Tuple-Marks

NULLs everyone can live with

v0.1 [mlf-970602]

Mark L. Fussell

1220 N. Fair Oaks Ave, #1314
Sunnyvale, CA 94089
408.734-9068

Mark.Fussell@ChiMu.com
www.chimu.com

Abstract

There has been a great deal of discussion about NULL values in relational algebra. There is the argument that NULLs help us with real database problems of handling lack of information. There is a counter-argument that NULLs and three-valued logic (3VL) cause major problems with relational algebra and are unnecessary. These arguments have been very informative and there is no reason to repeat them. Instead I will change the playing field.

I propose a simple but unusual meaning for NULLs. A NULL describes a tuple, not an attribute value: A tuple with a NULL attribute belongs to a different relation than a tuple without a NULL attribute. A NULL becomes a Relation Distinguishing Tuple-Mark. This meaning provides the standard benefits of NULLs: It helps us model missing information with a small number of relation variables (i.e. tables) and simplifies understanding and interacting with a database. On the other hand, because NULLs are never attribute values (or value "marks") they have no impact on domain operations or the 2VL of basic predicate logic. These are NULLs I believe everyone can live with.

Background Requirements

This paper rests heavily on the work of E. F. Codd, C.J. Date, and all the rest of the people who helped grow relational theory. There are many references to their writings within the text and it will be hard to follow this document without a good understanding of the relational model. For example, you will need to be familiar with the precise meanings of relation, relation variable, relation value, tuple, domain, attribute, and value. I will be using these terms because they are more precise and correct for the relational model than table, column, and row.

It would also help to have read some of the previous debates on NULLs and missing information in relational modeling although this paper's approach does not directly participate in those debates.

Overview

This paper progresses as follows. First we briefly discuss how the relational model represents knowledge. We then turn to how it can represent lack of knowledge without any additions to the relational model (e.g. no NULLs or special values). This approach for representing lack of knowledge works well but has the problem that the database model can become very complex. The next part of the paper introduces simple additions to the relational model, which will allow us to simplify the database with as little impact on relational algebra as possible.

First we introduce the concept of a Multi-Relation Variable that can simplify a complex scheme while still being as expressive and correct as the original scheme. Next we make Multi-Relation Variables easy to use through the Relation-Distinguishing Tuple-Mark. The tuple-mark is the main topic for the rest of the paper: We cover tuple-marks in the NULL debate, comparisons of tuple-marks to other approaches, and how to implement tuple-marks with SQL. Finally we close with some quotes from related work and a summary.

Table of Contents

Knowledge and Meaning

To discuss how a database can handle lack of information, we must first define how we put knowledge into a relational database[1]. To start, I will bring up the familiar Suppliers relation variable (S).

S

S#	SName	City	…

The first thing me must do is to define what putting a tuple (i.e. row) into S means: We must give S a predicate. Without stating what S means we can not translate between human knowledge and the database model. Everything we tell the database in the database's language will be "true" (i.e. provable) to the database, and the database will even be able to prove other "new" truths from its model and our axioms. All of these proofs are meaningless unless a person can interpret them in human terms, which must be the identical interpretation as the person who designed the database, and the person who entered the data, and so on. The only way to be sure all these interpretations are identical is to document and disseminate the meaning of each relation (and attribute, domain, etc.) in the database.

What does S mean? Well it could mean, if there exists a tuple in S then:

S.0: *{S#, SName, City, …} – There is a supplier with identity S# who hates the name SName but likes the city City, and …*

But that is unlikely. This does show the importance of precise predicates and human interpretation. More likely the predicate[2] for S is:

S.1: *There exists a supplier with identifier S#, who has the name SName, who is located in city City, and …*

So adding the tuple

S#	SName	City	…
S1	Jones	London	…

States that there exists a supplier with identifier S1, who has the name Jones, and is located in London.

Adding a couple more tuples gives us:

S : *S.1*

S#	SName	City	…
S1	Jones	London	…
S2	Smith	Bristol	…
S4	Eiffel	Paris	…

We can now ask the database:

Q1: What are the names of suppliers
Q2: What suppliers are in London?
Q3: What suppliers are not in London?

Actually, we can't. Although the above questions use natural wording, the database can not possibly answer the questions as posed. They are in terms of our world. We can only ask the database about what it can prove about its "world" not what is true in our world. We need to do the translation to and from the database so our questions should reflect that. The corrected questions are:

Q1.2: What names can be proven to be the name of a supplier?
Q2.2: What suppliers can be proven to be in London?

[1] See [Date+M 94] for a fuller discussion of this topic.
[2] Usually I will not include the relation's attributes in the predicate specification for space purposes.

Q3.2: What suppliers can be proven not to be located in London?

These questions the database can answer. For the first question the database returns:

SName
Jones
Smith
Eiffel

And for the second question it returns:

S#
S1

And for the third question it…can't answer the question. We have no predicate that states:

There exists a supplier with identifier S#, who is not located in city City

Given how the query (Q3.2) is phrased we would need such a relation (in which we might put S2 and S4 if they don't have a location in London). But with what we have at the moment we can only answer the question:

Q3.3: What suppliers can be proven not to be provably located in London?

Which would then return

S#
S2
S4

But say we really wanted to be able to answer questions like Q3.2, how can we? We could modify the S predicate:

S.2: *There exists a supplier with identifier S#, who has the name SName, who is located only in city City, and …*

We can now answer Q3.2 because we can produce a derived relation value[3] of:

NotIn = f(S) : *NotIn.1*

S#	City
S1	Bristol
S1	Paris
S2	London
S2	Paris
S4	London
S4	Bristol
…	…

Which has the predicate we wanted:

NotIn.1: *There exists a supplier with identifier S#, who is not located in city City*

It is impossible to produce the NotIn.1 relation value without modifying the predicate for S or adding a new base relation variable to record new information. The information simply was not recorded in the database.

