File: Multirelations.d8.doc
Printed at: 12:07 on Thursday, 30 September, 2010

Multirelations

by Hugh Darwen

*** REVIEW DRAFT ***
(now superseded: please see ***Note to Reviewers*** on page 2)

Reviewers are expected to be familiar with the relational model of data and preferably with The Third Manifesto *([2]) and* ***Tutorial D***.

CONTENTS

1. Introduction 2
2. Terminology 5
3. The multirelation 6
4. Multirelation types 9
5. The multirelation selector 9
6. Multirelation operators 10
6.1 Participant extraction 11
6.2 Multiprojection 11
6.3 MR-projection 12
6.4 MR-extraction 12
6.5 MR-join 13
6.6 MR-union 14
6.7 MR-intersection 15
6.8 MR-difference 15
6.9 NOT MR_MATCHING 16
6.10 MR-restriction 17
6.11 MR_extension 22
6.12 MR_renaming 22
6.13 Internal join 23
6.14 Internal decomposition 25
7. Multirelation comparison 26
8. Multirelation variables 27
9. Multirelvar constraints 27
9.1 IS_EMPTY 27
9.2 MR-keys 29
9.3 Participant keys 29
9.4 MR-6NF 29
10. Normal forms for multirelvars 30
11. Multirelvar update operators 30
11.1 Assignment 30
11.2 MR-insertion 31
11.3 MR-deletion 31
11.4 MR-update 32
12. Virtual relvars based on multirelvars 32
13. Interpretation of a multirelation 32
14. Applications of multirelations 34
15. Some topics for further investigation 35
16. Acknowledgements 36
17. References 37

Note to reviewers:

Chris Date is collaborating with me on a revision of this paper for possible inclusion in a forthcoming book.[1] Among the points to be addressed are:

1. Section **13, Interpretation of a multirelation**, proposes a predicate in which one of the parameters, *P*, stands for a predicate. This is nonsense: the tuples of a multirelation do not contain attribute values denoting predicates. The section will be either revised or dropped.
2. The problematical section **6.10, *MR-restriction***, will be revised to address some of the problems.
3. The structure of the paper will be revised to present the operators in a more appropriate sequence.
4. The operator definitions will be accompanied by illustrative examples to aid comprehension.
5. The Introduction will be updated to mention other approaches that have recently come to light: Darren Duncan's approach in Muldis D (http://www.muldis.com), and an approach using subtyping suggested by Erwin Smout (see his joint paper with me, *How to Handle Missing Information Using S-by-C* at http://www.thethirdmanifesto.com).

Suggestions from reviewers of previous drafts have been taken into account and further suggestions are still welcome.

1. Introduction

In connection with the so-called "missing information" problem, we have seen several suggestions over the years involving data structures that look somewhat like relations in that their bodies consist of tuples, but differ from relations in that not all the tuples in the same body are necessarily of the same heading. The latest, which prompted this discussion paper, is in reference [6], where the author floats the idea that such structures be admitted to the database, subject to certain restrictions on their use which I do not need to mention here. An earlier one is reference [9], by my friend the late Adrian Larner. Larner did discuss his idea with me but I failed to understand it properly and I still fail to understand it fully now.

Examples are given in Figure 1 on the next page of the kind of thing I (and presumably the others I have referred to) have in mind, shown in tabular form.

The vacant spaces in these tables do not represent appearances of NULL, as they might if it were an SQL table. Nor do they represent the empty character string, `''`. Rather, they represent the complete absence of the attribute in question from the tuple in question. The

[1] The book, *Database Explorations: Essays on The Third Manifesto and Related Topics,* was published in 2010. Further information is at www.thethirdmanifesto.com.

attribute in question is of course the one whose name appears at the top of the column in which the gap appears; and the tuple in question is the one whose attribute values are given by the row in which the gap appears.

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	
S4	Clark		London
S5	Adams	30	Athens
S6			Rome
S7			

S#	P#	QTY
S1	P1	300
S1	P2	200
S2	P1	
S3	P2	200
S4	P4	

Figure 1: The Suppliers and Shipments multirelations

Superficially it would seem that the idea means abandoning relations, and that is why in the past I have tended to reject such proposals out of hand. The publication of reference [6], an article criticizing some of my own (joint) work, compelled me to produce a detailed justification of my rejection of it. To do that I needed to work out exactly what such a scheme would have to look like in a database language; that, I thought, would clearly expose the fatal flaws in the idea. However, it occurred to me that the kind of structure under consideration, whose body is a set of tuples, does in a sense include one or more *relations* that could be extracted from it. If we can extract from such a structure whatever relations we wish to work with, then the relational model might be considered to be preserved. But—and it is a big "but"—although it could be argued that Codd's "Information Principle"[2] (that all information in the database shall be cast in the form of relations) is adhered to, the kind of object we are now talking about is certainly not a relation. With that caveat in mind, I still wished to see where the idea might lead.

It is easy to spot the five nonempty relations "included" in my "suppliers" example. One of them consists of the tuples with S# values S1, S2, and S5, all of which have the same four

[2] A more appropriate name is The Principle of Uniform Representation (or The Principle of Uniformity of Representation), as proposed in [3].

attributes. Another consists of the single (ternary) tuple with S# S3, and so on. But how many *empty* relations does it "include" in that same sense? And what exactly *is* that sense? Such questions, and many more, need to be answered, I thought, before I can begin to think about the operators that might be defined for operating on such objects.

What to call these objects? Because I was expecting that my paper would show why the idea should be rejected, I first considered the derogatory term *pseudorelation*, but as something approaching a precise definition began to unfold I became less convinced that the idea should be rejected out of hand; so I decided instead to call them *multirelations*. Unfortunately I then found that this term has been used before, for purposes that are (a) different from the one at hand and (b) multifarious. Because of (b), and my inability to think of any suitable alternative, I have decided to stick with *multirelation* for the time being. I do think my use of it is more appropriate than some others that I have found (one of them being apparently for an SQL table, the body of which is in general a bag or, as the SQL international standard has it, a *multiset*). See also references [5] and [7], with which I may or may not be significantly at odds.

I am now presenting my findings, such as they are, for discussion, without at this time offering much in the way of my own opinion of them. The reader is invited to consider the following questions:

1. Are there any "fatal flaws" in the scheme? (I realize the term is undefined, but it certainly subsumes logical unsoundness and inconsistency.)
2. Is the scheme too complicated?
3. Does the scheme suffer from too much that is counterintuitive?
4. If the answer of any of the above is "yes", can the scheme be suitably revised? (Presumably not if the answer to the first is "yes".)

I should stress the point that I do not regard the problem addressed by multirelations as a previously unsolved one. For example, I still stand by the solution I described in reference [1], which conforms to reference [2], Chapter 4: *The Third Manifesto* (*TTM*). David McGoveran adopts a similar approach in reference [10] and the two approaches are briefly compared by C.J. Date in reference [4]. However, many people have raised psychological objections[3] to that kind of approach without proposing any alternative approaches, apart from abandoning relations altogether (as SQL does). Fabian Pascal ([11]) discusses McGoveran's approach and appears to be moving towards developing it along the lines of the present paper, putting the decomposition "under the covers", so to speak. But he doesn't spell out any details in the way I attempt to spell them out here.

The most commonly expressed objections to decomposition are that it gives rise to an excessive multiplicity of relvars and an excessive multiplicity of required constraints. From bitter experience, I can to some extent understand those objections. In 1982 the very first

[3] Some people also raise performance objections, but these tend to assume the common SQL DBMS designs of the present time, which tend to punish decomposition instead of encouraging it.

customer of the relational DBMS Business System 12, available to users of IBM's time-sharing Bureau Service, was an organization offering information to investors about various companies. Over 300 separate items of information were identified as of possible interest to investors but only a very few of those items were available for every company. Under both my proposal and McGoveran's at least 300 separate relvars would be needed, together with a huge number of foreign key and other constraints. In 1982 we crammed everything into a single relvar, using "impossible" values such as -9999999.99 to indicate missing information. Those "impossible" values gave rise to all sorts of traps for the unwary and all sorts of complications in queries to avoid those traps.

Multirelations are intended to address those objections and to avoid those traps and complications.[4] Whether they can succeed in doing so is the matter on which this paper seeks to open discussion.

2. Terminology

In this paper I use some terms that may not be familiar to all my reviewers. These are:

common attribute An attribute <*an, at*> is a common attribute of relations *r1*…*rn* (*n*>1) or of tuples *t1*…*tm* (*m*>1) if and only if each of *r1*…*rn* (or *t1*…*tm*) has an attribute named *an* of type *at*.

joinable

(a) Relations *r1* and *r2* are joinable if and only if the set theory union of their headings is a heading (i.e., they are joinable unless *r1* and *r2* each have an attribute with the same name that is not a common attribute).

(b) Tuples *t1* and *t2* are joinable if and only if their set theory union is a tuple. It follows from this definition that (a) the union of the headings of *t1* and *t2* is a heading, and (b) for each common attribute of *t1* and *t2* the value of that attribute in *t1* is equal to its value in *t2*.

(c) Relations *r1, ..., rn* are joinable if and only if for every pair {*rj, rk*} (1≤*j*≤*n* and 1≤*k*≤*n*), *rj* and *rk* are joinable; similarly, tuples *t1, ..., tm* are joinable if and only if for every pair {*tj, tk*} (1≤*j*≤*m* and 1≤*k*≤*m*), *tj* and *tk* are joinable.

