example, are assigned into buckets depending on the length of the interval, and within the buckets sorted on the lower bound of the interval. (See figure).

For flat lattices there are only two buckets, one of which just contains



FIGURE

the null value, and the other the precise values, which can be sorted perfectly in the normal way. In other words, this scheme reduces to the obvious method described earlier for flat lattices.

There is a description of an elegant algorithm for sorting real intervals in [WoEd], which also uses buckets. This algorithm is adaptive, in the sense that the elements are allocated to buckets depending on the existing population, rather than in a fixed way. This avoids the need to have any idea of the possible distribution in advance. It is not really suitable here, however, because

* it does not easily generalise to other lattices
* elements migrate from bucket to bucket, which would be very inconvenient here
* it is necessary to keep all the elements of the lattice in one place, to control the algorithm.

5.2 How lattices are described

The experimental program allows the user to specify the lattice domains that he requires, and store and manipulate elements of these lattices. The definition of a lattice consists of the definitions of the various objects that are required to be able to operate on elements of it. These are:

* lub and glb functions
* bottom -- the bottom element of the domain
* is-precise -- tells whether a given element is precise
* the function giving the approximation ordering of two elements
* read and print functions for the domain
* arithmetic and ordering functions as appropriate
* definitions of any special functions on this domain
* functions to implement the retrieval scheme
* mappings to corresponding imprecision domains

The last three of these require further comment: 'special functions' covers things like 'concatenate' for the domain of strings, which are accessed by their names. The standard symbols for plus, minus multiply, divide, and comparison are recognised in expressions, and if used must refer to the usual arithmetic functions, since assumptions are made about their behaviour in order to enable arithmetic expressions to be inverted when performing updates through 'computed-column' type views. Any other functions must be named, and no assumptions are made about such functions. Hence the plus-sign cannot, for example, be used to denote string concatenation. There are two functions to implement the retrieval scheme: the first gives, for an element of the domain, the bin-number to which it is assigned, and a key by which it can be sorted into that bin. (It does not matter if different elements may have the same bin and key-values). The other function takes a bin-number and a selection predicate

on elements of that domain, and returns upper and lower bounds on the key-values of elements that can satisfy the predicate within that bin. The access-method code refers to these functions. The mappings to other domains correspond to the imprecise-domain building operations in this system. There are two of these: the domain of intervals, and the P-domain. Hence for any flat lattice, there may be functions giving the corresponding elements in either of these domains; correspondingly in a domain of either of these types, there can be a function giving the set of elements of the flat lattice which a given element represents. This may not always be possible; we cannot in general give a list of all the precise real numbers represented by an element of the interval-domain. However when this is possible, it is used in forming Joins, and in Grouping, as described in the sections on those operations.

The definitions of these functions are written in BCPL for each domain, compiled, and loaded by the system as required. The definitions of many of the functions take on a standard form; for example the P-domains of all domains essentially have the same code, with references to the definition of the underlying domain inserted. It would be possible to compile just one definition for all P-domains, and supply the reference to the appropriate underlying domain at run-time. The same is true of the other domain-forming operations.

All the values in relations are stored as 32-bit words which represent lattice elements, either directly or by pointing into the value-set storage system.

The domains provided are

* variable-length character strings (flat lattice)
* integers (flat lattice)
* imprecise integers (interval-domain of above)
* truth-values (flat lattice)
* real numbers (flat lattice)
* imprecise real numbers (interval-domain of above)

The character strings and imprecise numbers are stored in the value-set system, and the representation is the pointer to this, whereas the precise numbers and truth-values are stored as themselves, with a value reserved for the null element.

In addition, a general routine is provided which allows the formation of the P-domain of any flat domain. This is implemented as follows: An element (other than bottom) of the P-domain of a lattice consists of an finite non-empty subset of that lattice. Hence it is represented by a pointer to a block giving the list of the elements in the subset. The first word of the block gives the number of elements, and the remaining words are the elements. Least upper bound in this lattice is the intersection of the subsets; if this is empty then they are not lub-compatible. A precise element is a subset with exactly one element. Ordering and arithmetic operations can be defined, rather expensively, by performing the operation required on every possible pair of elements from the arguments, in the underlying domain, and then forming the GLB of the possible results. Performing arithmetic on these domains is unlikely to be useful in practice.

5.3 Predicates

One of the main data-types used in the program is called a 'predicate'; it describes a predicate on the tuples of a relation, such as

'column one is approximated by (3.0 to 3.1)'.

It consists of a series of terms, each of which is of of the form <Col> <OP> <Value> where the OP can be 'is approximated by' or 'is compatible with'. The predicate is true of tuples that satisfy all the terms. Note that these operators have results in two-valued logic, and the predicate is evaluated in two-valued logic. These predicates look quite like select-expressions for the relational Select operation, and indeed they are used for this purpose. They are also used as 'index requests' to describe to the access method the tuples that should be retrieved, and also could be used for predicate locking in a way that will be described, although this has not been implemented.

(Predicate locking [EGLT] is a method of locking part of a database where rather than recording which records are actually locked, for example by tagging them, they are specified by a predicate. This means that any new records created while the lock is active that satisfy the predicate will also be locked, which overcomes certain problems.)

The question as to whether the predicates should be expressed in two-valued or three-valued operations is a delicate one. Since the predicates describe select-expressions, which will be expressed by the user in three-valued logic comparisons such as 'X = Y', we might expect that the operation we would use would be the three-valued equality comparison operator, described earlier. There is a slightly more suitable three-valued operator, which I named 'Meets' in [Gral]. The

problem with three-valued equality is that it can only be True when both values are precise. By contrast the value 'X meets A' gives the truth-value of the statement 'X lies in the range described by A', and so the definition is

```
X meets A :==  if   A approximates X
               then TRUE
               else (X = A)     // three-valued comparison
```

which forces X meets A to be True rather than Maybe when X lies wholly within the range of A. It can be seen that

(X meets A) is True iff X is approximated by A

(X meets A) is True or Maybe iff X is compatible with A

and hence Meets is equivalent in power to the pair of two-valued operations described earlier, whereas three-valued equality is not.