Summary

First, for a database to have any meaning we must be able to uniquely translate between human terms and database statements. For a relational database this requires precisely specifying what each relation, domain, attribute, and tuple means. It is especially important to remember specifying what a relation

[3] This derived value is unrealistic because it would have to include the cartesian product of the extent of S# and the possible values (minus one) of the Domain City.

means because without a formal specification a user will assume its meaning and misinterpret the answers to questions.

Second, a database can only answer questions about its world. It is up to the user to make the database's world contain the model and information necessary to provide answers that are useful for understanding the "real world".

Missing Information

Now we can deal with representing the lack of knowledge (or missing information) in a relational database. We have already seen what a database does not know: everything it has not been told it *does* know. So we don't have a problem of telling the database: "You don't know this", we have a problem of telling a database "You only know this".

Fully Determined Relations

This is the technique of using only fully determined relations[4]: We can add any relations needed to express what we know, but each should only include exactly what we know about those tuples. If this approach can suitably handle all our needs for representing missing information then we should not add another mechanism to the relational model. That would be adding complexity without any new expressiveness.

To try out this technique, what if we don't know where a particular supplier is located? So far our only base relation is S which has a predicate that states

S.2: *There exists a supplier with identifier S#, who has the name SName, who is located only in city City, and ...*

Since we don't know where the supplier is located we can't use this relation. No problem, we can add a new relation to our database.

S_NoCity : *S_NoCity.1*

S#	SName	…

S_NoCity.1: *There exists a supplier with identifier S#, who has the name SName, and ...*

And if we add two entries to S_NoCity we get:

S_NoCity : *S_NoCity.1*

S#	SName	…
S3	DuPont	…
S5	Grid	…

We now have two relation variables that each express exactly what we know about their tuples. Everything in fully determined. One of the relations has more information than the other and that is the only sense in which we are "missing" information. The S_NoCity relation is "half-full" so it could also be considered "half-empty".

So how does this new relation and the new tuples affect our last two "tough" questions?

Q2.2: What suppliers can be proven to be in London?

Q3.2: What suppliers can be proven not to be located in London?

The new tuples don't affect the queries at all. Since the S_NoCity predicate doesn't mention cities in any way, it can not affect the outcome of any query that mentions a city. It has nothing to do with it.

On the other hand, the easy question:

Q1.2: What names can be proven to be the name of a supplier?

[4] See [McGoveran 94c] and [Date 94b].

Should now return:

SName
Jones
Smith
DuPont
Eiffel
Grid

The problem is that asking this question of the database is now a bit cumbersome for the user. We need to take the two base relations, project SName, and union the results together. Not terribly difficult, but this requires a user to always remember the two tables when querying. We could also define the following derived relation value (i.e. View) to make this particular query easier for the user.

S_All = f(S,S_NoCity)

S#	SName	…
S1	Jones	…
S2	Smith	…
S3	DuPont	
S4	Eiffel	
S5	Grid	

S_All.1: *There exists a supplier with identifier S#, who has the name SName, and …*

The user can now query from S_All when asking about names, use S when asking about cities, and use S_NoCity when asking about … hmm … that isn't completely clear from our predicates. Why would we use S_NoCity instead of S_AllNames? What distinguishing trait places a tuple in S_NoCity?

Predicates that specify known "unknowns"

So far we have three predicates (renaming them slightly):

S_City.2: *{S#, SName, City, …} – There exists a supplier with identifier S#, who has the name SName, who is located only in city City, and …*

S_NoCity.1: *{S#, SName, …} – There exists a supplier with identifier S#, who has the name SName, and …*

S_AllNames.1: *{S#, SName, …} – There exists a supplier with identifier S#, who has the name SName*

We appear to have been a bit vague about the meaning of putting a tuple into S_NoCity: why is the set of cities in S_NoCity different from the set in S_AllNames? We can now decide what "unknown" information we want to represent in our database by specifying what the known information means. We could decide that putting a city into S_NoCity means that we don't know the city for that supplier, or we could mean that the supplier is a multi-national corporation that is located in multiple cities, or any number of other meanings. For a fuller discussion of the different possible meanings of missing information see [McGoveran 94b]. If we needed more than one meaning we would create multiple relations and variables. For example we could have:

S_UnknownCity

S#	SName	…
S3	DuPont	…

S_MultiNational		
S#	**SName**	**…**
S5	Grid	…

We would then need to have S_AllNames be a function of all three relations to union them together.

At the moment we choose to simply correct the predicate of S_NoCity:

S_NoCity.2: *There exists a supplier with identifier S#, who has the name SName, and … and for which we do not know the city it is located in.*

This new version of the predicate allows us to ask the question:

Q4.1: What suppliers can be proven to be among those that we don't know their location?

It has no impact on the results of our previous questions:

Q1.2: What names can be proven to be the name of a supplier?
Q2.2: What suppliers can be proven to be in London?
Q3.2: What suppliers can be proven not to be located in London?

Complexity

All these tables and views add complexity to the database and make life more difficult for the user and also for the manager of these base and derived relations. This is with just one "optional" attribute. We will get combinatorial explosions of relations if for some suppliers we knew the City but not the Name, and some we knew the previous months purchase-quota, and so on. We get an even greater explosion if we need to have multiple meanings for the missing information.