[4] One of my reviewers (Adrian Hudnott) has pointed out that multirelations save the designer not only from having to name each relvar in the decomposition approach, but also from having to decide exactly how to do that decomposition. Also, the structure of a given decomposition might have to be changed if the business rules change with regard to what attribute values can be missing and what combinations of attribute values must be either all absent or all present. With multirelations, such changes can be reflected just in the constraints, without any change to the database structure.

join From [3]: 1. (*Dyadic case*) Let relations *r1* and *r2* be [joinable]. Then the join of *r1* and *r2*, *r1* JOIN *r2*, is a relation with heading the set theory union of the headings of *r1* and *r2* and with body the set of all tuples *t* such that *t* is the set theory union of a tuple from *r1* and a tuple from *r2*. 2. (*N-adic case*) Let relations *r1*, *r2*, ..., *rn* (*n*≥0) be [joinable]. Then the join JOIN { *r1*, *r2*, ..., *rn* } is defined as follows: if *n*=0, the result is TABLE_DEE; if *n*=1, the result is *r1*; otherwise, choose any two relations from the set and replace them by their (dyadic) join and repeat this process until the set consists of only one relation *r*, which is the overall result.

union From [3]: 1. (*Dyadic case*) The union of two relations *r1* and *r2*, *r1* UNION *r2*, where *r1* and *r2* are of the same type *T*, is a relation of type *T* with body the set of all tuples *t* such that *t* appears in either or both of *r1* and *r2*. 2. (*N-adic case*) The union of *n* relations *r1*, *r2*, ..., *rn* (*n*≥0), UNION { *r1*, *r2*, ..., *rn* }, where *r1*, *r2*, ..., *rn* are all of the same type *T*, is is a relation of type *T* with body the set of all tuples *t* such that *t* appears in at least one of *r1*, *r2*, ..., *rn*. *Note:* If *n*=0, (a) some syntactic mechanism, not shown here, is needed to specify the pertinent type *T*, and (b) the result is the empty relation of that type.

subtuple Tuple *t1* is a subtuple of tuple *t2* if and only if *t1* is a subset of *t2*. If in addition *t1* ≠ *t2*, then *t1* is a *proper* subtuple of *t2*.

supertuple Tuple *t1* is a supertuple of tuple *t2* if and only if *t1* is a superset of *t2*. If in addition *t1* ≠ *t2*, then *t1* is a *proper* supertuple of *t2*.

3. The multirelation

Now I need to pin down precisely what a multirelation *is*. First, notice that I wrote that a multirelation *includes* relations, rather than *containing* them. That is because it turns out to be more convenient to regard such relations as subsets[5] of the including[6] multirelation, rather than as elements of it. I use the term *participant* (or *participating relation,* when a need for clarity demands) for such a relation. Note that the body of a participant with heading *H* consists of those tuples in the including multirelation that have heading *H*; thus, no proper

[5] Loosely speaking. Really it is the bodies of such relations that are subsets of the "body" of the multirelation ("body" in quotes because it is not a real body in the sense in which that term is defined for relations). And the headings of such relations are subsets of the "heading" of the multirelation.
[6] Also loosely speaking, for the same reason.

subset of a participant's body is itself the body of some participant. Note also that every tuple of a multirelation is a member of the body of exactly one of its participants. More precisely:

1. Let *mr* be a multirelation. Like a relation, *mr* has a heading and a body. The heading of *mr* is identical in appearance to that of a relation but does not mean quite the same thing as the heading of a relation. I therefore call it an MR-heading. The body of a multirelation, like that of a relation, is a set of tuples, but it differs from the body of a relation in that the tuples do not have to have the same heading. I therefore call it an MR-body.

   Let *MRH* be the MR-heading of *mr* and *MRB* its MR-body. The *degree* of *mr* is the number of attributes in *MRH* and the *cardinality* of *mr* is the number of tuples in *MRB*.

2. The heading of each tuple in *MRB* is a subset of *MRH*.

3. A relation *pr* is a *participant* (or *participating relation*) in *mr* if and only if (a) the heading *PH* of *pr* is a subset of *MRH* and (b) the body *PB* of *pr* consists of all and only those tuples in *MRB* whose heading is *PH*. It follows that if the degree of *MRH* is *n*, then the number of participants in *mr* is 2^n. It also follows that if the degree of *MRH* is 0, then there is exactly one participant, this being either TABLE_DEE or TABLE_DUM. As each tuple in *MRB* appears in exactly one participant, the bodies of the nonempty participants form a partitioning of *MRB*.

   Note that the Supplier multirelation depicted in Figure 1 has 11 empty participants[7] in addition to the 5 visible, nonempty ones.

   We say that *mr* is *empty* if and only if *MRB* is empty (contains no tuples)—equivalently, if and only if every participant in *mr* is empty. It should be clear that emptiness here refers to a lack of tuples, not participants, because, as we have already noted, the number of participants in *mr* is 2^n, where *n* is the cardinality of *MRH*.

4. It follows from the definitions of *MRH* and *MRB* that if relations *r1* and *r2* are participants in *mr,* then *r1* and *r2* are joinable. This fact allows the attribute names of *mr* to be used for purposes very similar to those of a relation, as we shall see.

5. If no more than one participant in *mr* is nonempty, then the tuples in *MRB* have all the same heading, a subset of *MRH*. So *MRB* is the body of a relation, whose heading is that subset. One is tempted to remark that *mr* is to all intents and purposes a relation.

[7] One of these is TABLE_DUM. If, instead, TABLE_DEE were a participant, then an all-blank row would appear in the table.

I define IS_RELATION(*mr*)[8] to be *true* if no more than one participant in *mr* is nonempty, otherwise *false*.

When IS_RELATION(*mr*) is *true,* one of the participants in *mr* is called the *prime participant*. This is the single nonempty participant if it exists; otherwise it is the participant whose heading is *MRH*.

For expository purpose only I define RELATION(*mr*) to denote the prime participant in *mr* if IS_RELATION(*mr*) is *true* and to be otherwise undefined. (Note that this construct cannot appear in a language that supports static type checking, because the heading of the prime participant is not known at compile time. However, the effect of RELATION(*mr*) can be obtained using the operator defined later in Section ***6.1, Participant Extraction***.)

RELATION(*mr*) clearly deserves a counterpart for converting in the opposite direction, as it were, so MULTIRELATION(*r*), where *r* is a relation, will denote the multirelation whose MR-heading is the heading of *r* and whose MR-body consists of the tuples of *r*. Note that in general it is not the case that MULTIRELATION(RELATION(*mr*)) = *mr*, because the heading of the prime participant of *mr* might be a proper subset of the MR-heading of *mr*. However, RELATION(MULTIRELATION(*r*)) = *r* for all relations *r*, thanks to the careful choice of prime participant of *mr* in the case when *mr* is empty.

6. The term *common attribute*, as defined for relations and tuples, is applicable in the same sense to multirelations.

Note that it is wrong to say that *mr* actually *is* a relation when it satisfies the given condition (hence the inaccuracy, previously noted, of the keyword IS_RELATION). Several of the operators I propose in this paper for operating on multirelations are, for obvious reasons, not normally defined as available for operating on relations; and those normally defined for operating on relations are not defined for multirelations.

[8] The keyword IS_RELATION is slightly but dangerously inaccurate, for reasons given later. Suggestions for its replacement would be welcome.

4. Multirelation types

Just as tuples and relations are subdivided into types distinguished by their headings, the set of multirelations can usefully be subdivided into *multirelation types* distinguished by their MR-headings. In the style of **Tutorial D** a specific multirelation type would be denoted by a keyword followed by a list of attribute definitions enclosed in braces. For example:

```
MULTIRELATION { S# S#,
                SNAME NAME,
                STATUS INTEGER,
                CITY CHAR }
```

5. The multirelation selector

TTM (reference [2]) requires every value to be denotable by some literal. In general, a literal of type *T* is an invocation of some *selector* for type *T* in which each argument to the invocation is itself a literal. Just as the selector invocation denoting relation *r* needs to specify both the heading and the body of *r*, the multirelation selector invocation denoting *mr* needs to specify both the MR-heading and the MR-body of *mr*. Again, the style of **Tutorial D** can be followed closely. For example:

```
MULTIRELATION { TUPLE { S# S#('S1'),
                        SNAME NAME('Smith'),
                        CITY 'London' },
                TUPLE { S# S#('S2'),
                        SNAME NAME('Jones'),
                        STATUS 30 } }
```

Here the MR-heading is implied by the headings of the tuples of the MR-body. Note that the two tuples in this multirelation have different headings. Where two or more tuples have attributes with the same name, those attributes must also be of the same type. This follows from the joinability rule for the participating relations. The default MR-heading is the union of the tuple headings (even if the tuple list is empty). The MR-heading could be given explicitly; thus, the above example is short for

```
MULTIRELATION { S# S#,
                SNAME NAME,
                STATUS INTEGER,
                CITY CHAR }
              { TUPLE { S# S#('S1'),
                        SNAME NAME('Smith'),
                        CITY 'London' },
                TUPLE { S# S#('S2'),
                        SNAME NAME('Jones'),
                        STATUS 30 } }
```

but in fact any superset of the default MR-heading would be valid. This leads to the following observation:

> If *MRB* is the MR-body of a multirelation of heading *MRH*, then for every MR-heading that is a superset MRH^+ of *MRH* there is a multirelation of MR-heading MRH^+ and MR-body *MRB*. Moreover, if *a* is an attribute of *MRH* but not of the heading of any tuple in *MRB*, then *MRB* is the body of some multirelation of heading *MRH* MINUS{*a*}.

There are just two multirelations of type MULTIRELATION{ }. They are called MR_DEE and MR_DUM. MR_DEE is in fact MULTIRELATION{ }{TUPLE { }} and MR_DUM is MULTIRELATION{ }{ } (which can be abbreviated to MULTIRELATION{ }). It can clearly be seen that MR_DEE and MR_DUM are in fact multirelation counterparts of the well known relations TABLE_DEE and TABLE_DUM. IS_RELATION(MR_DEE) and IS_RELATION(MR_DUM) are both *true*. RELATION(MR_DEE) is TABLE_DEE and RELATION(MR_DUM) is TABLE_DUM. In fact, TABLE_DEE is the *only* participant in MR_DEE and TABLE_DUM is the only participant in MR_DUM.