Essentially the choice is as described earlier in the section about assigning three-valued logic to the membership function of relations. If we have two-valued predicates, then they simply describe some subset of the tuples; if we have three-valued then they describe two subsets, and we must additionally specify for the operation overall whether we want the maybe-tuples to be included or not. This means that if we have three-valued predicates, we cannot select for example 'all tuples such that attribute X is **definitely** equal to 3 and attribute Y **may** be equal to 4', whereas with the two-valued functions we can mix maybe- and definite-type operations in this way. My system implements two-valued rather than three-valued predicates so as to allow these more flexible 'mixed' predicates. Thus the predicates correspond exactly to the types of retrieval request described earlier.

It should also be noted that a predicate is adequate to carry all the information in a tuple; we can characterise a given tuple by a predicate describing all the tuples above the given tuple. Hence we can regard tuples as corresponding to a subset of predicates.

5.4 Cursors

The other data-type which features commonly in the program is the 'cursor'. This points to a relation currently being processed for retrieval or update, giving a 'current position'. This notion of currency is internal to the system, and is not visible to the user in any way. When it is necessary to perform any operation on a relation (whether derived or base), a cursor must be opened on it; if it is a derived relation, this will in turn involve opening cursors on all relations involved in its definition; if it is a base relation, it will point to the disk relation that represents it. Associated with each cursor is a predicate, which defines the subset of the tuples in the relation that may be touched in this operation. Thus if a cursor is opened on a derived relation that is a Selection of tuples from a base relation, the cursor that is opened on the base relation will have a predicate that specifies that only tuples passing that selector are concerned. These predicates can be regarded as predicate locks [EGLT] which can be used, at base relation level, to determine whether two cursors represent operations which may interfere.

5.5 The relational operations available

The experimental program provides the following operations on relations: Project, Select, Join, Computed column, Group-by. These are defined on Open- and Closed-world relations as follows.

Project

The projection of a relation onto a set of columns is, as always, formed by removing the other columns and then performing any removal of redundant tuples that may be necessary. In the standard relational algebra, redundant tuples are just those that are duplicates; in this system, any tuple that is an approximation to another tuple is redundant. This is just the normal rule in this system for removing redundant tuples in open-world relations. We are not allowed to remove dominated tuples in this way in a closed-world relation, but the condition for a projection to be closed ensures that there will be no such tuples anyway.

A projection can be guaranteed to be closed-world if the base relation is closed-world, and it is not the kind of project that causes tuples to be combined, in other words none of the key columns are removed. This kind of projection is just the removal of some of the attributes, without really changing the entities described; it would be used for example to remove some information about the entities in question that was unnecessary or privileged. It can also be guaranteed to be closed (given a closed base relation) if all the columns left by the projection are constrained to be precise; in fact in this case it is 'fully-closed'.

We can widen this condition for closedness of the result if we have a list of the functional dependencies for each relation, so that we can deduce the key of the result of the projection, but as this system does not carry this information, the above condition is used.

Selection

Selection in this system consists of discarding those tuples that are not accepted by a given predicate. As we have chosen to use two-valued predicates, we do not require a maybe-select and a true-select operation, as the predicates themselves contain these options. The reasons for choosing this kind of predicate have just been discussed. The user can enter selection-expressions in the usual form, and this is translated into the lattice operations, as described in that section.

The result of selection can be guaranteed to be closed if the selection predicate refers only to columns which are not allowed to be imprecise; otherwise it may be impossible to tell whether a value satisfies the selection condition. (It would in fact be possible to note, during the computation of the projection at any one time, whether such an undecidable condition arose, and flag the result relation as being closed if it did not; this would allow the result to be closed if the selection predicate can always be decided 'in practice'; however an approach which allows a given relation to be sometimes open-world and sometimes closed seems dangerous, and has not been followed here.)

Computed column

This operation adjoins an extra column to the relation, whose value is given by an expression in terms of the other columns. This can involve arithmetic, comparison, or any of the operations defined on the relevant domains. The arithmetic and other operations are defined with due regard to the domains in question, as mentioned in section 2.5.

The result will be closed-world if and only if the base relation is closed-world.

Note that this operation allow the selection of tuples on conditions more general than those of a predicate; if we require to select all tuples where attribute A is greater than the square of attribute B, we can define a computed column with the expression

A > (B*B)

which will be in the domain of truth-values, and then use Select to select those tuples which have 'true' in this column.

Group-by

It is also possible to extend the GROUP-BY operator [Gra2] which is a combination of projection and forming aggregates. This operation forms a given aggregate for all tuples in groups defined by a particular attribute or attributes.

Aggregate operations can be defined in exactly the normal way in terms of arithmetic operations; once the arithmetic operations are defined, the definitions of the aggregate operations follow automatically. For example, SUM(A) where A is some attribute of a relation is just the sum of the values of the attribute from each tuple. Of course, if any tuple has 'unknown' for A, then the sum will be 'unknown'. This is the correct answer. It is however because of this that the implementors of System R chose to define instead that all tuples with unknown values should be excluded when computing the aggregate. This allows the aggregate to be computed even when there are unknown value present, but only at the cost of discarding the correctness of the sum, and producing the anomalies criticised by Date [Dat2]. (This is rather like interpreting the query in the 'internal' model [Lips]). With this approach, they take imprecise information and appear to produce a precise answer from it; this is a dangerous practice. We would suggest the principle that the result of any database operation should always be at least an approximation to the true result that would be obtained if all the precise values were available for the stored information. (In other words the query must be interpreted in the 'external' model unless explicitly told otherwise). If the best approximation to the true answer that can be given is 'unknown' then this is what must in all honesty be delivered. If what the user really wants is the sum of all precisely known values, he can of course ask for this.

In fact it will often be possible to do rather better than this. If we define the SUM aggregate as the arithmetic sum of the values, then if any value is completely unknown, the result will also be unknown. However very often there may be a constraint on the range of values that the unknown can take. With this constraint, the value is not entirely unknown, and so the possible range of the sum or average is restricted.

Given that we know how to form aggregates, we can define GROUP-BY. The base relation must be closed-world, as explained before, and the result can be guaranteed to be closed if all the columns upon which grouping is being done are constrained to be precise.

