Chapter 24

The Multirelational Approach

Do I contradict myself?
Very well then I contradict myself,
(I am large, I contain multitudes.)

—Walt Whitman: *Song of Myself*

If we had some ham, we could have ham and eggs,
if we had some eggs.

—Anon.

Quite a few of the various approaches that have been proposed at one time or another to the "missing information" problem involve data structures that look like relations but aren't—they look like relations in that their bodies consist of tuples, but they differ from relations in that tuples in the same body don't all have to have the same heading. The latest such proposal (and the one that prompted the investigations described in this chapter) is in reference [10], where once again the idea is floated that such structures be admitted to the database, subject to certain restrictions I don't need go into here. An earlier proposal can be found in reference [12], by my friend the late Adrian Larner. Larner did discuss his ideas with me at the time, but I failed to understand them properly then and I still fail to understand them fully now.

Fig. 1 overleaf depicts in tabular form a couple of examples of the kinds of structures all such approaches involve (table S represents suppliers and table SP shipments).[1] Please note, however, that the gaps or vacant spaces in the tables in that figure aren't meant to represent SQL-style nulls; nor are they meant to represent blanks or the empty character string. Rather, they're meant to represent *the complete absence* of the attribute in question from the tuple in question—where, of course, the attribute in question is 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 appear in the row in which the gap appears. Thus, to spell the point out (and to adopt an obvious shorthand notation for tuples), the tuples <S3,Blake,30> and <Smith,20,London>, although they both appear in whatever it is that table S is supposed to represent, have different headings and are therefore of different types. What's more, the heading of the first of these tuples in particular is different from the heading of whatever it is that table S is supposed to represent.

[1] If those tables represented relations and those relations were meant to be values of relvars, then I would normally mark certain attributes by double underlining to show they were components of the primary—or at any rate sole—key for the relvar in question. I omit such double underlining in this chapter (except when the table in question really does represent a relation), for reasons to be explained in the discussion of *MR-keys* in the section "Constraints," later in the chapter.

S

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			

SP

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

Fig. 1: Suppliers and shipments—sample values

At first glance, it would seem that such schemes mean we have to abandon relations, which is why in the past I've tended to reject them out of hand. But the publication of reference [10], an article criticizing some of my own (joint) work, more or less compelled me to produce a detailed justification for those rejections. To do that, I needed to work out in more detail what such a scheme might look like in practice; that exercise, I thought, would clearly expose the flaws in the idea. In thinking about this question, however, it occurred to me that the kind of structure under consideration, though not itself a relation, does in a sense *include* certain relations, and those relations might somehow be extracted from it. And if we did all our real work (as it were) in terms of those extracted relations, then it might be argued that we would have preserved the relational model, and we would have abided by Codd's *Information Principle* (which requires that all information in the database be cast in the form of relations, and nothing but relations).[2] But—and it is a big but—it must still be understood that the kind of object that includes those relations is, to repeat, not itself a relation. With that caveat in mind, I still wished to see where the idea might lead.

It's easy to spot the five nonempty relations included in table S in Fig. 1. One contains the tuples with S# values S1, S2, and S5; each of these three tuples has all four of the attributes S#, SNAME, STATUS, and CITY. The other four relations, which happen to contain just one tuple each, are as follows:

- One contains just the tuple for supplier S3, with attributes S#, SNAME, and STATUS.
- One contains just the tuple for supplier S4, with attributes S#, SNAME, and CITY.
- One contains just the tuple for supplier S6, with attributes S# and CITY.
- And one contains just the tuple for supplier S7, with S# as its sole attribute.

But how many empty relations does that table include in that same sense? And what exactly is that sense? Such questions, and many more, needed to be answered, I thought, before I could begin to think about the operators that might be defined for operating on such objects.

What to call these objects? Because I was expecting my investigations to show why the idea should be rejected, I first considered the somewhat derogatory term *pseudorelation*. As my investigations proceeded, however, I became less convinced that the idea should be rejected out of hand—though I hasten to add that I'm certainly not yet arguing in favor of the idea either—and so I decided to call them *multirelations* instead.

---

[2] As reference [5] points out, this principle might more appropriately be called *The Principle of Uniform Representation,* or even *The Principle of Uniformity of Representation*.