6. Multirelation operators

I examine operators that operate on multirelations to yield relations and ones that operate on multirelations to yield multirelations. My examination is with an eye to what might be needed for practical purposes in a database language such as **Tutorial D**; I do not attempt, here, to develop a formal specification for an algebra of multirelations. I do note some interesting properties (such commutativity, associativity, idempotence) but I do not attempt to identify some minimal or otherwise agreeable set of primitive operators.

Some of the operators are have obvious relational counterparts. Where a relational counterpart exists, a concrete syntax might well use the same operator name for both the relational operator and its multirelational counterpart. In this paper I do not employ such "overloading", because I think distinct names are more useful for the purposes of discourse.

The examples used in this section assume that S and SP are the names of multirelation variables. The declared type of S is

```
MULTIRELATION { S# S#,
                SNAME NAME,
                STATUS INTEGER,
                CITY CHAR }
```

and that of SP is

```
MULTIRELATION { S# S#,
                P# P#,
                QTY INTEGER }
```

Sample values for these variables are shown in Figure 1.

Let *mr*, *mr1*, and *mr2* be multirelations. Let their MR-headings be *MRH*, *MRH1*, and *MRH2*, respectively and let their MR-bodies be *MRB*, *MRB1*, and *MRB2*, respectively.

I start with operators that yield relations.

6.1 Participant extraction

First, an operator to extract a participating relation *pr* of *mr*, given the attribute names of the heading of *pr*. Example:

```
PARTICIPANT { S#, SNAME, STATUS } FROM S
```

Let relation *r* = PARTICIPANT {*a1*, ..., *an*} FROM *mr*. Then the heading *HR* of *r* is the subset of *MRH* specified by {*a1*, ..., *an*} and the body of *r* consists of just those tuples of *MRB* that have heading *HR*. Equivalently, *r* is that participant in *mr* whose heading is {*a1*, ..., *an*}.

In **Tutorial D**, wherever a list of attribute names can appear, referring to attributes of a given relation *r*, this list can be preceded by ALL BUT to denote the attributes of *r* other than those listed. The same construct can be used here, so the given example is equivalent to

```
PARTICIPANT { ALL BUT CITY } FROM S
```

A similar observation applies, where appropriate, to all the operators proposed in this paper.

6.2 Multiprojection

This is just a generalization of relational projection but for distinctness I'll call it by the preposition, ONTO, that is often used in connection with projection (we project a given relation *onto* a given subset of its attributes). Example:

```
S ONTO { S#, SNAME, STATUS }
```

Let relation *r* = *mr* ONTO{*a1*, ..., *an*}. Then the heading *HR* of *r* consists of the attributes {*a1*, ..., *an*} of *MRH* and the body *BR* of *r* is such that tuple *t* is an element of *BR* if and only if *t* has heading *HR* and is a subtuple (subset) of some tuple in *MRB*. Equivalently, the body of *r* is the union of the bodies of the projections onto {*a1*, ..., *an*} of all and only those participants in *mr* whose heading is some superset of {*a1*, ..., *an*}.

Note that multiprojection reduces to relational projection when IS_RELATION(*mr*) is *true*.

An SQL counterpart of the given example is

```
SELECT DISTINCT S#, SNAME, STATUS
FROM S
WHERE S# IS NOT NULL
```

```
AND SNAME IS NOT NULL
AND STATUS IS NOT NULL
```
[9]

That's all I define, in this paper, by way of operators that operate on multirelations to yield relations. I turn to operators for deriving multirelations from multirelations. To discover the operators that I describe next, I first considered multirelational counterparts of the usual relational operators. I do not say much about the possible usefulness of these operators. They are chosen for their *teachability*, under an assumption that if multirelations are useful at all then these operators are as useful in the context of multirelations as their relational counterparts are in the context of relations.

6.3 MR-projection

Given multirelation *mr,* MR-projection yields a multirelation *ms* formed by discarding zero or more attributes. The attributes to be discarded are either specified explicitly or implied by specifying explicitly the ones to be retained

The syntax is as for ONTO apart from the operator name, which is MR_ONTO. Examples:

```
S MR_ONTO { S#, SNAME, STATUS }
S MR_ONTO { ALL BUT CITY }
```

These two examples are equivalent.

Let *ms = mr* MR_ONTO {*a1, ..., an*}. Then *ms* is the multirelation whose MR-heading consists of the attributes specified in the invocation and whose MR-body consists of just those tuples that appear in the body of the projection of some participant *p* in *mr* onto the intersection of {*a1, ..., an*} and the attributes of *p*.

Note that IS_RELATION(*ms*) is *true* whenever IS_RELATION(*mr*) is *true*. In this case, RELATION(*ms*) is some projection of RELATION(*mr*). Note also that TABLE_DEE is a participant in *ms* whenever the heading of some nonempty participant in *mr* contains none of the attributes *a1, ..., an*.

An SQL counterpart of the given examples is

```
SELECT DISTINCT S#, SNAME, STATUS
FROM S
```

6.4 MR-extraction

The purpose of this operator is to extract from a given multirelation all the tuples that contain values for certain specified attributes and do not contain values for certain other attributes, the result being presented as a multirelation.

---

[9] The WHERE condition cab be abbreviated to (S#, SNAME, STATUS) IS NOT NULL, but this is rather less clear.

Note to reviewers: Section **6.11, *MR-restriction*** suggests an alternative approach that might be preferred as more intuitive and more flexible. It uses unconventional operators named PRESENT and ABSENT, for use in conditional expressions in certain special contexts only. If this approach is acceptable, then we can perhaps dispense with MR-extraction. *End of note to reviewers.*

The syntax uses the operator name MR_WITH and the key words PRESENT and ABSENT operand to specify attributes required to be present and absent, respectively. Example:

```
S MR_WITH PRESENT { S#, SNAME } ABSENT { STATUS }
```

Let *ms* = *mr* MR_WITH PRESENT {*a1, ..., an*} ABSENT {*b1, ..., bm*}. Let *incl* be the attributes specified by {*a1, ..., an*} and let *excl* be the attributes specified by {*b1, ..., bm*}. Then *ms* is the multirelation whose MR-heading *MSH* is *MRH – excl* and whose MR-body consists of just those tuples in *MRB* whose heading is a superset of *incl* and a subset of *MSH*.

If PRESENT is omitted, PRESENT { } is assumed by default. Similarly, if ABSENT is omitted, ABSENT { } is assumed by default.

In the given example, *MSH* is {S# S#, SNAME NAME, CITY CHAR}. A counterpart of the example in SQL would be

```
SELECT DISTINCT S#, SNAME, CITY
FROM S
WHERE S# IS NOT NULL
AND   SNAME IS NOT NULL
AND   STATUS IS NULL
```

Note that IS_RELATION(*ms*) is *true* whenever IS_RELATION(*mr*) is *true*. If in addition ABSENT { } is specified or implied, then *ms* = *mr*. IS_RELATION(*ms*) is also *true* whenever the union of *incl* and *excl* contains every attribute of *mr*.

6.5 MR-join

MR-join is the multirelational counterpart of relational join. The syntax is as for relational join apart from the operator name, which is MR_JOIN. Example:

```
S MR_JOIN SP
```

Let *ms* = *mr1* MR_JOIN *mr2*, where *MRH1* and *MRH2* are such that each participant in *mr1* is joinable with each participant in *mr2*. Then *ms* is the multirelation whose MR-heading *MSH* is the union of *MRH1* and *MRH2* and whose MR-body *MSB* consists of just those tuples that can be formed by taking the union of *t1* and *t2* where *t1* is a tuple in *MRB1* and *t2* is a tuple in *MRB2* (in which case *t1* and *t2* are joinable). It follows that *MSB* is such that *pr1* JOIN *pr2* is a participant in *ms* if and only if *pr1* is a participant in *mr1* and *pr2* is a participant in *mr2*.

Note that IS_RELATION(*ms*) is *true* whenever IS_RELATION(*mr1*) and IS_RELATION(*mr2*) are both *true*. In this case, if *pr1* and *pr2* are the prime participants in *mr1* and *mr2*, respectively, then RELATION(*ms*) = *pr1* JOIN *pr2*.

Like relational join, MR-join is commutative and associative. Moreover, MR_DEE is the identity under MR-join, so an *n*-adic version with $n \geq 0$ can also be defined. MR_JOIN { } = MR_DEE and MR_JOIN { *mr* } = *mr*.

Unlike relational join, MR-join is not idempotent, even though S MR_JOIN S does happen to yield S if S has the value shown in Figure 1. For a counterexample, let MRX be MULTIRELATION { TUPLE { X 1 }, TUPLE { Y 1 } }. Then the MR-body of MRX MR_JOIN MRX contains the tuple TUPLE { X 1, Y 1 } in addition to those of the MR-body of MRX.

I considered defining a second kind of multirelation join in which only those tuples whose headings include the common attributes of the operand MR-headings participate and the others are discarded (so to speak). But this can easily be expressed using MR_WITH and MR_JOIN. For example:

```
( S MR_WITH PRESENT { S# } )
MR_JOIN
( SP MR_WITH PRESENT { S# } )
```

6.6 MR-union

The syntax is as for relational union apart from the operator name, which is MR_UNION. Example:

```
S MR_UNION SP
```

Let *ms* = *mr1* MR_UNION *mr2*. Then *ms* is the multirelation whose MR-heading *MSH* is the union of *MRH1* and *MRH2* and whose MR-body is the union of *MRB1* and *MRB2*. It follows that *pr1* UNION *pr2*, where *pr1* and *pr2* are of the same type, is a participant in *ms* if *pr1* is a participant in *mr1* and *pr2* is a participant in *mr2*. And if *pr3* is a participant in *mr1* such that no participant in *mr2* is of the same type as *pr3,* then *pr3* is a participant in *ms*. Similarly, if *pr4* is a participant in *mr2* such that no participant in *mr1* is of the same type as *pr4,* then *pr4* is a participant in *ms*. In fact, each participant in *ms* is either a participant in *mr1*, or a participant in *mr2*, or the relational union of participant in *mr1* and a participant in *mr2* (and could be more than one of these three).