Consider grouping the SAMPLES relation in the introductory section.

GROUP SAMPLES by SITE forming average(age)

Site	Avg age
T44	59 to 63
T41	61 to 64.6
T54	60 to 63

The problem is how to define the groups if some of the grouping attribute values are imprecise. For example, suppose we wish to group samples by Class. We will remove sample S5 where the class is unknown, since we have no restriction on the values that this might take. If the algorithm forms a new group whenever a value is found that does not compare equal with any existing group, (which is what system R does, for example) then each tuple where the grouping attribute is not precise will occur in a group of its own.

LET SAMPLE2 = SAMPLE - (S5, T41, ?, 58 to 60)

SAMPLE2

Sample	Site	Class	Age
S1	T44	D3	59 to 63
S2	T44	D4	59 to 63
S3	T41	D4 or D6	60 to 67
S4	T41	D6	65 to 67
S6	T54	D3	60 to 63
..	...	...	...

GROUP SAMPLE2 by CLASS forming count

Class	Count
D3	2
D4	1
D6	1
D4 or D6	1

This seems quite attractive, but it does not follow the principle mentioned above, that the values in the relation should be an approximation to the 'true' values. How can the tuples in the above relation be interpreted? The meaning of the tuples should be 'the total for group X is Y'. The result relation produced is not an approximation to the truth; it contains tuples giving precise values for aggregates that may well be wrong. We should insist that the results given indicate the true range of uncertainty of the result. Also the above relation, which could well be closed-world, has imprecise values in the keys. What must be done is to form a result tuple for each group, and where an input

tuple cannot be precisely grouped, its contribution must be taken into account for every group to which it could possibly belong. Thus an imprecise value for the aggregate will be produced.

GROUP SAMPLE2 by CLASS forming count

Class	Count
D3	2
D4	1 to 2
D6	1 to 2

Thus the key columns, i.e. the set of attributes on which grouping is done, contain only precise values in the result, and the result relation can be closed-world. Those columns can be defined to be over the 'flattened' versions of the domains in the base relation, since they must be precise.

It is apparent why we had to remove sample S5. If a tuple were present with completely undefined Class, we would have to produce a result tuple for every possible Class, or report it as an error.

It will be noticed that if the totals for the groups are then added together, the result is not the precise total for the whole relation, but an approximation to it; information has been lost. However it is not a disaster that information has been lost by computing the 'grand total' in this way; there are many expressions that can be computed in the database that discard information; at least no incorrect information has been introduced. The precise grand total can always be found by asking for it directly.

Join

The discussion of this has been left till last, since there is some difficulty in deciding how to define Join in a system which caters for imprecise values. This difficulty arises because of the very syntactic way in which Join is defined. The normal Join operation is the primitive in the relational model for combining together statements represented by different relations, but it is defined in the simplest possible way; this has its virtues, but it fails to take account of the many ways in which relations can be used to express statements. A better system (such as Codd's RM/T [Codd]) would set out a description of a number of fixed ways in which relations are to be used and could then define joining operations which are appropriate to them.

Unfortunately there was not time to build into my system a full semantic data model such as RM/T, and so the problem remained of defining a joining operation to make sense in all cases. The normal extension of Join to take account of nulls is the Outer Natural Join, in which tuples that would normally take no part in the Join are mated with tuples consisting of all nulls. Thus for example if we are joining the SAMPLES relation with the SITE relation, and there is a sample tuple which would not be joined to anything, rather than lose that sample we put it into the result with nulls for country and other attributes of Site.

This is done because it is better not to lose all record of the sample; looked at in the light of the present discussion, we can say that it may enable us to produce a closed-world result, whereas if we discarded tuples we would be left with an open-world result.

Codd points out that the use of this depends on the relation semantics. He says:

> 'If an operator generates one or more nulls, these are of the value unknown type, which is consistent with the open-world view. If we were dealing with the closed-world view, the 'inapplicable' type would be more appropriate'.

In other words, if a tuple fails to be matched in a Join, we can extend with nulls if the other relation was open-world, since there could be a missing tuple that should have joined with it; however if the other relation was closed-world, there can be no missing tuple, and so there can be no valid value to extend with; the 'inapplicable' value should be used if supported, or in my system, the tuple should not be joined. However the above analysis is partly wrong; a tuple in a Join can fail to be joined for two reasons: either there is no corresponding tuple in the other relation, in which case the above comments apply, or because the tuple is imprecise in the joining column(s) and so its matching tuple(s) cannot be identified even if present. I would thus amend Codd's analysis to say that the 'inapplicable' case arises if the unmatched tuple is precise in the joining columns, and the other relation closed-world, and yet no match can be found. In this case, my system must discard the unmatched tuple, just as in the traditional Join.

With the above tinkering, we could supply the Outer Natural Join as the joining operation to be used in the system quite satisfactorily. There are one or two things that can be improved, however, by considering the way in which Join is used. Usually when using Join, we have one relation which we are 'interested in', and we are joining it to the other relation to apply some function to the tuples; for example, when joining the Sample and Country relations, we might be interested in samples, and want

to know for each sample what country it came from, or we might be interested in countries, and want to know for each country what samples came from it. We are unlikely to have both points of view at once. From here on the relation of interest will be written on the left-hand side of a Join symbol, and referred to as the left-hand relation. This will tend to result in a natural-looking form for expressions.

When we use a Join in this way, we want each tuple in the relation we are 'interested in' to participate, and so we wish to use the 'outer' operation to make sure that no tuples are lost; however we don't care if some tuples in the other relation are unused. The sensible operation to have is therefore an outer join which is 'outer' on one side only. If we accept this point of view, which is implemented in my system for experiment, there is one further improvement we can make to the Join operation, in a particular case.