Summary

The approach in this section correctly handled lack of information by adding new predicates that contain only the information we know. We didn't need to add any new mechanisms to relational algebra. We did need to specify our predicates precisely enough to identify why tuples are placed into one predicate over another. This precision is also what enables a user to know what the results of a query mean.

Unfortunately, managing all the new predicates proved more difficult than we might desire and the resulting scheme may be difficult for a user to understand. It would be good if there was a mechanism that can ease this management and simplify the scheme, but it should have as little impact as possible on relational algebra.

Multiple Relations in a Variable

One approach would be to try to reduce the number of relation variables (tables) by allowing one variable to hold multiple types of relations. In our example we have two base relation variables:

S_City : *S_City.2*

S#	SName	City	…

S_NoCity : *S_NoCity.2*

S#	SName	…

And we have a useful derived relation value of

S_All = f(S_City, S_NoCity)

S#	SName	…

Which would be calculated as the union of S_City [-City] (projecting away City[5]) and S_NoCity.

The interdependence among these three relations is very clear, so it would be nice to be able to express and manage them all together. This will alleviate some of the complexity in using the basic normalized approach.

Multi-Relation Variables

What if we allowed S_All to have multiple relations within it, where each tuple knew what relation it belonged to? Something like this:

S_All : (*S_City.2, S_NoCity.2, f(…)*)

Relation	S#	SName	City	…
S_City.2	S1	Jones	London	…
S_City.2	S2	Smith	Bristol	…
S_City.2	S4	Eiffel	Paris	…
Relation	**S#**	**SName**		**…**
S_NoCity.2	S3	DuPont		…
S_NoCity.2	S5	Grid		…
Relation	**S#**	**SName**		**…**
S_All.1	S1	Jones		…
S_All.1	S2	Smith		…
S_All.1	S3	DuPont		…
S_All.1	S4	Eiffel		…
S_All.1	S5	Grid		…

We seem to have simplified the number of variables and derived values significantly, from three to one. Notice that we have the same number of relations:

S_City.2*: There exists a supplier with identifier S#, who has the name SName, who is located only in city City, and …*

S_NoCity.2: *There exists a supplier with identifier S#, who has the name SName, and … and for which we do not know the city it is located in.*

S_All.1: *There exists a supplier with identifier S#, who has the name SName, and …*

[5] See later the next chapter for a description of this projection notation.

This is good. We don't want to reduce the useful meanings in our database; we just want to reduce how many entities we have to manage.

He examples has a visual duplication of rows. We repeated all the tuples from S_City and S_NoCity to specify they are in S_All as well. This "duplication" would happen automatically: S_All is still a derived value (i.e. the union) of the S_City and S_NoCity tuples within S_All.

Interacting with Multi-Relation Variables

The first question that might come to mind is "How do we interact with a multi-relation variable"? How do we insert relations into it and how do we query a multi-relation variable? For example, how do we ask:

Q4.1: What suppliers can be proven to be among those that we don't know their location?

It would appear we would have to add a way to specify which tuples we want to consider from the table. Something like "SELECT … FROM S_All.{S_NoCity}". This leads us to as much complexity as if we had separate tables. This approach may organize the relations and variables but it will not reduce the complexity of the database scheme.

Relation Distinguishing Tuple-Marks

Our problem with multi-relation variables is that it is as complex to distinguish among the different relations in a single multi-relation variable as if there were multiple separate variables. What if we used a simpler marker? Instead of having a marker outside the normal attributes of the relations, we can use a marker within the tuple's attributes to identify which relation the tuple belongs to. I will call this a Relation Distinguishing Tuple-Mark (tuple-mark for short) and use a dash "-- " to indicate it in a table. Using this approach for our example gives us:

S_All : (*S_City.2, S_NoCity.2*)

Relation	S#	SName	City	…
S_City.2	S1	Jones	London	…
S_City.2	S2	Smith	Bristol	…
S_City.2	S4	Eiffel	Paris	…
S_NoCity.2	S3	DuPont	--	…
S_NoCity.2	S5	Grid	--	…

The "-- " tuple-mark is not a type of City or any comment on the value of a tuple's city: that tuple has no attribute City. The tuple-mark identifies that the tuple belongs to a different relation than a tuple that does not have the mark. Supplier-S3 is part of the relation S_NoCity that does not have a city and the mark simply specifies that to be true.

This approach is only possible if each base relation has a distinguishing set of attributes. For our example this is true: S_City has attributes of {S#, SName, City, …} and S_NoCity has attributes of {S#, SName, …}. We could also have a relation S_NoName with attributes of {S#,City,…} and S_NoNameOrCity with attributes of {S#,…}. We could not handle our two relations S_UnknownCity and S_MuliNational, and we will return to this problem later.

The full relation S_All is functionally derived as the union of S_City and S_NoCity so it does not need to be distinguished (all tuples in the variable are also part of S_All). It is not completely clear how we query on S_City or S_NoCity instead of S_All, but that will be dealt with in the next section.

Interacting

We can now return to the question "How do we interact with this multi-relation variable?" How do we insert relations into it and how do we query a multi-relation variable? Specifically how do we answer the questions:

Q1.2: What names can be proven to be the name of a supplier?
Q2.2: What suppliers can be proven to be in London?
Q3.2: What suppliers can be proven not to be located in London?
Q4.1: What suppliers can be proven to be among those that we don't know their location?