Note that IS_RELATION(*ms*) is *true* whenever IS_RELATION(*mr1*) and IS_RELATION(*mr2*) are both *true* and *mr1* and *mr2* are of the same type. In this case, if *pr1* and *pr2* are the prime participants in *mr1* and *mr2*, respectively, then RELATION(*ms*) = *pr1* UNION *pr2*.

Like relational union, MR-union is commutative and associative. It is also, in contrast with MR-join, idempotent.

As with MR-join, an *n*-adic version of MR-union can be defined. As with relational union, the type of the result must be specified for the niladic case.

6.7 MR-intersection

The syntax is as for relational intersection apart from the operator name, which is `MR_INTERSECT`. Example:

```
S MR_INTERSECT SP
```

Let *ms* = *mr1* `MR_INTERSECT` *mr2*. Then *ms* is the multirelation whose MR-heading *MSH* is the intersection of *MRH1* and *MRH2* and whose MR-body *MSB* consists of just those tuples that are contained in both *MRB1* and *MRB2*. In other words, *MSB* is such that *pr1* `INTERSECT` *pr2*, where *pr1* and *pr2* are of the same type, is a participant in *ms* if and only if *pr1* is a participant in *mr1* and *pr2* is a participant in *mr2*.

Note that `IS_RELATION(`*ms*`)` is *true* whenever `IS_RELATION(`*mr1*`)` and `IS_RELATION(`*mr2*`)` are both *true*, and also in several other circumstances (e.g., when *MRB1* and *MRB2* are disjoint or either of them is empty).

Like relational intersection, MR-intersection is commutative, associative, and idempotent. However, whereas relational intersection is a special case of join, MR-intersection is not a special case of MR-join.

As with MR-join, an *n*-adic version of MR-intersection can be defined. As with relational intersection, the type of the result must be specified for the niladic case.

6.8 MR-semijoin

The syntax is as for relational semijoin (`MATCHING`) apart from the operator name, which is `MR_MATCHING`. Example:

```
S MR_MATCHING SP
```

Let *ms* = *mr1* `MR_MATCHING` *mr2*. Then *ms* is the multirelation of MR-heading *MRH1* whose MR-body *MSB* is such that for each tuple *t1* of *MRB1*, *t1* is a member of *MSB* if and only if there is some tuple *t2* in *MRB2* that is joinable with *t1*.

Note that `IS_RELATION(`*ms*`)` is *true* whenever `IS_RELATION(`*mr1*`)` is *true*. In this case, if *pr1* is the prime participant in *mr1* and *pr2* `...` *prn* are the participants in *mr2*, then `RELATION(`*ms*`) = UNION {(`*pr1* `MATCHING` *pr2*`), ..., (`*pr1* `MATCHING` *prn*`)}`.

Whereas relational intersection is just a special case of relational semijoin, MR-intersection is not a special case of MR-semijoin. For consider, there might be a tuple *t1* in *MRB1* that is

not also in *MRB2* but is nevertheless joinable with some tuple in *MRB2*. In that case, *t1* appears in the result of *mr1* MR_MATCHING *mr2* but does not appear in the result of *mr1* MR_INTERSECT *mr2*.

6.9 MR-difference

The *MR-complement* of *mr* is the multirelation whose participants are the relational complements of the participants in *mr*. Its MR-body therefore consists of just those tuples with heading some subset of *MRH* that do not appear in *MRB*.

We do not define an operator to denote the MR-complement of *mr*, the reason being the same as that given for not defining one to denote relational complement. However, we can and do define the multirelational counterpart of relational difference (MINUS). The syntax is as for relational difference apart from the operator name, which is MR_MINUS. Example:

```
S MR_MINUS SP
```

Let *ms* = *mr1* MR_MINUS *mr2*. Then *ms* is the multirelation whose MR-heading is *MRH1* and whose MR-body consists of just those tuples that are members of *MRB1* and not members of *MRB2*. It follows that *pr1* MINUS *pr2*, where *pr1* and *pr2* are of the same type, is a participant in *ms* if *pr1* is a participant in *mr1* and *pr2* is a participant in *mr2*. And if *pr3* is a participant in *mr1* such that no participant in *mr2* is of the same type as *pr3,* then *pr3* is a participant in *ms*. In fact, each participant in *ms* is either a participant in *mr1*, or the relational difference of a participant in *mr1* and a participant in *mr2* (and could be both of these).

Note that IS_RELATION(*ms*) is *true* whenever IS_RELATION(*mr1*) is *true*. In this case, if *pr1* is the prime participant in *mr1* and *pr2* is a participant in *mr2* with the same heading as *pr1*, then RELATION(*ms*) = *pr1* MINUS *pr2*. If there is no such participant in *mr2*, then RELATION(*ms*) = *pr1*.

6.10 NOT MR_MATCHING

The syntax is as for relational NOT MATCHING apart from the operator name, which is NOT MR_MATCHING. Example:

```
S NOT MR_MATCHING SP
```

Let *ms* = *mr1* NOT MR_MATCHING *mr2*. Then *ms* is the multirelation of MR-heading *MRH1* whose MR-body *MSB* is such that for each tuple *t1* of *MRB1*, *t1* is a member of *MSB* if and only if there is no tuple *t2* in *MRB2* that is joinable with *t1*.

Whereas relational difference is just a special case of relational NOT MATCHING, MR-difference is not a special case of NOT MR_MATCHING. For consider, there might be a tuple *t1* in *MRB1* that is not also in *MRB2* but is nevertheless joinable with some tuple in *MRB2*. In

that case, *t1* does not appear in the result of *mr1* NOT MR_MATCHING *mr2* but does appear in the result of *mr1* MR_MINUS *mr2*.

Note that IS_RELATION(*ms*) is *true* whenever IS_RELATION(*mr1*) is *true*. In this case, if *pr1* is the prime participant in *mr1* and *pr2 ... prn* are the participants in *mr2*, then RELATION(*ms*) = (...(*pr1* NOT MATCHING *pr2*)...) NOT MATCHING *prn*.

6.11 MR-restriction

Note to reviewers: This operator has raised some particular difficulties and might even be the one to kill the entire idea. My proposals are accompanied by much more discussion than I have given with the other operators and should be treated as tentative; I ask you to consider them especially carefully—countersuggestions would be most welcome! *End of note to reviewers.*

Compared with its relational counterpart, multirelational restriction turns out to be a complicated affair, reminiscent of SQL's WHERE clause when appearances of NULL in the input table need to be coped with. Relational restriction is usually expressed as

```
r WHERE cond
```

where *r* is a relation and *cond* is a conditional expression including zero or more references to attributes of *r*. We can consider *cond* as a predicate, whose parameters are represented by those attribute references. Each tuple *t* in *r* provides a value for each of those attributes, allowing *cond* to be instantiated to give a truth-valued expression *p*. If *p* is *true*, then *t* is said to *satisfy cond* and appears in the body of the result; otherwise *t* does not appear in that body.

Note that each tuple of *r* appears in the result of either *r* WHERE *cond* or *r* WHERE NOT (*cond*) and no tuple of *r* appears in both results. This is a manifestation of *the law of the excluded middle*.

Note also that the relation *r* WHERE *cond* represents the extension of the predicate *pr* AND *pcond*, where *pr* is a predicate whose extension is represented by *r* and *pcond* is the predicate represented by *cond* as already described.

Now, a multirelational counterpart of relational restriction might be *mr* MR_WHERE *cond*. Example:

```
S MR_WHERE STATUS = 20
```

Unfortunately, the attribute reference STATUS is not necessarily defined for each tuple of S. In Figure 1, for example, it is undefined for suppliers S4, S6 and S7. We would normally expect such cases to give rise to run-time exceptions. To avoid those exceptions the user would have to write, for example,

```
( S MR_WITH PRESENT { STATUS } ) MR_WHERE STATUS = 20
```

The result would contain all those tuples of S that have a STATUS value that is equal to 20 and would not contain any tuples having a STATUS value that is not equal to 20; nor would it contain any tuples that do not appear in S. Those tuples in S that do not have a STATUS attribute are also excluded from the result, by that invocation of MR_WITH. Intuitively, we might say that such tuples cannot possibly be said to satisfy the given condition and therefore should not appear in the result. So we might be tempted to make invocation of MR_WITH implicit in the definition of MR_WHERE:

> Let *ms* = *mr* MR_WHERE *cond*. Then *ms* is the multirelation of MR-heading *MRH* whose MR-body consists of just those tuples in *mr* that satisfy *cond*. The heading of a tuple that satisfies *cond* must contain every attribute referenced in *cond*.

This definition suffers from several problems.

First, consider

```
S MR_WHERE STATUS = 20 OR CITY = 'London'
```

Looking at Figure 1, we can see that the tuple for supplier S1 clearly satisfies the given condition, but what about the tuple for supplier S4? That satisfies CITY = 'London' but lacks a STATUS. And if Figure 1 represented an SQL table, the row for supplier S4 certainly *would* satisfy SQL's WHERE condition STATUS = 20 OR CITY = 'London'. It seems, then, that the definition should be revised along lines such as this:

> Let *ms* = *mr* MR_WHERE *cond*. Then *ms* is the multirelation of MR-heading *MRH* whose MR-body consists of just those tuples in *mr* that satisfy *cond*. The heading of a tuple that satisfies *cond* must contain such attributes as may be needed to permit *cond* to be evaluated for that tuple.

There is gross arm-waving here but my assumption is that that system would make use of the theorems of propositional logic to evaluate, for example, invocations of dyadic logical operators where the truth value of one of the operands is sufficient to determine the truth value of the result.

Note that IS_RELATION(*ms*) is *true* whenever IS_RELATION(*mr*) is *true*. In this case, if *pr* is the prime participant in *mr*, then RELATION(*ms*) = *pr* WHERE *cond*, unless the heading of *pr* fails to "contain such attributes as may be needed [etc.]", in which case RELATION(*ms*) is empty.