The 'function' we are applying to the left-hand relation in a Join can in general be multi-valued; several tuples can match each left-hand tuple. For example, if we are interested in countries, there will be in general several samples that belong to each country. However there is the special case where the function is known to be single-valued; this is when the joining columns contain the key of the right-hand relation. In this case the cardinality of the Join will be equal to the cardinality of the left-hand relation. This case can be easily detected, and two observations apply to it: firstly, this is the one case in which we can deliver a closed-world result, if the left-hand relation is itself closed. This follows from the fact that the key of the Join is in this case the key of the left-hand relation. Secondly, if the joining column(s) in the left-hand relation are imprecise, we may be able to do slightly better than the normal outer operation of extending with null values; if the joining value is imprecise, but not completely null, it

may be possible to represent it as a choice of some finite set of precise values, find the tuples that join to those, and take the GLB of those tuples as the result. For example, if a sample is given as coming from one of two sites, the country value joined to it can state that it comes from one of the two corresponding countries; if we are so lucky that they are the same, then we can fill in the country even though the site was not precisely known. This would happen, for example, if the site information was originally derived from an input giving just the country of origin, in the first place.

SAMPLE

Sample	Site
S1	T44
S2	T41
S3	T41 or T44
S4	?
..	...

SITE

Site	Name	Country
T44	Delmar	USA
T41	Branimir	Bulgaria
T54	Elsworth	England
...	...	...

SAMPLE * SITE

Sample	Site	Name	Country
S1	T44	Delmar	USA
S2	T41	Branimir	Bulgaria
S3	T41 or T44	Delmar or Branimir	USA or Bulgaria
S4	?	?	?
..	...	...	...

To summarise, the join operation provided is not symmetrical, but takes a left-hand relation and applies the right-hand relation to it; every tuple in the left-hand relation participates, being extended with nulls if it cannot be matched (but this is not done if the left-hand value is

unmatched even though precise and the right-hand relation is closed-world); if the joining columns contain the key of the right-hand relation, then instead of extending with nulls, we may be able to use the GLB of the possible matches on the right.

5.6 How updates are propagated

In the experimental system, there are two update operations, ADD and DELETE, for inserting and deleting tuples from relations (stored or derived). In most cases it is fairly clear how these operations should be performed on a derived relation.

A request for an update operation on a relation will be taken in conjunction with the locking predicate that was placed on the relation for this operation; see the section about predicates. This means that only tuples satisfying the predicate are concerned, and so any tuple to be added or deleted must also be compatible with the predicate, and if so can be assumed to satisfy it. For example, if we attempt to insert a tuple with a null in column 3 into a relation opened with the restriction passed down from above that only tuples where column 3 = 7 are being accessed, the system will assume that col 3 = 7. This would be done if for example there were a select operation at some point higher up specifying that col 3 = 7.

There are thus three pieces of information to be considered when translating a request for an update to a derived relation: the values in the tuple, the locking predicate, and any information resulting from the definition of the derived relation itself; for example if the derived relation is a computed-column relation, then the definition itself gives an arithmetic relationship amongst the columns which may be used to fill in missing values.

Projection

A tuple in a projection corresponds exactly in meaning to a tuple in the base relation with nulls filled in for the missing columns. No extra information is provided by the definition of the operation, and the locking predicate can be translated exactly into a locking predicate on the base relation, thus avoiding any need to consider it further. Hence when we access a projection, the locking predicate is translated and passed down, and then insertions and deletions are translated into requests on the base relation by expanding the tuple with the necessary nulls.

Selection

A tuple in a selection corresponds directly to a tuple in the base relation; similarly the locking selector can be passed back directly; however there is extra information about the tuple provided by the definition of the selection; any tuple referred to as being in the selection must satisfy the selection predicate. This is dealt with by intersecting this predicate with the locking predicate to be passed back, ensuring that only tuples which pass the selection predicate will be accessed. Thus when we access a selection, the locking predicate passed back is the conjunction of the one handed down and the selection predicate; then requests for insertions and deletions are simply passed on to the base relation.

Computed column

It is very important to be able to update computed columns in relations, since otherwise we cannot use the operation to implement such useful things as change of scale or units. However there will clearly be some kinds of arithmetic and other operations that are not invertible.

When we pass the tuple back to the base relation, we remove the computed column. Hence to capture the information in that column, we would like to ensure that the other values are sufficiently precise that it can be recomputed from them. We can try to improve any imprecise values in the tuple using the precise values in it, and the definition of the computed column, and also any information in the locking predicate, which in general we will not have been able to pass back perfectly.

It can be seen that the problem of translating the locking predicate onto the base relation is a slight generalisation of that of translating a tuple from the derived relation to the base. A predicate can hold all the information in a tuple, as described in the section about predicates. When we are trying to translate the locking predicate into a predicate about the base relation, any terms that refer to columns present in the base relation can be translated directly; however for terms referring to the computed column we must try to invert its definition, just as when mapping tuples. For example, if the computed column is defined as (7.0 times column three) and the locking predicate is (computed col = 42) then we ought to translate this into a locking predicate on the base relation saying (column 3 = 6.0).

The least we can reasonably do is to make such deductions as

if (unknown)*7 = 42 then (unknown) = 6

that is, we can reasonably expect to invert arithmetic operations which have inverses when all the operands except one are precise. However we cannot really deduce much from knowing that the product of two unknowns is 42.

When we access a computed column relation the locking predicate will be translated as best we can, using the definition of the computed column. It will not matter if the translation is not perfect; this will simply mean that (a) we lock slightly too much of the database and (b) we must refer to the locking selector again when we come to translate tuples. When translating update requests, we must attempt to ensure that the computed column can be reconstructed from the other columns, using its definition and the locking predicate. This means trying to fill in any imprecise values present by inverting the expression defining the computed column.

What are the implications if we cannot map back all the information present? For insertion, it means that we are losing information, and the update will not succeed completely. However, for deletion, it is more serious; the result would be that too much information would be deleted. While in a sense this is only 'losing some information' too, we do not want to lose information that we already had stored in the database. We would regard this as an error. Consequently, the system must adopt a safe approach to deletion, and this is described below.

Group-by

There seems to be no justification for trying to update this kind of aggregate relation, and so it is treated as read-only in my system.

Join

A tuple appearing in a Join corresponds in meaning to two tuples, one in each base relation. Additional information provided by the definition of the Join is that the joining columns of the tuples are equal. The locking predicate can be represented precisely by two predicates on the base relations.