Querying

A query over a multi-relation variable will only consider (be restricted to) the tuples for which the query is applicable. A query is applicable to a relation if all the attributes the query mentions exist for that relation. For example, the attribute SName exists in the relation S_City and S_NoCity, so a query "SELECT SName FROM S_All" would consider all the tuples:

S#	SName	…
S1	Jones	…
S2	Smith	…
S4	Eiffel	…
S3	DuPont	…
S5	Grid	…

If a query mentions City (e.g. "SELECT City FROM S_ALL") it would only consider the tuples that have a relation with an attribute City:

S#	SName	City	…
S1	Jones	London	…
S2	Smith	Bristol	…
S4	Eiffel	Paris	…

This allows us to answer Q1-Q3 and gives the same answers as our original S_City, S_NoCity, and S_All solution. So far so good, but to make any more progress requires extending the relational algebra.

Extending Projection

To answer Q4 we need to be able to retrieve the tuples that have distinguishing marks that identify them to be part of the S_NoCity relation. This sounds like a restriction but it can't be; we have the rule that tuples without an attribute will be immediately discarded so they will never make it to the restriction stage. We need to choose the tuples with a mark in a attribute and at the same time throw away that attribute (to prevent the tuple from being later discarded). We can do this by extending the "innermost" projection operation: the direct projection of our multi-relation variable "before" it is involved with the rest of the query.

Although not part of SQL it will be easiest to use and extend C.J. Date's projection notation:

A [X, Y, …, Z]

Which produces a relation value subset of A "obtained by eliminating all attributes not specified in the attribute commalist and then eliminating duplicate (sub)tuples from what is left)." [Date 95, Page 151]. In this notation Q1 can be answer by "S_ALL [SName]" and finding all the cities of suppliers can be answered by "S_ALL [City]".

To the projection notation I will add three new pieces: the include-all ("*") indicator, the eliminate ("-") prefix and the choose-marked ("!") prefix. The first two are primarily convenience additions to the projection syntax and they have nothing to do with multi-relation variables. They are added to the projection notation because they conceptually prepare it for the third addition, which will allow us to discriminate among different marked tuples.

Include-All

The include-all indicator is similar in meaning to the SQL version: it specifies that all the attributes of the relation should be used included in the projection and (by itself) is equivalent to the identity projection.

A ≡ A [*]

Eliminate

The eliminate prefix allows you to specify that an attribute should be eliminated from the projection instead of included in it. This allows you to say "S_ALL [*,-City]" and get

S#	SName	…
S1	Jones	…
S2	Smith	…
S4	Eiffel	…
S3	DuPont	…
S5	Grid	…

Instead of specifically stating the attributes to include "S_ALL [S#, SName, …]". If you have a projection that has only eliminated attributes you can leave off the include-all indicator:

S_ALL [-X, -Y, -Z] ≡ S_ALL [*, -X, -Y, -Z]

Choose-marked

Finally, the choose-marked prefix is similar to the eliminate prefix in that it will eliminate the attribute from the projection, but in addition it will also eliminate tuples (restrict the result to not include tuples) that have anything other than a tuple-mark (Relation Distinguishing Mark) for that attribute's value. For our example, we can select the tuples that are part of S_NoCity by the projection "S_ALL [*, !City]" which would give:

S#	SName	…
S3	DuPont	…
S5	Grid	…

Now we can answer Q4:

Q4.1: What suppliers can be proven to be among those that we don't know their location?

with "S_ALL [S#, !City]"

It may seem strange to have a projection operation perform a restriction but notice that this pseudo-restriction is complementary with the pseudo-restriction performed by including the attribute in the projection. "S_ALL [S#, SName, City]" gives:

S#	SName	City
S1	Jones	London
S2	Smith	Bristol
S4	Eiffel	Paris

"S_ALL [S#, SName, !City]" gives:

S#	SName
S3	DuPont
S5	Grid

In both cases the restriction occurs in the process of getting rid of tuple-marks in the particular attribute, in the first case it eliminates the marked tuples and in the second it eliminates the non-marked tuples.

Summary

To handle multi-relation variables in queries we proposed two additions to relational algebra.

1. A tuple with a marked attribute is excluded from all queries that mention that attribute
2. A choose-marked attribute projection will eliminate that attribute from the projection and will restrict the result to only include tuples that have a tuple-mark for that attribute

These will be the only additions to relational algebra. There is no need to add three-valued logic or change any domain operations' behavior. This is a much smaller change to relational algebra compared to supporting "normal" NULLs.

Adding tuples

To add a relation-distinguishing tuple-marked tuple to the database we can use the same approach as is used for NULLs. We can "pretend" to set the value of the attribute to "NULL" which will instead result in the tuple being marked as belonging to a different relation which does not include that attribute. As

mentioned earlier in chapter, to do this operation requires that each relation in a multi-relation variable be distinguishable by its set of attributes. For our example, we can use

INSERT INTO S_ALL (S#, SName, City) VALUES ("S6", "Java", NULL)

To add an S_NoCity tuple.

Relation	S#	SName	City	…
S_NoCity	S6	Java	--	…

Because NULLs are being used to identify the tuple's relation this may cause confusion with other insertion features that use NULLs as a flag. For example, if a particular table uses NULLs for a default value it will effectively change the inserted tuple's relation before completing the insertion. Actually adding a tuple with a particular relation (that does not include the defaulted attribute) may be impossible because of this.