But the definition is still problematical. Next, note that if *cond* is a tautology, then the result is not necessarily equal to the operand (as it is with relational restriction). For example:

```
S MR_WHERE STATUS = STATUS
```

The body of the result consists of all the tuples of S except for those that have no STATUS attribute. Thus, in spite of our careful avoidance of 3-valued logic, we still suffer from some of the counterintuitive phenomena that we have complained about in SQL's treatment of NULL. For the SQL counterpart of this example would be

```
SELECT *
FROM S
WHERE STATUS = STATUS
```

and the result "loses" those rows of the table S where NULL appears in place of a value for STATUS.

Next, note that the relation *mr* MR_WHERE *cond* does not in general represent a predicate of the form *pr* AND *pcond*, as described for relational restriction. For consider what happens when the proposition represented by an S tuple that lacks a STATUS value is ANDed with the predicate "STATUS = 20", as in

"Supplier S4 is named Clark and is located in London and STATUS = 20"

The extension of that monadic predicate consists of the following single proposition:

"Supplier S4 is named Clark and is located in London and 20 = 20"

If the restriction condition is STATUS < 20 instead, then the extension consists of as many propositions as there are integers less than 20 (INTEGER being the declared type of the attribute STATUS). By contrast, S MR_WHERE STATUS = 20 and S MR_WHERE STATUS < 20 are both defined to contain no tuple at all for supplier S4.

Next, consider

```
S MR_WHERE NOT ( STATUS = 20 )
```

Under the proposed definition, the result would contain all and only those tuples of S that have a STATUS value that is not equal to 20. But intuitively we would probably expect the result to contain all the tuples of S apart from those that have a STATUS value that is equal to 20. That would require the tuples that have no STATUS value to appear in the result too. To make that happen we would have to write something like

```
S MR_WHERE NOT ( STATUS = 20 )
MR_UNION
( S MR_WITH ABSENT { STATUS } )
```

[10]

Perhaps we need another shorthand to cater for such cases. One approach might be to provide two restriction operators, according to the treatment required of tuples for which the restriction condition cannot be evaluated. Suppose these two operators are MR_WHERE as defined and MR_UNLESS, defined as follows:

[10] and hope that the optimizer is clever enough to work out that this can be evaluated in one pass of the body of S!

Let *ms* = *mr* MR_UNLESS *cond*. Let {*a1, ..., an*} be the attributes of *mr* referenced in the condition *cond*. Then *ms* is the result of

```
MR_UNION {
   ( mr MR_WHERE NOT ( cond ) ),
   ( mr MR_WITH ABSENT { a1 } ),
   ...,
   ( mr MR_WITH ABSENT { an } ) }
```

Recall that in general *r* WHERE *cond* and *r* WHERE NOT (*cond*) yield relations with disjoint bodies whose union is the body of *r*; also that in SQL a similar observation does *not* apply to SELECT * FROM *t* WHERE *cond* and SELECT * FROM *t* WHERE NOT (*cond*). If both MR_WHERE and MR_UNLESS are available, then we can note with some degree of satisfaction that in general *mr* MR_WHERE *cond* and *mr* MR_UNLESS *cond* yield multirelations with disjoint MR-bodies whose union is the MR-body of *mr*.

Now we must consider cases where *cond* is a compound of negated and non-negated elements, such as in

```
S MR_WHERE STATUS = 20 OR NOT ( STATUS < 40 )
```

If, as shown, we choose MR_WHERE, we will exclude S tuples that have no STATUS value even though such tuples might be thought of as not having one that is less than 40; so MR_UNLESS is probably a more suitable choice. However, the situation is less clear-cut when the two disjuncts reference different attributes, as in

```
S MR_WHERE STATUS = 20 OR NOT ( CITY = 'Paris' )
```

This excludes S tuples that have no STATUS value but do have a CITY value other than 'Paris'. If we use MR_UNLESS instead, we will include S tuples that have CITY = 'Paris' but have no STATUS value.

The situation is different again if conjunction is used instead of disjunction:

```
S MR_WHERE STATUS = 20 AND NOT ( CITY = 'Paris' )
```

This "correctly" excludes S tuples that have no STATUS value but do have a CITY value other than 'Paris', but it also excludes tuples that have STATUS = 20 and no CITY value. If we use MR_UNLESS instead, we will include S tuples that have no STATUS value.

It appears, then, that even offering a choice of MR-restriction operators does not satisfactorily address the issue of appropriate treatment of tuples against which the restriction condition cannot be evaluated. We might therefore need an additional operator to test for the absence, from a tuple, of one or more attributes. For example, if *t* is a tuple,

```
ABSENT ( t, { STATUS, CITY } )
```

evaluating to *true* if `t` lacks the specified attributes, otherwise *false*. Now, the tuple on which a restriction condition operates is implied and in fact we have no way of denoting it explicitly, but we could perhaps allow the tuple operand to be implied, as in

```
S MR_WHERE ABSENT { STATUS, CITY }
```

This would be equivalent to

```
S MR_WITH ABSENT { STATUS, CITY }
```

yielding the multirelation whose MR-body consists of those tuples of `S` whose headings lack both `STATUS` and `CITY`. That would put the user in control over the inclusion or exclusion of tuples that lack attributes referenced in restriction conditions. Examples:

```
S MR_WHERE NOT ( STATUS = 20 ) OR ABSENT { STATUS }

S MR_WHERE STATUS = 20 OR NOT ( CITY = 'Paris' )
                          OR ABSENT { CITY }

S MR_WHERE STATUS = 20 AND ( NOT ( CITY = 'Paris' )
                               OR ABSENT { CITY } )
```

It is tempting but wrong to equate `ABSENT` with SQL's `IS NULL` operator. `IS NULL` takes a list of arbitrary expressions and returns *true* if and only if each expression is "the null value". `ABSENT` operates on a tuple and a list of attribute names, returning *true* if and only if each of the specified attribute names is not the name of an attribute of that tuple.

Of course it would be natural to offer `PRESENT` as the inverse of `ABSENT`. In that case we could perhaps dispense with `MR_WITH` (see Section **6.4, *MR-extraction***), because

```
mr MR_WITH PRESENT {a1, ..., am}
```

can now be expressed equally succinctly as

```
mr MR_WHERE PRESENT {a1, ..., am}
```

which would allow combinations of `MR_WITH` and `MR_WHERE` to be expressed in a single `MR_WHERE` invocation, such as:

```
S MR_WHERE NOT ( STATUS = 20 ) AND PRESENT { CITY, STATUS }
```

Note carefully that attribute references appearing as operands of `ABSENT` and `PRESENT` denote attributes *per se*, not attribute values.

6.12 MR_extension

The syntax is as for relational extension[11] apart from the operator name, which is MR_EXTEND. Example:

```
MR_EXTEND S ADD ( STATUS + 10 AS NEWSTATUS )
```

Let *ms* = MR_EXTEND *mr* ADD (*expr1* AS *a1*). Then *ms* is the multirelation whose MR-heading *MSH* is *MRH* extended with the attribute *a1*. The MR-body of *ms* consists of tuples *t'* derived from each tuple *t* of *MRB* as follows:

- Each attribute value of *t* is an attribute value of *t'*.
- If and only if every attribute referenced in *expr1* is an attribute of *t,* then *a1* is an attribute of *t'* with the value denoted by *expr1*. If the operators PRESENT and ABSENT are provided, as suggested in connection with MR-restriction, then invocations of those operators are allowed to appear in *expr1*.

Note that IS_RELATION(*ms*) is *true* whenever IS_RELATION(*mr*) is *true*. In this case, if *pr* is the prime participant in *mr*, then RELATION(*ms*) is some extension of *pr* (possibly the identity extension), depending on the attributes referenced in *expr1*.

As with relational extension, a multiple form of MR-extension can also be defined. MR_EXTEND *mr* ADD (*expr1* AS *a1, ..., exprn* AS *an*) is equivalent to MR_EXTEND (... (MR_EXTEND *mr* ADD (*expr1* AS *a1*)) ...) ADD (*exprn* AS *an*).

6.13 MR_renaming

The syntax is as for relational renaming apart from the operator name, which is MR_RENAME. Example:

```
MR_RENAME S ( S# AS S1 )
```

MR_RENAME *mr* (*a1* AS *b1*) is equivalent to

```
( MR_EXTEND mr (a1 AS b1 ) )
MR_ONTO { ALL BUT a1 }
```

Note that IS_RELATION(*ms*) is *true* whenever IS_RELATION(*mr*) is *true*. In this case, if *pr* is the prime participant in *mr*, then RELATION(*ms*) is some renaming of *mr* (possibly the identity renaming), depending on the attributes referenced by *a1 ... an*.

[11] At the time of writing we are considering an extension to the EXTEND operator in **Tutorial D** (as defined in [2]) to incorporate the functionality of SUMMARIZE.

As with relational renaming, a multiple form of MR-renaming can also be defined. `MR_RENAME mr ( a1 AS b1, ..., an AS bn )` is equivalent to is equivalent to `MR_RENAME (... ( MR_RENAME mr ( a1 AS b1 ) ) ...) ( an AS bn )`.

I do not at this time define any further `MR_` counterparts of relational operators. Apart from `TCLOSE`, other relational operators in **Tutorial D** are all defined in terms of ones for which I have defined `MR_` counterparts and it is quite possible that `MR_` counterparts could be defined for these too. Examples are `GROUP/UNGROUP`, `COMPOSE`, `MATCHING`, and `SUMMARIZE`. Nor, at this time, do I consider any possible `MR_` counterparts of `GROUP` and `UNGROUP` involving multirelation-valued attributes in place of relation-valued attributes. Regarding `SUMMARIZE`,[12] my current thinking is that the `PER` operand should remain a relation, but I have not given a great deal of thought to the matter.

My next multirelational operator is one that has no relational counterpart. It looks rather bizarre but it comes into its own later, when we consider a possibly useful role for multirelation variables ("multirelvars") in database designs.