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 also because of my inability to think of any suitable alternative, I decided to stick with *multirelation* for the time being. I do think my use of that term is more appropriate than some others I've found; some people apparently even use it to mean an SQL table, on the grounds that the body of such a table is (in general) a *bag* or, as the SQL international standard has it, a "multiset." See also references [9] and [11], with which I might or might not be significantly at odds.

I've said I'm not currently advocating support for multirelations. In fact, I'd like to emphasize that I don't regard the problem that multirelations address as a previously unsolved one. To be specific, I still stand by the "decomposition" solution presented in reference [1], which is in full conformance with the prescriptions of *The Third Manifesto* [6]. A similar though not identical approach is described by David McGoveran in reference [13] (the two approaches are briefly compared in reference [4]). In addition, Fabian Pascal discusses McGoveran's approach in reference [14] and might possibly be moving toward developing it along lines similar to those of the present chapter, though putting the decomposition under the covers (so to speak). But he doesn't attempt to spell out the details in the way I do here.

Of course, many people have raised psychological objections to these decomposition approaches, though without proposing any alternatives (apart from abandoning relations altogether, as SQL does).[3] The most commonly expressed objections are that they give rise to an excessive number of relvars and an excessive number of associated constraints. I can sympathize with these objections, somewhat. In 1982, the very first customer of Business System 12 (a relational DBMS available at that time to users of IBM's timesharing Bureau Service) was an organization offering information to investors about various companies. Over 300 separate items of information were identified as being of possible interest to investors, but only a very few of those items were available for every company. Under both my proposal [1] and McGoveran's [13], at least 300 separate relvars would be needed, together with a huge number of foreign key and other constraints. So we crammed everything into a single relvar, using "impossible" values such as –9999999.99 to indicate missing information. As we soon learned, however, those "impossible" values gave rise to all sorts of traps for the unwary and led to all sorts of complications in attempts to avoid those traps.

So the crucial question is: Can multirelations solve the "missing information" problem while addressing such objections and avoiding such traps and complications? As a basis for discussing this question, I now present the results of my investigations, without at this time offering much in the way of my own opinion on them. In other words, the chapter should be regarded not as a definitive statement but, rather, as a kind of discussion paper. You're invited to consider the following questions:

1. Are there any fatal flaws in the scheme? (I won't attempt to define exactly what I mean by "fatal flaws" here, but lack of logical soundness would certainly qualify.)
2. Is the scheme too complicated?
3. Does it suffer from too much that is counterintuitive?
4. If the answer to any of the foregoing questions is *yes,* can the scheme be suitably revised? (Presumably not if the answer to the first one is *yes*.)

Finally, the following list of major section headings gives some idea of the scope of the chapter:

- Some relational terminology
- What's a multirelation?

---

[3] Many have raised performance objections also; however, these latter objections invariably assume the approach to implementation found in current SQL DBMSs, which tends to punish decomposition instead of encouraging it.

- Selector operators
- Comparison operators
- Algebraic operators
- Multirelation variables
- Constraints
- Normal forms
- Update operators
- Virtual relvars and multirelvars
- Interpretation
- Potential applications
- Some outstanding questions

SOME RELATIONAL TERMINOLOGY

In this section I define for reference purposes some terms from relational theory that I'll be relying on heavily in the pages to come. The definitions are based for the most part on ones to be found in reference [5].

- **Definition (common attribute):** An attribute that's common to two or more tuples or relations. To be specific, attribute <*A,T*> is (a) a common attribute of relations *r1, r2, ..., rn* ($n > 1$) if and only if each of *r1, r2, ..., rn* has an attribute named *A* of type *T,* and (b) a common attribute of tuples *t1, t2, ..., tn* ($n > 1$) if and only if each of *t1, t2, ..., tn* has an attribute named *A* of type *T*.

- **Definition (joinable):** Relations *r1, r2, ..., rn* ($n \geqslant 0$) are joinable if and only if the set theory union of the headings of those relations is a heading. Tuples *t1, t2, ..., tn* ($n \geqslant 0$) are joinable if and only if the set theory union of those tuples is a tuple. *Note:* The term *joinable* might better be *mutually joinable,* since as we've just seen it's always defined with respect to a given context (viz., a given set of relations or a given set of tuples), and I'll use this latter term occasionally in what follows. Note too that it follows from the definitions that attributes of the relations or tuples in question that have the same name (a) must be of the same type—i.e., must be common attributes—and (b) in the tuple case, must have the same value.