Summary

The approach of using multi-relation variables allows us to simplify some databases and especially databases with missing information. If two or more relations are related to each other by them having similar but slightly different attributes than these relations can "share" the same multi-relation variable. Although adding this concept of multi-relation variables with marked-tuples adds complexity to the relational model, database scheme's can be made much less complex. What used to require two, three, or more base relation variables and multiple derived relation values (views) can now be merged into a single variable. Doing so has no impact on what can be express with the database: the defined and derivable relations are the same.

Using Relation Distinguishing Tuple-Marks made it possible to easily interact with multi-relation variables and only required a couple additions to the relational model:

1. A tuple with a marked attribute is excluded from all queries that mention that attribute
2. A choose-marked attribute projection will eliminate that attribute from the projection and will restrict the result to only include tuples that have a tuple-mark for that attribute

This appears to be a good approach because of its simplicity, its limited impact on the relational model, and possibly because of its similarity to how NULLs are used now. Tuple-marks have many advantages over NULLs as currently implemented or considered because:

1. Tuple-marks do not use three valued logic
2. Tuple-marks have no impact on domains and domain operations

We will return to further detail tuple-marks in a later chapter, but first we should consider how the approach handles the issues brought up in the NULL debate.

Tuple-Marks in the NULL Debate

Although Tuple-Marks could be proposed as an addition to the basic relational model (without NULLs) that is not the current historical context. There has been much discussion of the value of NULLs and their associated problems. Although tuple-marks are a completely different mechanism from NULLs, they are similar enough in use that it would be reasonable to see how tuple-marks would hold up in the debates. If tuple-marks have as many issues as NULLs than they won't be a better alternative to them.

Looking at previous NULL criticisms

I will select a few example criticisms from the authors who have written on the NULL topic. Some of these authors believe NULLs are a useful mechanism and some believe NULLs and 3VL cause major problems. In either case there are arguments about correct relational modeling and the correct behavior of relational operators (e.g. queries) in the face of missing information. I will place tuple-marks in the middle of the fray and say they can handle the arguments from both sides.

Codd-1

The example running through the first part of this paper was identical to the one in [Codd 90 § 9.2] where he discusses criticisms towards his approach for missing information. In that section he criticizes the results of a Special Value approach (at the time named as and mixed with the default value concept)

S_All : (*S_City, S_NoCity*)

S#	SName	City	…
S1	Jones	London	…
S2	Smith	Bristol	…
S3	DuPont	--	…
S4	Eiffel	Paris	…
S5	Grid	--	…

Which I will define as having the same relations (predicates) as earlier in this paper:

S_City: *There exists a supplier with identifier S#, who has the name SName, who is located only in city City, and …*

S_NoCity: *There exists a supplier with identifier S#, who has the name SName, and … and for which we do not know the city it is located in.*

S_All: *There exists a supplier with identifier S#, who has the name SName, and …*

The questions posed of this table are:

Q1: Find the suppliers in London

Q2: Find the supplier NOT in London

To rephrase these in terms the database can understand gives:

Q1a: Find the suppliers that can proved to be in London.

Q1b: Find the suppliers that can proved to be possibly in London.

Q2a: Find the suppliers that can proved to be NOT in London.

Q2b: Find the suppliers that can proved to be possibly NOT in London.

Q1a is easy: " SELECT S# FROM S WHERE City = 'London' ". By mentioning City we automatically restrict the considered tuples to those with a relation that includes City (so we drop S3 and S5).
Q1b requires a union between Q1a and "SELECT S# FROM S [!City]".
Q2a is easy again: " SELECT S# FROM S WHEREH City <> London "
Q2b again requires a union between Q2a and "SELECT S# FROM S [!City]".

Specifying each query correctly requires understanding the relation definitions above, but the tuple-marks give correct and predictable results.

McGoveran-1

David McGoveran [McGoveren 94a,b,c] covers properly normalizing a database to avoid needing NULLs. This approach is the best first approach possible and is what I superficially described in Chapter 5. But it can lead to a significant increase in tables and database complexity. Using a slightly modified version of one of McGoveran's examples:

Vehicles : (*PV, UPV, UPMV*)

VIN	Make	Model	Color
1	Ford	Escort	Green
2	Pontiac	Grand Prix	Red
3	Porsche	Carrera	--
4	Chrysler	LeBaron	--
5	DeLorean	--	--

PV: PaintedVehicles: *There exists a care with vin VIN, made by Make, of model Model, with color Color*
UPV: UnpaintedVehicles: *There exists…*
UPMV: UnpaintedModellessVehicles: *There exists…*

If we were to divide this table using the different relations involved we would get a total of three tables and would have to create two views to get "Makes" and "Makes and Models" for a user to easily query on. By keeping it all in one multi-relation variable with tuple-marks to distinguish the different relations we have much less complexity of the database scheme.

Date-1

C.J. Date discussed why three-valued logic is a mistake in [Date 95c] and in which he shows what answers supplied by 3VL are considered wrong answers. Based on the discussion of database knowledge and the use of tuple-marks I will show what answers tuple-marks would provide and respond to those that are considered wrong answers.

Wrong answers of the first kind

Date's first example of a wrong answer is the result of the query

```
SELECT E#
FROM EMP
WHERE Job = 'Clerk'
OR NOT Job = 'Clerk';
```

Assuming a relation and variable of:

EMP

E#	Name	Job

Date asserts that this query should return "all employee numbers" even when some tuples have JOB = NULL.

Right answers of the first kind

Although Date's assertion seems intuitive and correct, with our multi-relation variables we can see that Date is incorrect. If there were two relation variables:

EMP_OnJob

E#	Name	Job

EMP_NoJob

E#	Name

We would not expect that

```
SELECT E#
FROM EMP_OnJob
WHERE Job = 'Clerk'
OR NOT Job = 'Clerk';
```

would return any of the entries in EMP_NoJob.