6.14 Internal join

Consider the following multirelation:

```
MULTIRELATION { TUPLE { S# S#('S1'),
                        SNAME NAME('Smith'),
                TUPLE { S# S#('S1'),
                        CITY 'London' } }
```

Those two tuples are joinable. If we join them and produce a result in which their join appears in their place, we obtain (in this example) a multirelation of cardinality 1 from one of cardinality 2, the result in this case being

```
MULTIRELATION { TUPLE { S# S#('S1'),
                        SNAME NAME('Smith'),
                        CITY 'London' } }
```

Internal join (`INTRAJOIN`) derives a multirelation from a given multirelation by joining certain sets of its joinable tuples. For example:

```
INTRAJOIN ( S )
```

Let *ms* = `INTRAJOIN ( mr )`. Then *ms* is a multirelation of MR-heading *MRH*. The MR-body *MSB* of *ms* is defined as follows.

For each subset *SSH* of *MRH*, let {*SS1, ..., SSn*} be a set of participants in *mr* whose common attributes are *SSH*. For each tuple *x* of heading *SSH*, let *SST* be the set of tuples whose projection onto *SSH* is *x* such that each is taken from at most one of {*SS1, ..., SSn*}. Then tuple *t* is a member of *MSB* if and only if

[12] See footnote 11.

- *t* is the join of the tuples of some such *SST* and
- *t* is not a proper subtuple of *t'* where *t'* is the join of the tuples of some such *SST*.

Intuitively, *mr* and *ms* can be regarded as "informationally equivalent" (see section 6.15, *Internal decomposition*), but *ms* is such that if *t1* and *t2* are distinct tuples of *ms*, then the union of *t1* and *t2* is not a tuple, whereas *mr* does not necessarily have that property (and if it does, then *mr* = *ms*). Note that the result of the example, `INTRAJOIN ( S )`, is equal to `S`, because there are no two distinct tuples in `S` that are joinable. However, if the multirelation `S'` were as shown in Figure 2, then `INTRAJOIN (S')` would yield `S`. `S'` deliberately exhibits some redundancy, as shown in the rows for S2 and S3 in Figure 2. S2's city is shown twice, as is S3's status; and the third row for S2 merely confirms that supplier S2 exists. One of the effects of internal join is to eliminate such redundancy.

S#	SNAME	STATUS	CITY
S1	Smith		
S1		20	
S1			London
S2	Jones	10	Paris
S2			Paris
S2			
S3		30	
S3	Blake	30	
S4	Clark		London
S5	Adams	30	Athens
S6			Rome
S7			

Figure 2: S'—an alternative arrangement for suppliers
(not recommended!—note redundancy)

Note that `IS_RELATION(`*ms*`)` is *true* whenever `IS_RELATION(`*mr*`)` is *true*, in which case `RELATION(`*ms*`) = RELATION(`*mr*`)`. The present author cannot help additionally noticing the unimportant fact that `TABLE_DEE` is a participant in *ms* if and only if *mr* = `MR_DEE`.

As we will see later, the internal join of a multirelation can be regarded as a useful canonical form, based on a certain equivalence relationship that we can observe between *mr* and `INTRAJOIN (` *mr* `)`.

6.15 Internal decomposition

Consider again the multirelation given by `INTRAJOIN S`:

```
MULTIRELATION { TUPLE { S# S#('S1'),
                        SNAME NAME('Smith'),
                        CITY 'London' } }
```

As we have seen, this is the internal join of

```
MULTIRELATION { TUPLE { S# S#('S1'),
                        SNAME NAME('Smith') },
                TUPLE { S# S#('S1'),
                        CITY 'London' } }
```

and also of

```
MULTIRELATION { TUPLE { S# S#('S1'),
                        SNAME NAME('Smith') },
                TUPLE { S# S#('S1'),
                        CITY 'London' },
                TUPLE { S# S#('S1') } }
```

Each of these three multirelations can be interpreted as the single proposition, "S1 is a supplier under contract and S1 is named Smith and S1 is located in London" and are thus informationally equivalent. Like the internal join, that last form might be regarded as a useful canonical form, in which case we will need an operator to yield it. The required form is specified by giving that subset of the MR-heading that is to appear in the heading of each tuple of the result. For example:

```
INTRADECOMPOSE S ON { S# }
```

Let *ms* = `INTRADECOMPOSE` *mr* `ON {`*a1, ..., an*`}`. Then *ms* is a multirelation of MR-heading *MRH*. The MR-body *MSB* of *ms* is such that tuple *t* is a member of *MSB* if and only if:

(a) *t* is a subtuple of some tuple in *MRB*, and

(b) the heading of *t* includes the attributes {*a1, ..., an*} and at most one other attribute.

ms is informationally equivalent to *mr* only when each of the attributes *a1, ..., an* appears in the heading of each tuple of *mr*.

Note that if the heading of *t* is a proper superset of {*a1, ..., an*}, then the tuple that is the projection of *t* onto {*a1, ..., an*} also appears in *MSB*. `IS_RELATION(`*ms*`)` is *true* whenever *mr* is empty or the only nonempty participant in *mr* is the one whose attributes are precisely {*a1, ..., an*}.

7. Multirelation comparison

If every participant in *mr1* is such that its body is a subset of the body of some participant in *mr2,* then it follows that *MRB1* is a subset of *MRB2*. Therefore, just as in **Tutorial D** we can write *r1* ⊆ *r2* where *r1* and *r2* are relations, we can also write *mr1* ⊆ *mr2*. However, *r1* and *r2* are required to be relations of the same type. I propose no analogous restriction on the types of *mr1* and *mr2*, because none is logically necessary. It might be thought appropriate at least to require the MR-headings of the comparands not to be disjoint, but it cold be argued that the advantages of such a restriction are too slight to warrant it. If the MR-headings *are* disjoint, then *mr1* ⊆ *mr2* is *true* if and only if *mr1* is empty.

Now, the relational concept of *inclusion dependency* will need a multirelational counterpart. For example, in the suppliers-and-shipments database the relvar `SP` is subject to an inclusion dependency to the effect that the projection of `SP` onto `S#` is at all times a subset of the projection of `S` onto `S#` (and because `S#` is a key of `S`, this inclusion dependency is commonly referred to as a *foreign key* constraint). When `S` and `SP` are multirelation variables instead of being relvars, we can define the inclusion dependency in similar fashion but using MR-projection instead of projection. We can say that body of the MR-projection of `SP` onto `S#` must at all times be a subset of the MR-body of the MR-projection of `S` onto `S#`. The comparison can be expressed thus:

```
SP MR_ONTO { S# } ⊆ S MR_ONTO { S# }
```

Note that if `S#` is absent from any tuple of `SP`, then `TABLE_DEE` is a participant in the first operand and the comparison yields *true* only if `S#` is also absent from some tuple of `S`.

It is easily shown that comparison using ⊆ is sufficient for relations. For example, if we wish to see if every tuple of relation *r1* is a subtuple of some tuple of relation *r2,* we can first check that every attribute of r1 is also an attribute of *r2* and, that being the case, compare *r1* with the projection of *r2* onto the attributes of *r1*. However, ⊆ is not sufficient for all the comparisons that can be envisaged on multirelations. Additional operators would be needed to support tests such as the following, if there is any need for them:

- every tuple of multirelation *mr1* is a subtuple of some tuple of multirelation *mr2*
- every tuple of *mr1* is a supertuple of some tuple of *mr2*
- no tuple of *mr1* is a subtuple of some tuple of *mr2*
- no tuple of *mr1* is a supertuple of some tuple of *mr2*

Seeing no compelling need for these tests, I offer no suggestions for supporting them at this time.

8. Multirelation variables

The declaration of a multirelation variable, or multirelvar, looks very similar to that of a relvar. Here is an example:

```
VAR S BASE MULTIRELATION
           { S# S#, SNAME NAME, STATUS INTEGER, CITY CHAR }
           MR_KEY { S# } ;
```

The declared type of this multirelvar is `MULTIRELATION{S# S#, SNAME NAME, STATUS INTEGER, CITY CHAR}` but the `MR_KEY` specification (see **Multirelvar constraints**, below) further constrains the values that can be assigned to it, just as a `KEY` specification constrains the values that can be assigned to a relvar. Note, however, that whereas every relvar has at least one key, not every multirelvar has an MR-key.

9. Multirelvar constraints

The single comparison operator ("is subset of") defined for relations theoretically suffices for *relational* database constraint declarations, but in practice there is a compelling need for useful shorthands to address certain common requirements. We have already seen, in Section **7, Multirelation comparison**, that there are certain tests that cannot be expressed using just that single comparison operator. I do not address those possible requirements at this time, but I do propose some shorthands that immediately spring to mind as needed in connection with the problem that the author of [6] was addressing when he floated the idea.

9.1 IS_EMPTY

I see no reason to have a distinguishing name such as `MR_IS_EMPTY` for this shorthand. `IS_EMPTY(`*`mr`*`)` is *true* if and only if the body of *`mr`* contains no tuples at all—in other words, if and only if, for every participant *`pr`* in *`mr`*, `IS_EMPTY(`*`pr`*`)` is *true* (in which case *`mr`* has a prime participant that is empty).