Thus when a Join relation is opened for access, the locking predicate will be split into two predicates on the base relations. Then when any tuple is passed down, it will be similarly split, and the precision improved as far as possible by using the definition of the Join and the locking selector. For insertion requests, the two halves will be inserted into the base relations. For deletion requests, there is a choice of deleting from the left, from the right, or both. Given the semantics of the Join operation as described earlier, the right-hand side represents a function which for the purpose of this operation is fixed, and the left-hand side represents the entities of interest; hence the deletion is performed on the left. It would of course be easy enough to provide a switch to allow the user to over-ride this.

General methods to improve insertion and deletion

As has been explained before, if a tuple to be inserted into a Join is imprecise in the joining column(s), this gives rise to two imprecise entries A and B which refer to the same real-world value. This information cannot be represented explicitly, because we have no way of flagging two values to say that they are the same; if one of the entries were to be precise, we could use all the information by setting the other to the same value. If both are unknown we can do nothing. Thus our ability to use the information available is limited by the imprecision in the tuple. Furthermore since the source of the above extra information A=B is from the definition of Join itself, it follows that if one value is unknown then both will be, and so we can do nothing. However it may be that the database can supply the value of one of the unknowns. If we can retrieve from the database the value of one of A and B, we can set the other to the same value, and thus improve the completeness of the update operation.

What the above describes, in a rather roundabout way, is the operation described much earlier for updating a Join, whereby we 'invert' the function represented by the Join, to work out what the missing value in the Join column is. Now that we have settled on a function-applying meaning for Join, this is even clearer. If the value in the Join column is imprecise, we attempt to discover its value by retrieval from the right-hand relation, in order to use this in the left-hand relation.

Going back to general terminology, the strategy is as follows: if in an insertion operation on a derived relation (of any kind) there is information available which cannot be used because some value has been passed down as missing, attempt to retrieve from an underlying relation the value of that attribute.

This is implemented as follows. When a tuple is inserted into any relation, a result tuple is passed back, whose interpretation is

'this stronger statement must in fact be made'.

Formally speaking the definition of this is that the tuple passed back is the GLB of the tuples above the one passed down that can actually be stored. The restriction on what can actually be stored may be any kind of constraint, such as the key of the relation, or the fact that it is defined in a particular way. Hence, for example, if we pass a tuple down with attribute A undefined, if the underlying relation into which we are inserting is the result of a selection such that A=7, the tuple fed back will have A set to 7; similarly if a tuple already exists with the same key, and A=7 in that tuple, then A=7 in the tuple fed back.

These fed-back tuples can be used to improve update operations as described above. The routine to insert into a Join, for example, inserts first into the right-hand relation, and then uses the fed-back tuple to make possible improvements before inserting into the left-hand relation. If the fed-back tuple from the left were also to yield extra information, then the above could be repeated, although this is unlikely.

Similarly for the computed-column relation, if the operation cannot be inverted because some values are imprecise, the tuple is inserted, and the fed-back tuple examined to see if the values have been made available. If so, the improved tuple can be inserted.

Turning to deletion, again it may be impossible to use all the information present, but in this case, as explained earlier, the result would be the deletion of too many tuples.

For example, if we were given a tuple to delete from a Join, with the Join column unknown, we could not simply split this tuple and pass it on to be deleted from the left-hand relation, because this would mean the deletion of tuples with **any** value in that column, not just those that participate in the Join. Similarly we could not describe the set of tuples to be deleted below a computed-column operation, as it might be all those tuples satisfying some complicated arithmetic condition. Thus the problem again arises from our inability to translate the predicate through the relational operation.

In this case the solution adopted is to scan the derived relation for the tuples to be deleted, and then delete them one by one. Once a tuple to be deleted has been found, the values in it are known exactly and so we can request its deletion. Thus the general strategy is as follows: if, in a request for deletion from a derived relation, there is information about the tuples to be deleted which cannot be passed down, then rather than pass down one request for the appropriate deletion, we must instead scan the derived relation, indexing as efficiently as possible to find tuples satisfying the deletion condition, and request their deletion one by one.

It can be seen from the above discussion that the crux of the matter is understanding the relational operations as transformers of predicates. The possible tuples correspond to a subset of the predicates. The choice of the set of predicates to be used will determine the efficiency and power of the system. Allowing completely general logic formulas as predicates might produce a very powerful system, perhaps complete in ways which this one is not, but would be more expensive and rather less like a traditional database system.

5.7 Constraint enforcement

The system provides for specifying constraints on the database by attaching a predicate to any relation, specifying that all the tuples of that relation must satisfy the predicate. This is fairly general, since the relation can be a derived relation, and thus involve joining of several relations, and computation of arithmetic expressions by computed-column. No more generality would be achieved by allowing several constraints on a relation, since in their place we can attach the single predicate which is the their conjunction.

This mechanism does not cover every kind of constraint we might want; it is necessary to make at least some attempt to model functional dependencies, and in this system that is done by noting which columns of each relation constitute the key. We might well also want some implementation of referential constraints [Dat1].

The system deals with the predicate constraints attached to relations in the way described in section 4.3 on the meaning of constraints. We wish to regard the constraints as being evaluated in three-valued logic. If a constraint is found to be true, no action is required; if false, the database state is invalid; if 'maybe', the database state needs to be refined so as to make the constraint true. The predicates which are acting as constraints are expressed in two-valued logic, as described earlier, so this procedure is equivalent to: if a constraint is true, no action is required; if false, then the database should be altered to the least state above its current state that makes the constraint true, or if there is no such state, then the database is invalid. (The word 'above' refers to the approximation ordering in the lattice.)

At the end of each transaction, the constraints are checked. For each relation, the attached constraint is evaluated on each tuple. If a tuple is found which does not satisfy the predicate, then it is replaced by the least tuple above it which does. If no such tuple exists, the transaction is in error and is rolled back. When a tuple is replaced, this is done by inserting the improved version into the relation, which alters the database state to a state above the one it was in. Since the relation may have been a computed relation, this can affect many of the base relations, as the change is propagated back to them. Thus this mechanism can produce effects on the database as a whole, not just on the new information introduced by the transaction. Because the change only takes the database state to a more precise state, however, the change cannot affect the satisfaction of any previously checked constraint; there is no danger that the enforcement of a later constraint will cause the failure of one checked before it. An example of how this mechanism works is given in the chapter on the example schema.