It is also nonsensical to form the query:

```
SELECT E#
FROM EMP_NoJob
WHERE Job = 'Clerk'
OR NOT Job = 'Clerk';
```

but if the query were formed it would certainly return no tuples: Job is unmentioned and has no value, not either 'Clerk' or something other than 'Clerk'. We can see this by translating the query into a more formal question:

Q1: Find the employees that can proved to have a Job 'Clerk' or can be proved to be have a Job that is not 'Clerk'.

This does not mean we can't find employees who are not on the Job 'Clerk' simply because they aren't on any job, but we have to express it in a sensible manner given our relation predicates. The predicate of EMP_NoJob is likely to be:

EMP_NoJob: There exists an employee with id E# and name Name who is not currently on a job.

So our second query would be:

```
SELECT E#
FROM EMP_NoJob;
```

We simply select all the tuples from EMP_NoJob to determine who is not on any particular Job.

To get the answer to the question:

Q1: Find the employees that can proved to not be on the Job 'Clerk'

We can form the query:

```
(SELECT E#
FROM EMP_OnJob
WHERE Job = 'Clerk'
OR NOT Job = 'Clerk')
  UNION
(SELECT E#
FROM EMP_NoJob) ;
```

Requiring the separate expressions joined together makes sense: we are dealing with two different relations that have different attributes and predicates.

Muli-relation variables

For multi-relation variables we have the same problem and the same results but it is slightly more obscured by them sharing the same variable (table). Our relations and variables are:

EMP : (*Emp_OnJob, Emp_NoJob*)

E#	Name	Job

To get the answer to the question:

Q1: Find the employees that can proved to not be on the Job 'Clerk'

We must form the query:

```
(SELECT E#
FROM EMP
WHERE Job = 'Clerk'
OR NOT Job = 'Clerk')
  UNION
(SELECT E#
FROM EMP [!Job]) ;
```

Where the projection of EMP using a choose-marked allows us to get to the tuples that are members of the relation Emp_NoJob.

Unbound variables

The argument Date proposes for why the original query should return all tuples would be correct if a "NULL" represented an unbound attribute-value variable which during a Prolog-like unification (see [Clocksin+M 81]) actually took on all possible values of that attribute-value. In that case the condition would apply to all tuples (because all had a relation that mentions Job) and the WHERE would be a tautology. A relation distinguishing tuple-mark is not an unbound attribute-value variable (nor anything to do with an attribute-value) so this reasoning does not apply to it.

Summary

Tuple-marks do not cause wrong answers. The answers to queries over a multi-relation variable will be correct for the applicable relations for that variable. Users will have to remember that certain variables have multiple relations and must form queries appropriately to use the relations desired.

Date's arguments over the problems with 3VL causing wrong answers are cause by NULLs being considered an attribute value in a single relation. Tuple-marks are not attribute values.

Summary

I believe relation-distinguishing tuple-marks have correct behavior in the relational model and are superior to both attribute NULLs (3VL) and special values for handling missing information.

Further Details

This chapter fleshes out some more details of the concept of a Relation Distinguishing Tuple-Mark.

Types of Missing Information

Earlier I mentioned that there could be multiple meanings of missing information, which would have to have independent relations and relation variables. The example was

S_UnknownCity

S#	SName	…
S3	DuPont	…

S_MultiNational

S#	SName	…
S5	Grid	…

Which have identical attribute sets and the same "missing" information (the City) but have different meanings for the missing information. For example, maybe all MultNationals must be located in Delaware so their location is guaranteed to not be in London which is quite different from just not knowing the location (so it could be in London).

The tuple-marks discussed so far do not allow for a multi-relation variable to have two relations with the same set of attributes. There would be no way to distinguish among the tuples. There is nothing that prevents this if the appropriate mechanisms are added. If there were multiple marks available ("--m1--", "--m2--", etc.) and the choose-marked restriction could select which mark, we could have any number of relations in a single variable:

S_All : (*S_City.2, S_NoCity.2*)

Relation	S#	SName	City	…
S_City	S1	Jones	London	…
S_City	S2	Smith	Bristol	…
S_City	S4	Eiffel	Paris	…
S_UnknownCity	S3	DuPont	--m1--	…
S_MultiNational	S5	Grid	--m2--	…

We can answer the questions:

Q1: Find the suppliers that can proved to be in London.
Q2: Find the suppliers that can proved to be possibly in London.
Q3: Find the suppliers that can proved to be NOT in London.
Q4: Find the suppliers that can proved to be in Deleware

Using the following queries:

Q1: SELECT S# FROM S_All WHERE City = 'London'.
Q2: Q1 UNION (SELECT S# FROM S_All [!m1!City])
Q3: (SELECT S# FROM S_All WHERE City <> 'London') UNION (SELECT S# FROM S_All [!m2! City])
Q4: (SELECT S# FROM S_All WHERE City = 'Deleware') UNION (SELECT S# FROM S_All [!m2! City])

Although adding the functionality to the database appears to mostly a syntax addition, it is certainly a much more difficult mental model to keep track of. The invisibility of the multiple overlapping relations would significantly hinder a user from understanding the database.

Relations with no Attributes: Dee and Dum

What if we have only a single attribute column that can be tuple marked? For example:

Names : (*Name, NoName)*

Name
Jones
Smith
DuPont
--

What does it mean to have a "NULL" in the only column left? Well, to know that we have to define what relations we have. Suppose we have the following two base relations.