Consider our example multirelvar, `S`. If this is to be used as a replacement for the usual *relvar* of that name, that decision will have been motivated by a desire to be able to record information about suppliers for whom not all of the information that *might* be recorded about a supplier is available. The totality of information that *might* be available for any given supplier is indicated by the MR-heading of `S`. That MR-heading implies that every subset of it is the heading of some participant in the multirelation assigned to `S`. In practice we will want to make sure that certain of those participants are empty at all times. For example, we will definitely want to make sure that every participant is empty whose heading does not contain the attribute `S#`. We can express that like this:

```
IS_EMPTY ( S MR_WITH ABSENT { S# } )
```

or, if `ABSENT` is supported in `MR_WHERE`:

```
IS_EMPTY ( S MR_WHERE ABSENT { S# } )
```

I venture to think that constraints of this kind will be very common if multirelvars are used at all. Indeed, further shorthands based on it will probably be desired. For consider the case of a football club. It has a fixture list. For each match in the fixture list, the result is eventually entered. So we could envisage a multirelvar whose nonempty participants at all times number no more than three, one for the matches that need to be scheduled but for which no date has been agreed yet, one for those that are scheduled but haven't yet been played, and one for the ones that have been played. The multirelvar probably includes attributes named GOALS_FOR and GOALS_AGAINST. Obviously whenever one of these attributes has a value for a particular match, then so must the other. So we will need

```
IS_EMPTY ( FIXTURE MR_WITH PRESENT { GOALS_FOR }
                           ABSENT  { GOALS_AGAINST } )
AND
IS_EMPTY ( FIXTURE MR_WITH PRESENT { GOALS_AGAINST }
                           ABSENT  { GOALS_FOR } )
```

or, using ABSENT and PRESENT with MR_WHERE,

```
IS_EMPTY ( FIXTURE MR_WHERE ( PRESENT { GOALS_FOR }
                              AND
                              ABSENT  { GOALS_AGAINST } )
                         OR ( ABSENT  { GOALS_FOR }
                              AND
                              PRESENT { GOALS_AGAINST } )
```

Such constraints get increasingly more complex as the number of attribute values that must appear together if they appear at all increases.

The other side of the coin—cases where certain pairs of attributes are mutually exclusive in the sense that if a value appears for one no value must appear for the other, in the same tuple, can also be expressed using IS_EMPTY:

```
IS_EMPTY ( S MR_WITH PRESENT { STATUS, REASON } )
```

Here I have added the attribute REASON to multirelvar S. REASON is used only for cases where there is no status value (for the reason given). But now we might want also to insist that every tuple has either a STATUS value or a REASON value. This does it:

```
IS_EMPTY ( S MR_WITH ABSENT { STATUS, REASON } )
```

Using ABSENT and PRESENT with MR_WHERE these can be combined as

```
IS_EMPTY ( S MR_WHERE PRESENT { STATUS, REASON }
                      OR ABSENT  { STATUS, REASON } )
```

Such constraints, however, will not by themselves serve the kind of purpose I have indicated. They will need to be accompanied by the multirelvar analogue of relational keys ...

9.2 MR-keys

An MR-key (of a multirelvar) has the same properties of uniqueness and irreducibility as are defined for keys. Note in particular that the scope of uniqueness now covers several relations, namely all the participants in the multirelation assigned to the variable to which the MR-key constraint applies.

I propose the `MR_KEY` shorthand shown in the example because I strongly suspect that if multirelvars are to be used at all, then at least one such constraint should be specified in every case as a matter of good practice. (By "every case" here, I mean the variables that constitute the database *per se*. I do not include ones that might exist from time to time to contain results of evaluating queries.)

`MR_KEY {`*`k1, ..., kn`*`}` specified for multirelvar *`mrv`* is equivalent to

```
IS_EMPTY ( mrv MR_WITH ABSENT {k1, ..., kn} )
AND COUNT ( mrv ONTO {k1, ..., kn} ) = COUNT ( mrv )
```

(I assume from here onwards that the operator `COUNT`, yielding the cardinality of its operand, is defined for multirelations as well as relations.)

9.3 Participant keys

`PARTICIPANT_KEY {`*`k1, ..., kn`*`}` specified for multirelvar *`mrv`*, where *`k1, ..., kn`* are names of attributes of *`mrv`*, defines a *participant key* for *`mrv`*.

A participant key *`PK`* of a multirelvar *`mrv`* is a subset of the MR-heading of *`mrv`* such that at all times, for each nonempty participant *`p`* in *`mrv`*, *`PK`* is included in the heading of *`p`* and constitutes a key[13] of *`p`*.

Note that if *`K`* is an MR-key of *`mrv`*, then *`K`* is *a fortiori* a participant key of *`mrv`*. However, if *`PK`* is a participant key of *`mrv`*, then *`PK`* is not necessarily an MR-key of *`mrv`*.

9.4 MR-6NF

MR-6NF is a constraint on a multirelvar to the effect that the only participants that are permitted to be nonempty are those whose headings have no more than one attribute in addition to those of a participant key. In other words, if *`mrv`* is a multirelvar that is in MR-6NF and we decompose *`mrv`* by assigning each participant to a relvar, then each resulting relvar that might be nonempty is in 6NF.

---

[13] Loosely speaking. When we say that *`k`* is a key of participant *`p`* in multirelvar *`mrv`* we really mean that *`k`* would be a *super*key of a virtual relvar defined on the MR-extraction of *`mrv`* corresponding to *`p`*. For if *`p`* is constrained at all times to be empty, then it has the empty set as key. In any case, even a relational `KEY` constraint does not and cannot fully enforce the irreducibility property that is defined in the theory for keys, and a similar observation applies to `MR_KEY`.

10. Normal forms for multirelvars

I propose two normal forms for multirelvars: MRK-NF and MR-6NF. Their definitions are simple. A multirelvar is in MRK-NF if and only if it is subject to an MR-key constraint. And a multirelvar is in MR-6NF if and only if it is subject to an MR-6NF constraint.

If *k* is a participant key for multirelvar *mrv1,* then its MR-6NF equivalent multirelvar *mrv2* can be obtained by the assignment

```
mrv2 := INTRADECOMPOSE mrv1 ON { k } ;
```

and its MRK-NF equivalent multirelvar *mrv3*, with MR-key *k*, can be obtained by

```
mrv3 := INTRAJOIN mrv1 ;
```

I suggest that in practice the only multirelvars that are not in one of these two normal forms will be those used for holding query results (i.e., snapshots). A multirelvar that is not in MRK-NF or MR-6NF (such as a single multirelvar constituting the entire database!) will give rise to all sorts of updating difficulties that I do not discuss in this paper. Rather, I discuss update operators that might usefully be defined on the assumption that one of these normal forms is in effect.

Under both normal forms it is impossible for two or more tuples to appear in the same multirelation such that, for some nonkey attribute, more than one value appears paired with the same key value. In other words, for every pair <*pr1,pr2*> of participants, every tuple of *pr1* has at most one matching tuple in *pr2*. Both normal forms prevent "accidents" such as

```
MULTIRELATION { TUPLE { S# S#('S1'),
                        SNAME NAME('Smith'),
                        CITY 'London' },
                TUPLE { S# S#('S1'),
                        CITY 'Paris' } }
```

in which two or more tuples appear to contradict each other. MRK-NF would prevent the appearance of two or more tuples for the same supplier. MR-6NF would prevent the appearance of more than one `CITY` value for the same supplier. The relevance of such constraints to the so-called missing information problem is apparent. We are dealing here solely with what the entity-relationship modellers call 1:1 relationships between entity types and the special cases of such relationships that can be denoted by entity subtypes. Those relationships are addressed in reference [1] by proposed constraint shorthands referred to as *distributed keys* and *foreign distributed keys.*

11. Multirelvar update operators

11.1 Assignment

… is defined as for variables of all types.

11.2 MR-insertion

Multirelvar insertion is very similar to relvar insertion. Example:

```
MR_INSERT S MULTIRELATION { TUPLE { S# S#('S2'),
                                    SNAME NAME('Jones'),
                                    CITY 'Paris' },
                            TUPLE { S# S#('S3'),
                                    SNAME NAME('Blake') },
                            TUPLE { S# S#('S3'),
                                    CITY 'Paris') } } ;
```

MR_INSERT *mrv mr* is equivalent to

```
mrv := mrv MR_UNION mr ;
```

However, if *mrv* is required to be in MRK-NF, then we would desire a shorthand for

```
mrv := INTRAJOIN ( mrv MR_UNION mr ) ;
```

Similarly, if *mrv* is required to be in MR-6NF, then we would desire a shorthand for

```
mrv := INTRADECOMPOSE ( mrv MR_UNION mr ) ;
```

As this is only a discussion paper, I do not at this time propose syntax for these desired shorthands.

We might also want to allow the input operand to be a relation.

11.3 MR-deletion

Multirelvar deletion is very similar to relvar deletion. Example:

```
MR_DELETE S MR_WHERE CITY = 'Paris' ;
```

loosely speaking, deletes every tuple having an attribute named CITY whose attribute value is 'Paris'. If S is in MRK-NF, this will—even more loosely speaking—delete every supplier located in Paris. If S is in MR-6NF, then every supplier located in Paris will become a supplier of no particular location.

MR_DELETE *mrv* MR_WHERE *cond* is equivalent to

```
mrv := mrv MR_UNLESS ( cond ) ;
```

Note the use of MR_UNLESS in this expansion. We must keep every tuple that lacks a CITY attribute.

Unless PRESENT and ABSENT are supported, some shorthand might be needed for deletion of entire participants. For example:

```
MR_DELETE S MR_WHERE ABSENT { SNAME } ;
```

is equivalent to

```
S := S MR_WITH ABSENT { SNAME } ;
```

No special varieties of deletion operator are needed for normal form preservation. Whichever normal form applies to *mrv* is always preserved.

11.4 MR-update

Multirelvar update is very similar to relvar update. Example:

```
MR_UPDATE S MR_WHERE S# = S#('S1') ( STATUS := 10 ) ;
```

The expansion is difficult but the effect is intuitively obvious, I hope. As with MR_DELETE, normal forms are guaranteed to be preserved.

12. Virtual relvars based on multirelvars

This subject needs further investigation, but one important observation can be made right away. Clearly, if {*a1, ..., an*} are the attributes of some subset of the MR-heading of multirelvar *mrv,* then a virtual relvar *prv* can be defined whose value is the participant, in the current value of *mrv,* whose heading consists of those attributes. The definition would look like this:

```
VAR prv VIRTUAL ( PARTICIPANT { a1, ..., an } ) FROM mrv;
```

Certain updates to *mrv* can now be expressed in terms of updates to *prv*. Such virtual relvars could provide a mapping from a database design based on multirelvars to a relational design based on the proposals of [1] or [10].