The actual job of checking every tuple of every relation on which a constraint is defined is done, in my system, the hard way; that is by computing the tuples of every such relation and checking them. This is a very expensive way of carrying out the procedure, but fortunately it is clear that there are ways of implementing it that will be more efficient. There has not been time to program these, and so my system produces the same effect in a more expensive way. The approach to this problem which seems most likely to help is that of differential maintenance of views. Koenig and Paige [KoPa] point out that this can be used for efficient maintenance of view relations, and checking of constraints on them. After making any change to a base relation, the effect on any views defined on it can be computed, and a list of alterations to the view relation produced. This can be used in two ways. Firstly, if it is desired to keep a stored copy of the derived relation then this can be

kept in step, avoiding the need to re-evaluate it every time reference is made to it. In a situation where the derived relation is referred to more often than it is changed, and it is not very cheap to compute, this represents a big saving. Secondly, even if a stored copy of the derived relation is not kept, the changes to be made to it can be inspected, and any insertions checked to see that the new tuples satisfy the constraint on the relation. Any realistic system to support views would have to include at least some facilities for maintaining derived relations differentially.

Another approach to short-cutting the checking of constraints is that of Hammer and Sarin [HaSa], which involves looking at the definition of each update transaction, and observing which constraints can be affected by it, and conditions under which it is safe. By this means, a system will probably be able to avoid checking most of the constraints for any given update transaction. Thus there are some grounds for hope that the enforcement of constraints on a database is not a hopelessly impractical scheme.

CHAPTER 6

THE EXAMPLE SCHEMA

This section covers in detail the design of the example schema used earlier. This should demonstrate the use of some of the facilities discussed in the dissertation.

The entities of interest are samples, sites, experts on those sites, and items of equipment. Each instance of these has a unique key to identify it, which has been assigned to it in some way. All the relations are closed-world except EQUIPMENT.

The samples are described by the SAMPLE relation:

Sample	Site	Class	Age	Partof
S1	T44	D3	59 to 63	S2
S2	T44	D4	59 to 63	?
S3	T41	D4 or D6	60 to 67	?
S4	T41	D6	65 to 67	?
S5	T41	?	58 to 60	?
S6	T54	D3	60 to 63	?
S7	T54	D9	?	?
S8	T54	D3 or D6	59 to 60	?
..	...	...	...	...

In this relation, Sample is the key; the Site column gives the site at which the sample was found; the Class column gives a classification of the sample, which may be imprecise; the Age column gives the age as a real interval (in millions of years); and the Part-of column gives the sample number of another sample of which this was a part, if any. If we had an 'inapplicable' null in this system, it would be appropriate to use it here for samples that are not part of any other sample, but since this system does not provide 'inapplicable' I have used 'unknown'.

We can define selections from this relation:

SEL: Select SAMPLE (Age in [60.0, 70.0])

Sample	Site	Class	Age	Partof
S3	T41	D4 or D6	60 to 67	?
S4	T41	D6	65 to 67	?
S6	T54	D3	60 to 63	?
..	...	...	...	...

or on a 'maybe' condition:

SEL2: Select SAMPLE Maybe (Age = 66.0)

Sample	Site	Class	Age	Partof
S3	T41	D4 or D6	60 to 67	?
S4	T41	D6	65 to 67	?
S7	T54	D9	?	?
..	...	...	...	...

If we add a sample to the former selection with a null entry for the age, it will automatically be filled in as [60 to 70] but no similar deduction can be made for the second selection, since we cannot limit the possible range of the age merely by knowing that it could be equal to 66.
The sites are described by the SITE relation:

Site	Name	Country	Latitude	Longitude
T44	Delmar	USA	34 to 35	268 to 269
T54	Elsworth	England	52.3	0
T41	Branimir	Bulgaria	42.5	25.1
..	...	...	...	...

Here the Site column is the key, and the other information is fairly self-explanatory. The latitude and longitudes are allowed to be imprecise. We can of course Join this relation onto the SAMPLE relation to give all the above information for each sample.

The Class relation gives further information about each sample class:

Class	Group	Description
D3	A	Conglomerate
D4	B	Nodule
D6	B	Eclogite
D9	C	Shale
..	...	...

and we can again Join this onto the SAMPLE relation; this time, however, entries for samples which are imprecisely classified will cause the g.l.b of the relevant information to be formed.

Join SAMPLE CLASS

Sample	Site	Class	Age	partof	Group	Descr
S1	T44	D3	59 to 63	S2	A	Conglomerate
S2	T44	D4	59 to 63	?	B	Nodule
S3	T41	D4 or D6	60 to 67	?	B	?
S4	T41	D6	65 to 67	?	B	Eclogite
S5	T41	?	58 to 60	?	?	?
S6	T54	D3	60 to 63	?	A	Conglomerate
S7	T54	D9	?	?	C	Shale
S8	T54	D3 or D6	59 to 60	?	A or B	?
..	...	...	...	...	...	...

Since the description is in the domain of strings, if there are two different descriptions the result will simply be "?"; however the Group attribute is allowed to be imprecise, and so a list of possible groups will be formed, as for sample S8.

We can form a condensed version of the above Join by discarding some columns that are not of interest:

Sample	Site	Age	Group
S1	T44	59 to 63	A
S2	T44	59 to 63	B
S3	T41	60 to 67	B
S4	T41	65 to 67	B
S5	T41	58 to 60	?
S6	T54	60 to 63	A
S7	T54	?	C
S8	T54	59 to 60	A or B
..	...	...	...

If we were to insert a new tuple into this:

S9	T44	63 to 65	B

the result would be to insert a tuple filled out with nulls into the Join:

Sample	Site	Class	Age	partof	Group	Descr
S9	T44	?	63 to 65	?	B	?

and this would cause the set of possible Classes to be inferred from the Class relation; thus the tuple added to the SAMPLE relation would be:

Sample	Site	Class	Age	partof
S9	T44	D4 or D6	63 to 65	?