Name: *{Name} – There exists a supplier with the name Name.*

NoName: *{} – There exists a supplier for whom we do not know the name.*

So by adding a tuple with the relation NoName we specify that there exists a supplier for whom we do not know the name, where otherwise we know all the names of the suppliers. The relation for Names as a whole is:

Names: *{} – There exists a supplier*

We can ask several interesting (informally phrased) questions:

Q1: What are the names of the Suppliers? "Names [Name]"

Q2: Are there any suppliers without names? "Names [!Name]"

Q3: Are there any suppliers? "Names []"

Note that Q2 and Q3 will return a relation with no attributes and either zero rows or one row. These are Tweedle-Dum and Tweedle-Dee respectively (see [Warden 90]). Tuple-marks have no problem with them.

Outer Joins

Tuple-Marks can be used just as NULLs are normally used in outer joins. The number of intermediary relations will be quite extensive and it may be difficult to process the intermediary results. [To be completed]

Tuple-Marks in final results

It is expected that the final results of a query would include the tuple-mark "NULL"s for presentation and application purposes when no explicit mention of the attribute occurred within the query. For example, a query of

SELECT * FROM S_ALL

Would return

S#	SName	City	…
S1	Jones	London	…
S2	Smith	Bristol	…
S4	Eiffel	Paris	…
S3	DuPont	--	…
S5	Grid	--	…

Although

SELECT * FROM S_ALL [*]

would return

S#	SName	City	…
S1	Jones	London	…
S2	Smith	Bristol	…
S4	Eiffel	Paris	…

Note that the SELECT operation is performing two operations: it is projecting as part of the query and it is ordering the output columns for the application API. The second has nothing to do with the relational model itself so it would be nice to order the columns without implying a projection and without removing tuples that have tuple-marks in that column.

Other Approaches to Missing Information

Multi-relation variables and Tuple-Marks are meant to replace using standard NULLs, which require three-valued logic. On the other hand, tuple-marks are meant to augment most other approaches for representing missing information. These other approaches can be more appropriate in general or just for certain circumstances within a model.

Correct Normalization

Correct and fully determined normalization[6] should certainly be the first approach for any database. Having a couple extra relation variables in a database is simpler than adding the concepts of multi-relation variables and NULL-marked tuples. It is only when the scheme will get unmanageable from many "optional" attributes that multi-relation variables may be beneficial.

Special Values

Extending a domain to have special values is another valid approach for representing missing information[7]. This requires extending the domain so it understands how operations interact between "normal" values and the special values, but once that domain extension is accomplished it can be used by all attributes needing that domain throughout the database.

The main problem with (or feature of) special values is that they will be included in all "normal" value comparisons other than equals[8]. This leads to the special value results being returned for questions like

What suppliers can be proven to be in a location other than London?

This will return suppliers that have a location of "NotApplicable" and "Unknown" by default. These then have to be filtered out of the results if they aren't desired.

Tuple-marks have the opposite property. Marked tuples will be excluded from results of a query that has any type of operation on a marked attribute and in these cases the tuples will have to be explicitly included (usually by dealing with their relation independently) if they are desired.

Integrated multi-relation values

David McGoveran suggests [McGoveran 94c] multi-relation values and variables should be an integral part of the relational model. This would imply all relational operations could return sets that contain different types of tuples. Although an interesting concept, I am not sure that the added complexity would be worthwhile. The relational model is currently very simple but, even so, is frequently misunderstood. A more sophisticated model would be more likely to be confused.

[6] See [McGoveran 94c] and [Date 94b].
[7] See [Date 96,97a-c] and the other works by C.J. Date for a full discussion of the Special Values topic.
[8] This isn't strictly true with good domain support. With good domain support you can define the domain's operations as failing or implementing whatever truth table you desire. See [Date 97a].

Implementing Tuple-Marks in SQL

SQL does not have Relation Distinguishing Tuple-Marks, it has NULLs that use 3VL. SQL also does not have a choose-marked attribute projection operator. Even if everyone agreed that tuple-marks were the correct approach to missing information it would take a long time for the SQL standard and then the database vendors to change over. What do we do in the mean time?

Fortunately tuple-marks can be simulated within SQL. The only requirements for tuple-marks is for queries to include and eliminate particular tuples based on whether they are applicable. Although the SQL query engines won't do this for you automatically, you can manual code queries to have the correct results.

Returning to our original example of suppliers in S_All.

S_All : (*S_City, S_NoCity*)

Relation	S#	SName	City	…
S_City	S1	Jones	London	…
S_City	S2	Smith	Bristol	…
S_City	S4	Eiffel	Paris	…
S_NoCity	S3	DuPont	--	…
S_NoCity	S5	Grid	--	…

If we query over this table we need to simulate what a tuple-marked query engine would do. With a total of three relations in this table we have three options: we may want to ask questions that apply to S_All, S_City, or S_NoCity. This should be the first question for any query involving this table. Depending on the answer to that question we will have to:

1. Remove the rows that are inapplicable
2. Remove (or ignore) the columns that are inapplicable

To remove inapplicable rows we will ask either for rows "WHERE City NOT NULL" (to get S_City) or "WHERE City IS NULL" (for S_NoCity). To remove inapplicable columns we can "SELECT S#, SName, …" and leave out City. Together these two techniques (used carefully) will allow us to simulate the tuple-marked query engine.