13. Interpretation of a multirelation

Note to reviewers: As mentioned in the *Note to reviewers* on page 2, this section is due for deletion or major revision.

Recall that we interpret a relation in the context of some predicate whose parameters (free variables) correspond to the attributes of the relation. A tuple is said to satisfy that predicate if substitution of its attribute values for the parameters of the predicate yields a true proposition. Each tuple having the same heading as the relation either satisfies or does not satisfy that predicate. The relation body consist of those tuples that satisfy the predicate.

We can interpret a multirelation in similar fashion only by resorting to second-order logic. The general predicate for an arbitrary multirelation *mr* is a dyadic one denoted by the English sentence "tuple *t* satisfies predicate *P*", where *P* stands for the predicate that gives the interpretation of some participant in *mr*. This general predicate is not very informative, of course. Its parameters, unlike those of a relation predicate, have no corresponding attributes. If IS_RELATION(*mr*) is *true* we can substitute the prime participant's intended predicate for *P*.

Note that *P* stands for *the only* predicate for a participant. Consider a multirelation with attributes S#, SNAME, STATUS, and CITY. One of its participants has all of those attributes. The predicate for that relation might be "Supplier *S#* is under contract, is named *SNAME*, has status *STATUS*, and is located in *CITY*." Another participant has just the attributes S#, SNAME, and CITY. Which of the following predicates might be appropriate?

(a) Supplier *S#* is under contract, is named *SNAME*, and is located in *CITY*.

(b) Supplier *S#* is under contract, is named *SNAME*, and is located in *CITY* and there does not exist a status *STATUS* such that supplier *S#* has status *STATUS*.

(c) There exists a status *STATUS* such that supplier *S#* is under contract, is named *SNAME*, has status *STATUS* and is located in *CITY* (but we do not know *S#*'s status).

I argue that of these three only (b) is appropriate.

If (a) is chosen and `TUPLE {S# S#('S1'), SNAME NAME('Smith'), STATUS 20, CITY 'London'}` appears in the multirelation, then `TUPLE {S# S#('S1'), SNAME NAME('Smith'), CITY 'London'}` must also appear. For if it is true that supplier S1 is named Smith, has status 20 and is located in London, then it is most definitely true that supplier S1 is named Smith and is located in London!

If (c) is chosen and `TUPLE {S# S#('S1'), SNAME NAME('Smith'), CITY 'London'}` appears in the multirelation, then some tuple consisting of those attribute values and an additional attribute value for STATUS must also appear! This observation raises a big question mark in my own mind on all attempts that have been proposed to represent the concept of something being "missing" because it "exists but its value is unknown". However, the issue brings to mind Hodges's notion ([8]) that it is more accurate in general to think of a proposition asserted to be true as representing a *belief*, rather than a fact.[14] Then we could rephrase (b) as follows:

(b) Supplier *S#* is named *SNAME* and is located in *CITY* and there does not exist a status *STATUS* such that we believe that supplier *S#* has status *STATUS*.

We might believe that supplier S1 has some status but it does not follow that there is some particular status *s* such that we believe S1's status to be *s*.

There is a nice consequence of this interpretation when we consider the predicate represented by a multiprojection. Take this example:

```
S ONTO { S#, SNAME, CITY }
```

Some tuples in the result might be derived from the participant whose predicate is "Supplier *S#* is named *SNAME*, is located in *CITY* and has status *STATUS*." The predicate for the given projection of that participant is therefore "There exists a status *STATUS* such that

[14] C.J. Date gives a more in-depth discussion of appealing to belief rather than fact in reference [4].

supplier *S#* is named *SNAME*, is located in *CITY* and has status *STATUS*." Other tuples in the result are derived from the participant whose predicate is "Supplier *S#* is named *SNAME* and is located in *CITY* and there does not exist a status *STATUS* such that we believe that supplier *S#* has status *STATUS*." A predicate for the given multiprojection is therefore the disjunction of these two:

> Either there exists a status *STATUS* such that supplier *S#* is named *SNAME*, is located in *CITY* and [we believe] has status *STATUS* or supplier *S#* is named *SNAME* and is located in *CITY* and there does not exist a status *STATUS* such that we believe that supplier *S#* has status *STATUS*.

which simplifies to

> Supplier *S#* is named *SNAME*, and is located in *CITY*.

14. Applications of multirelations

Since publication of *The Third Manifesto* in the mid-1990s we have seen a gratifying amount of interest—and it is still growing—in the idea of providing a *TTM*-conforming (i.e., truly relational) interface to existing SQL databases. The tables in Figure 1, depicting multirelations, could equally well be depicting SQL tables, with `NULL` assumed to be occupying each of the vacant spaces. The apparently straightforward mapping from SQL tables to multirelations creates an obvious opportunity to provide an alternative language for operating on these objects. The operators proposed here are, unlike SQL's, based firmly in classical logic and set theory. That should make them significantly easier to teach, learn, and use, as well as providing a bridge to true relations and *their* operators.

As we will see, the main applications I have in mind for multirelation operators are in database constraints that would be needed if the database is to contain multirelation variables. The usefulness of the multirelation operators for query purposes is somewhat open to question because their complexity, as discussed in Section **13, Interpretation of a multirelation**, makes them in general more subject to *misinterpretation* than relations. I note, however, that the often-perceived requirement for "outer" operations—especially outer join—can be addressed by use of MR-union. Also, for a certain common kind of regular report, MR-union might be more suitable than relational join. Requirements such as these have occasionally given rise to suggestions that "relational" operators might be needed that yield sets of relations, an idea that differs from the multirelation approach primarily in that a participating relation in multirelation *mr* is a subset of the body of *mr* rather than an element of it. Here, for example, is how David McGoveran floated the idea in reference [10] (the emphasis being his):

> [The foregoing discussion] suggests some extensions to the relational algebra to support more general versions of the relational operators. In particular, relational union is a restricted version of the general set union. I propose that the system should automatically create several tables in the output (when appropriate), grouping like rows together by performing by performing the "restrict and throw away nulls" operation in the user's behalf. […] In effect, such set operations would be many-table result

> versions of existing relational operations; they would avoid the need for users to simulate such operations manually, via several SQL statements. Whether many-table *operands* (as opposed to results) should be permitted deserves additional and careful consideration, however. **For the time being I propose that such many-table values be supported only for output.**[15]

For example, assume that `S, SP` and `SPJ` are the usual relvars for suppliers, shipments, and shipments to projects and we want a report showing suppliers in supplier number order, each one followed by a list showing shipments for that supplier, each one followed by a list of the projects using such shipments. The multirelation denoted by the following expression provides all the information needed for that report:

```
MR_UNION { S, SP, SPJ }
```

Its tuples would have to be fed to the report generator in a suitable order to meet McGoveran's "grouping like rows together" requirement. Perhaps that could be specified by

```
ORDER ( ASC S#, ASC P#, ASC J# )
```

The resulting order would need to be defined such that an `SP` tuple comes immediately before its first matching `SPJ` tuple (if any), and an `S` tuple comes immediately before its first matching `SP` tuple (if any). Of course, if for some reason the usual "foreign key" constraints have not been defined for `SPJ` and `SP`, then the report will display some anomalies.

15. Some topics for further investigation

1. Resolution of outstanding issues in connection with MR-restriction.
2. Multirelation comparison operators and constraints.
3. Virtual relvars defined on multirelvars, and virtual multirelvars.
4. Subtypes of multirelation types under specialization by constraint as described in [2].
5. Aggregation and summarization of multirelations.
6. Support for temporal data in multirelations.
7. A formally specified algebra of multirelations.
8. What problems, if any, can be solved with multirelations that cannot be solved without them? (We think, none.)

[15] I hope I have given the matter that "additional and careful consideration", and that 13 years is sufficient to satisfy McGoveran's "[f]or the time being".

16. Acknowledgements

Chris Date commented, more than once in detail, on several early drafts. I agreed with nearly all of those comments. In one of his comments that I agree with but have not addressed he remarks that my use of the term "normal form" for certain canonical forms is somewhat out of kilter with its use in relational database design theory, where it applies to a series of such forms, based on projection and join, such that each *n*-1th form in that series is implied by the *n*-th such form. I agree with the observation but am reluctant to abandon the snappy terms MRK-NF and MR-6NF. At least it is the case that MR-6NF is derived from MRK-NF by a form of decomposition using projections.

Adrian Hudnott, of Warwick University, also reviewed an earlier draft and gave me some useful comments.

Dennis Ashley provided references [5] and [7], both being mathematical treatises using the term *multirelation*. It is not clear to me whether they are referring to exactly the same concept, nor how close either of them is to the concept I have defined here.

17. References

[1] Hugh Darwen. *How to Handle Missing Information Without Using NULL* at http://www.thethirdmanifesto.com.

[2] C.J. Date and Hugh Darwen. *Databases, Types, and The Relational Model: The Third Manifesto* (3rd edition). Addison-Wesley (2006).

[3] C.J. Date. *The Relational Database Dictionary*. O'Reilly (2006).

[4] C.J. Date. Chapter 4, "The Closed World Assumption" in *Logic and Databases: The Roots of Relational Theory*. Trafford publishing (2007).

[5] Roland Fraïssé and Norbert Sauer. *Theory of Relations.* Elsevier (2000).

[6] Maurice Gittens. *On Logical Mistakes and The Third Manifesto.* An English translation of an article in Dutch that appeared in the Database Magazine, Issue #2, April 2007.

[7] Wim H. Hesselink. *Multirelations are predicate transformers* at http://www.cs.rug.nl/~wim/pub/whh318.pdf

[8] Wilfrid Hodges. *Logic*, Penguin Books Ltd (1978)

[9] Adrian Larner. *A New Model of Data* at http://www.btinternet.com/~adrian.larner/database.htm.

[10] David McGoveran. "Nothing from Nothing" (in four parts), in C.J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994-1997.* Addison-Wesley (1998).

[11] Fabian Pascal. *The Final Null in The Coffin* at http://www.dbdebunk.com/publications.html.

End of paper