Enforcement of a constraint

Suppose we wish to place a constraint on the SAMPLE relation, to the effect that where a sample is part of another sample, it must have the same age and site. This can be specified as a constraint on the tuples of the Join of the SAMPLE relation with itself:

Sample	Site	Class	Age	P-of	Site	Class	Age	P-of
S1	T44	D3	59 to 63	S2	T44	D4	59 to 63	?
S2	T44	D4	59 to 63	?	?	?	?	?
S3	T41	D4 or D6	60 to 67	?	?	?	?	?
S4	?	D6	?	?	?	?	?	?
..	...	...	...	...	...	...	...	...

The constraint is that col 6 = col 2 and col 8 = col 4. The basic predicates in this system do not give operations comparing columns, so this is expressed by producing a computed column whose value is this condition, and constraining it to be TRUE. We produce

Arith: (#6 = #2) & (#8 = #4)

Sample	Site	Class	Age	P-of	Site	Class	Age	P-of	Constr
S1	T44	D3	59 to 63	S2	T44	D4	59 to 63	?	?
S2	T44	D4	59 to 63	?	?	?	?	?	?
S3	T41	D4 or D6	60 to 67	?	?	?	?	?	?
S4	?	D6	?	?	?	?	?	?	?
..	...	...	...	...	...	...	...	...	...

and then constrain the above relation ARITH with the predicate (#10 = T). If a tuple is inserted into SAMPLE marked as being part of S4, then the corresponding tuple in the above computed relation will have the constraint column 10 equal to 'maybe', because sample S4 has unknown site and age.

S11	T41	?	65 to 67	S4	?	D6	?	?	maybe

The action of the constraint enforcer upon detecting this will be to insert a new tuple into the computed relation, obtained from the above by strengthening it with the constraint (#10 = T). This is just the same tuple with column 10 set to True. Translating this insertion through the computed-column operation will cause the following tuple to be inserted into the Join:

S11	T41	?	65 to 67	S4	T41	D6	65 to 67	?

This was obtained by inverting the computed-column expression and using the existing values in the tuple. Finally the translation of the above by the Join operation will cause the following tuples to be inserted into the SAMPLE relation:

Sample	Site	Class	Age	Partof
S4	T41	D6	65 to 67	?
S11	T41	?	65 to 67	S4

so that the Site and Age of sample S4 are now available; they have been deduced from the constraint and the new data. Note that the tuple affected was not the new input, but an existing tuple in the database.

An open-world relation

The EQUIPMENT relation, as described in the section about open-world relations, is as follows:

Inv-no	Site	Descr	Value
I137	T41	Auger	300
I139	T41	Auger	320
I4290	T41	Concentrator	50
I3054	T44	Typewriter	50
..	...	...	...

Somebody concerned only with which facilities are available at each site might use the projection:

Site	Descr
T41	Auger
T41	Concentrator
T44	Typewriter
..	...

Inserting a new tuple into this:

T56	Landrover

would cause the base relation to be modified thus:

Inv-no	Site	Descr	Value
I137	T41	Auger	300
I139	T41	Auger	320
I4290	T41	Concentrator	50
I3054	T44	Typewriter	50
?	T56	Landrover	?
..	...	...	...

That is, a new tuple with a null key appears. This has the meaning 'there are one or more landrovers at T56'. The same effect would occur if we inserted 'landrover' into a view formed by selecting the items at T56 and projecting to get only the description.

A many-many relation

The EXPERT relation describes a many-many association between experts and sites. It is as follows:

Expert	Site
Mike	T44
Mike	T56
Ken	T56
John	T41
Hiyan	T44
..	...

If we Join SAMPLE to this, and project onto Sample and Expert, we get the following

Sample	Expert
S1	Mike
S1	Hiyan
S2	Mike
S2	Hiyan
S3	John
..	...

If we add a new tuple to this:

S9	Mike

then the most the system can do is deduce that the site is T44 or T56, and so the new sample S9 is added to SAMPLE with 'T44 or T56' for the site. The effect on the above view is:

Sample	Expert
S1	Mike
S1	Hiyan
S2	Mike
S2	Hiyan
S3	John
S9	?
..	...

In other words, the desired tuple does not appear, because the information cannot be stored in the given base relations.

CHAPTER 7

CONCLUSIONS

Early applications of databases often tended to be little more than flat files. The commonest examples used for database texts were employee record files and 'supplier-part-quantity' files. The main demands in cases like these were for systems that could access the data very rapidly, and maintain integrity of files across system crashes, even with concurrent use.

The requirement for application programs to remain usable after changes to the layout of the database led to the development of multi-layer models for databases, such as in the ANSI/SPARC report. This also provided the possibility for different application programs to see the database in different ways. At first, this 'subschema' type of facility consisted only of restricting each application program to the subset of

the database that it required. However there have been proposals to allow subschemas, or 'external schemas', that are structured very differently from one another; there have also been some systems, (not usually commercial systems) that implement this.

Applications are now envisaged for databases in which this kind of multi-level facility will be very important. To take one example, consider a database used in designing a VLSI chip. At the bottom level, the chip might be described in terms of rectangles of silicon in layers, giving real-number coordinates of the corners of rectangles. This information will be necessary to many programs, such as one that drives a machine to make the chip, or one that checks for interference between blocks. At the next level up, the description will be in terms of transistors, perhaps, and then gates. Above that, entities like 'register' or 'adder' might appear, and then even 'processor'. Different designers will want to have access to these different levels.

In the past it has been necessary to use different systems for storing, retrieving and manipulating entities on these different levels. It is far better, however, to have them integrated in one system, which can perform the mapping of information from one level to another when necessary. The different users can see different views of the same data, structured and described quite differently. The system would ideally allow users to perform any operation on their view of the database.

In order to allow this it is necessary for operations on views to be translated into operations on the conceptual schema level. We have pointed out three ways in which it may be impossible for an operation on a view to be translated. These are:

(a) The data model in use may not allow the required information to be represented.

(b) The base schema in use may not be able to represent the information.

(c) The translation algorithm may not be able to find the necessary operation on the base schema.