- **Definition (join):** 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* ($n \geqslant 0$) 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 distinct relations from the set *r1, r2, ..., rn* and replace them by their dyadic join, and repeat this process until the set consists of just one relation *r,* which is the overall result.

- **Definition (union):** 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* ($n \geqslant 0$), UNION {*r1,r2,...,rn*}, where *r1, r2, ..., rn* are all of the same type *T,* 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.

- **Definition (tuple join):** 1. *(Dyadic case)* Let tuples *t1* and *t2* be joinable. Then the tuple join of *t1* and *t2, t1* UNION *t2,* is the set theory union of *t1* and *t2*. 2. *(N-adic case)* Let tuples *t1, t2, ..., tn* ($n \geqslant 0$) be joinable. Then the tuple join UNION {*t1,t2,...,tn*} is defined as follows: If $n = 0$, the result is the empty

tuple; if $n = 1$, the result is *t1;* otherwise, choose any two distinct tuples from the set *t1, t2, ..., tn* and replace them by their dyadic tuple join, and repeat this process until the set consists of just one tuple *t,* which is the overall result. *Note:* Use of the keyword UNION rather than JOIN in the foregoing definition is not a mistake—tuple join could equally well be called tuple union; in fact, for reasons that aren't important here, **Tutorial D** in particular does use the keyword UNION rather than JOIN to denote the tuple join operation.

- **Definition (subtuple):** Tuple *t1* is a subtuple of tuple *t2* if and only if *t1* is a subset of *t2*. *Note:* Since every set is a subset of itself, every tuple is a subtuple of itself.

- **Definition (supertuple):** Tuple *t2* is a supertuple of tuple *t1* if and only if *t2* is a superset of *t1*. *Note:* Since every set is a superset of itself, every tuple is a supertuple of itself.

WHAT'S A MULTIRELATION?

Now I need to pin down precisely what a multirelation is. First of all, observe that I talked earlier in terms of multirelations *including* relations (as subsets), not *containing* them (as elements)—though in fact I think it would be possible to come down on either side of this debate, if debate it is.[4] Indeed, when I get to the multirelational operators later in this chapter, I sometimes give two distinct but equivalent definitions, one based on the inclusion perception and one on the containment perception. A trifle arbitrarily, however, I do tend to favor, if only for definiteness, the inclusion perception, where relations are subsets of the pertinent multirelation instead of being elements of it. *Note:* I'm speaking a little loosely here. It would be more correct to say those included relations have headings that are subsets of the "heading" of the multirelation and bodies that are subsets of the "body" of the multirelation—where "heading" and "body" are in quotes because, as we'll see in a few moments, those relational terms as such don't really apply to multirelations.

Given some multirelation, the included relations are the *participants* in that multirelation. Fig. 2 overleaf shows the same multirelations as Fig. 1, but with the rows in the tables rearranged slightly and with separator lines to show the participants more clearly (and I'll use this style throughout the remainder of this chapter to depict multirelations as opposed to relations). As the figure indicates, the body of a participant with heading *H* consists of those tuples in the pertinent multirelation that have heading *H*.[5] Note, therefore, that (a) no proper subset of a participant body is itself a participant body, and (b) every tuple of a multirelation is contained in exactly one participant body.

Here now are precise definitions:

1. Let *mr* be a multirelation; then *mr* has an *MR-heading* and an *MR-body*.[6] The MR-heading of *mr* (referred to below as *MRH*) is identical in appearance to a relational heading but doesn't mean quite the same thing as such a heading; hence my use of a different term. Similarly, the MR-body of *mr* (referred to below as *MRB*) looks a bit like a relational body—certainly it's a set of tuples—but it differs from such a body in that those tuples don't all have to have the same heading, and so again I use a different term.

---

[4] An analogy that comes to mind is the well known duality principle in quantum theory, according to which the very same phenomenon is interpreted sometimes in terms of particles, sometimes in terms of waves.

[5] Elsewhere—in reference [6] in particular—headings are denoted {*H*} instead of *H*. The simpler notation is more convenient for present purposes, however. An analogous remark applies to bodies also.