For some of our previous examples:

Tuple-marked	Relation	SQL Version
SELECT City FROM S_All	S_City	SELECT City FROM S_All WHERE City NOT NULL
SELECT SName FROM S_All	S_All	SELECT SName FROM S_ALL
SELECT SName FROM S_All WHERE City = 'London'	S_City	SELECT SName FROM S_All WHERE City = 'London' AND City NOT NULL
SELECT SName FROM S_All [!City]	S_NoCity	SELECT SName FROM S_All WHERE City NOT NULL
(SELECT S# FROM S_All WHERE City <> 'London') UNION (SELECT S# FROM S_All [!City])	S_City, S_NoCity	(SELECT S# FROM S_All WHERE City <> 'London' AND City NOT NULL) UNION (SELECT S# FROM S_All WHERE City NOT NULL)

The SQL Version can be significantly more complex that the natural SQL but its results will be consistent with tuple-marked queries and two-valued logic instead of the strange behavior exhibited with SQL's 3VL.

Related Work

The references attached at the end of this document show some of the other work done on this subject. Just to give a feel for how close people's thoughts were to Tuple-marks I include a few quotes:

David McGoveran

> *"...The lack of a value for a property should automatically imply an appropriate modification of the relation predicate."*

C.J. Date

> *"To say that certain properties might not be held by certain of those entities is thus a contradiction in terms – it's to say that those entities aren't of that type after all!"*

David McGoveran

> *"Furthermore, conditional operators would then be understood as operations on multiple relationships (masquerading as single relationships) and having a multiple entity result (again masquerading as a single entity)."*

James R. Alexander

> *"...what my database "knows" and how it "knows" is clear: It knows only what it contains and what it contains, I put there.."*

David McGoveran

> *"Whenever a value in a non-key base table column is optional (that is, the database designer permits it to be null), the column represents a conditional property or meaning criterion. Such columns indicate that multiple entity types are being represented in a single table. Each of these entities have distinct relation predicates"*

Summary

The topic of missing information and using NULLs in the relational model has had a very good debate over the years. The information from these arguments has been very useful and has improved technology and practices. Unfortunately this debate has not yet lead to a generally agreeable solution.

I believe multi-relation variables and Relation Distinguishing Tuple-Marks are part of that generally agreeable solution. Tuple-marks provide all the benefits of NULLs (and Codd's marks) without the corresponding problems. Tuple-marks allow a single relation variable (table) to hold tuples from multiple different relations and the query engine can determine which relations and tuples are appropriate for a particular query. This is accomplished without any negative side effects: it does not require three-value logic or any changes to domains and domain operations.

The only cost of tuple-marks is the conceptual complexity of having multiple relations within a single variable. This causes a tradeoff between having many simple fully determined relation variables and having fewer complex multi-relation variables. Tuple-marks seem to be the minimum cost approach for making this tradeoff. They can even be manually implemented using current SQL databases.

Multi-relation variables and tuple-marks can enhance the relational model in a positive manner. They help model missing information and interdependent relations. Along with proper normalization, fully determined relations, and full domain support, Relation Distinguishing Tuple-Marks provide a general solution to missing information. In this case, something is much better than nothing.

References

Alexander 94a James R. Alexander. Letter in *Database Programming & Design*, 7(3): 11, March 1994.

Clocksin+M 81 W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag 1981.

Codd 90 E.F. Codd. *The Relational Model for Database Management, Version 2.* Addison-Wesley, Reading, MA, 1990

Codd+D 93a E.F. Codd and C.J. Date. "Much Ado about Nothing" in *Database Programming & Design*, 6(10): 45-53, October 1993.

Date 90 C.J. Date. *Relational Database Writings 1985-1989*. Addison-Wesley, Reading, MA, 1990.

Date 92 C.J. Date. *Relational Database Writings 1989- 1991.* Addison-Wesley, Reading, MA, 1992.

Date 94 C.J. Date. "Relations and Their Meaning" in *Database Programming & Design*, 7(12): 19-22, December 1994.

Date 95 C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, Reading, MA, 1995.

Date 95b C.J. Date. *Relational Database Writings 1991- 1994.* Addison-Wesley, Reading, MA, 1995.

Date 95c C.J. Date. "Why Three-Valued Logic Is a Mistake" in [Date 95b] : 22-29.

Date 96 C.J. Date. "Faults and Defaults – Part 2 of 5" in *Database Programming & Design*, 9(12): 15-19, December 1996.

Date 97a C.J. Date. "Faults and Defaults – Part 3 of 5" in *Database Programming & Design*, 10(1): 16-20, January 1997.

Date+H 97 C.J. Date with Hugh Darwin. *A Guide to the SQL Standard – Fourth Edition.* Addison-Wesley, Reading, MA, 1997.

Date+M 94b C.J. Date and David McGoveran. "A New Database Design Principle" in *Database Programming & Design*, 7(7): 46-53, July 1994.

Ferg 94 Stephen Ferg. Letter in *Database Programming & Design*, 7(7): 9-10, July 1994.

McGoveran 93a David McGoveran. "Nothing from Nothing" in *Database Programming & Design*, 6(12): 33-41, December 1993.

McGoveran 94a David McGoveran. "Classical Logic: Nothing Compares 2 U" in *Database Programming & Design*, 7(1): 54-56, January 1994.

McGoveran 94b David McGoveran. "Can't Lose What You Never Had" in *Database Programming & Design*, 7(2): 43-48, February 1994.

McGoveran 94c David McGoveran. "It's In the Way That You Use It" in *Database Programming & Design*, 7(3): 54-63, March 1994.

Warden 90 Andrew Warden. "Table_Dee and Table_Dum" in [Date 90].