None of these problems can be easily solved. The second lies in the hands of the schema designer, and there has been much research into how to design schemas so that all the necessary facts can be represented. However it is not always possible to anticipate from the beginning how a database will be used. To deal with the third problem mentioned above, some methods are discussed in this dissertation to enable the translation algorithm to find appropriate actions whenever possible.

The first problem is the main one addressed in this dissertation. The only complete solution would be a system that could store 'any statement', which would at least mean any statement of first-order predicate calculus. This kind of system can be useful, but for many applications is not practical because of the power required to manipulate such a database. By considering operations on views defined by relational algebra expressions, it was shown that many of them naturally give rise to certain kinds of imprecise information. Hence a data model that catered for imprecise information would go some way towards solving problem (a). Such a system, having a level of power intermediate between the full predicate calculus and the flat relational model, would enable the translation of operations on many views.

It has been shown how the terminology of the theory of lattices can usefully be applied to describing incomplete information. Using this, we have proposed a system for handling incomplete information which will extend the relational model so that operations on derived relations are easier to translate. This system works by permitting the user to employ an assortment of 'blurring' operators which give, for any domain of 'precise' values, a domain of imprecise descriptions of these values which can be stored. These blurring operations effectively give predicates on the value of an imprecisely known attribute, and so the classes of predicates that can be stored depends on the choice of operations; these are chosen to correspond to those that arise naturally for common types of view definition in relational algebra.

Various other proposals for dealing with imprecise or missing information were also reviewed, most of which can also be described in terms of the theory of lattices.

The implications of modelling partial information have been discussed at some length; we have show how the relational operations can be extended to deal with partial information, and how the semantics of constraints and of relations themselves must be carefully considered with respect to partial information. It should be clear from the examples given, and from other papers on the subject, that the modelling of partial information in databases is very useful in itself.

Two particular kinds of blurring operation were described, though these are by no means the only ones that can be handled by the techniques discussed above. These are

* forming intervals over a base domain with an inherent sort-order, e.g. allowing real numbers with tolerances
* forming the P-domain of a base domain, e.g. the set of possible classifications of an object.

Finally, the implementation of an experimental system was described. This system was used largely to explore the consequences of various possible ways of handling views and partial information. It has not been possible to test it on any large collection of data, because of the limitations of the storage system it uses. Nonetheless, some of the algorithms used give hope that the theoretical proposals of the rest of the dissertation can be implemented at not too great cost.

REFERENCES

[Banc] Supporting view updates in relational databases
Bancilhon
Proc. IFIP Database Architecture Conference 1979 p. 213

[Bisk] A formal approach to null values in database relations
Biskup J.
Advances in Data Base Theory. Plenum Press 1981 (New York)

[Codd] Extending the relational model to capture more meaning
Codd E.F.
IBM Research Report RJ2599 (1979)
also ACM Trans. Database Systems 4 (1979) p. 397

[ChGT] Views, authorisation and locking in a relational database system
Chamberlin D., Gray J. and Traiger I.
Proc. of ACM National Conference 1975 p. 425

[DaBe] On the updatability of relational views
Dayal U., Bernstein P.A.
Proc. VLDB Conf. 1978

[Dat1] Referential integrity
Date C.J.
Proc. VLDB conference 1981 p. 2

[Dat2] Null values in database management
Date C.J.
Proc. second British National Conference on Databases 1982

[EsCh] Functional specifications of a system for database semantic integrity
Eswaran K. and Chamberlin D.
Proc. VLDB conference 1975 p. 48

[EGLT] The notions of consistency and predicate locks
Eswaran K., Gray J., Lorie R. and Traiger I.
Comm. ACM November 1976

[FuSS] Permitting updates through views of data bases
Furtado, Sevcik and dos Santos
Information Systems, Vol. 4 p. 269

[ImLi] On representing incomplete information in a relational database
Imielinksi T., Lipski W.
Proc. 7th VLDB Conf. (1981) p. 388

[Gra1] Implementing unknown and imprecise values in databases
Gray M.
Proc. First British National Conf. on Databases

[Gra2] The group-by operation
Gray P. M. D.
Proc. First British National Conf. on Databases

[HaSa] Efficient monitoring of database assertions
Hammer M., Sarin
Paper given at ACM SIGMOD conf. 1978

[Klug] Theory of database mappings
Klug A.
University of Toronto Computer Systems Research Group 98

[KoPa] A transformational framework for the control of derived data
Koenig, Paige
Proc. VLDB Conference 1981

[Lips] Semantic issues connected with incomplete information databases
Lipski W.
ACM Trans. on database systems 4,3 (Sept 79) p. 262

[Mcle] High-level domain definition in a relational database system
McLeod D.
ACM-SIGPLAN Notices 8,2 (1976) p.47
Proc. of conference on Data: abstraction, definition and structure

[Nijs] A gross architecture for the next generation DBMSs
Nijssen G. M.
Modelling in DBMSs. North-Holland 1976
(Proceedings of IFIP Conf. on Modelling in DBMSs).

[PePB] Mapping external views to a common data model
Pelagatti, Paolini and Bracchi
Information systems 3,2 (1978) p. 141

[Robs] The implementation of domains and referential constraints in CODD
Robson M.
Proceedings of second British National Conference on Databases

[Scio]The universal instance and database design
Sciore E.
Princeton University 1980

[SmSm] Database abstractions: aggregation and generalisation
Smith D. and Smith J.
ACM Trans. on database systems 2,2 (June 77) p. 105

[Spyr] Translation structures of relational views
Spyratos
Proc. VLDB conference 1980 p. 411

[Ston] Implementation of integrity constraints and views by query modification
Stonebraker M.
Proc. of ACM SIGMOD conference on management of data 1975

[Stoy] Denotational semantics -- the Scott-Strachey approach
Stoy J. E.
MIT Press 1977

[SysR] System R: A relational approach to database management
Astrahan et al.
ACM Trans. on Database Systems June 1976 p. 97

[Vass] A formal treatment of imperfect information in database management
Vassiliou Y.
University of Toronto Computer Systems Research Group report CSRG 123

[WoEd] Interval hierarchies and their application to predicate files
Wong and Edelberg
ACM Trans. on database systems 2,3 (Sept 77) p. 223