[6] I use the prefix "MR" ubiquitously in what follows. It can be pronounced either "emm are" or "mister" according to taste (and perhaps context).

S

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

SP

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

Fig. 2: The suppliers and shipments multirelations and their nonempty participants

2. The *degree* of *mr* is the number of attributes in *MRH;* the *cardinality* of *mr* is the number of tuples in *MRB;* and the *type* of *mr* is MULTIRELATION *MRH*. For example, the multirelation S depicted in Figs. 1 and 2 has degree 4 and cardinality 7 and—following the syntactic style of **Tutorial D**—is (let's agree) of the following type:

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

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

4. Relation *p* is a *participant* in *mr* if and only if (a) the heading *PH* of *p* is a subset of *MRH* and (b) the body *PB* of *p* consists of just those tuples of *MRB* that have heading *PH*. It follows that if the degree of *mr* is *n,* then the number of participants in *mr* is 2^n (some of which will be empty, in general; for example, multirelation S from Figs. 1 and 2 has 11 empty participants as well as five nonempty ones). In particular, if the degree of *mr* is 0, then *mr* has exactly one participant (because $2^0 = 1$), and that participant is either TABLE_DEE or TABLE_DUM.[7] Also, since each tuple in *MRB* appears in exactly one participant, the bodies of the nonempty participants form a *partitioning* of *MRB*.

5. Multirelation *mr* is *empty* if and only if *MRB* is empty—equivalently, if and only if every participant in *mr* is empty. Note that emptiness here refers to a lack of tuples, not participants, because (to repeat) a multirelation of degree *n* always has 2^n participants, even if some or all of those participants are themselves empty.

6. If at most one participant in *mr* is nonempty, then the tuples in *MRB* all have the same heading (namely, some specific subset of *MRH*), in which case *MRB* is in fact the body of a relation with heading that subset, and it's tempting to say that *mr is* a relation, to all intents and purposes. Of course, that temptation should be resisted; after all, there's certainly a logical difference between the two concepts. But such

[7] In fact, every multirelation has either TABLE_DEE or TABLE_DUM (and not both) as a participant. For example, the multirelation S depicted in Figs. 1 and 2 has TABLE_DUM as one of its 11 empty participants. If it had TABLE_DEE as a participant instead (a nonempty one, of course), then the tabular representation of that multirelation would have a row of all gaps—i.e., a row with vacant spaces in every attribute position.

multirelations might well have a special role to play if we were to try to define a mapping between relational theory and the theory whose development I sketch in the remainder of this chapter. I therefore define an operator HAMONEP ("has at most one nonempty participant"), as follows: HAMONEP(*mr*) returns TRUE if at most one participant in *mr* is nonempty and FALSE otherwise.

7. When HAMONEP(*mr*) is TRUE, one of the participants in *mr* is called the *prime participant*. That participant is the sole nonempty participant if such exists, otherwise it's the (empty) participant whose heading is *MRH*. For example, if multirelation S contained just the tuple shown for supplier S4 in Figs. 1 and 2, then HAMONEP(S) would be TRUE, and the prime participant in S would be the relation containing just that tuple. As it is, however (i.e., given multirelation S as actually depicted in those figures), HAMONEP(S) is FALSE, and there is no prime participant.

8. I define RELATION(*mr*) to return the prime participant in *mr* if HAMONEP(*mr*) is TRUE and to be undefined otherwise.[8] I also define MULTIRELATION(*r*), where *r* is a relation, to return the multirelation whose MR-heading is the heading of *r* and whose MR-body is the body of *r*. Note that HAMONEP(MULTIRELATION(*r*)) is necessarily TRUE, and *r* itself is the prime participant in MULTIRELATION(*r*) (even if *r* is empty). But note too that it's not the case that MULTIRELATION(RELATION(*mr*)), even if it's defined, is necessarily equal to *mr,* because the heading of the prime participant of *mr* might be a proper subset of the MR-heading of *mr*. By contrast, it is necessarily the case that RELATION(MULTIRELATION(*r*)) is equal to *r,* thanks in part to the definition of "prime participant" in the case where *mr* is empty.

9. 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. Because of this fact, we can use multirelation attribute names in ways very similar to those in which we use relational attribute names in the relational context, as we'll soon see. In particular, we can extend the definition of the term *common attribute* in the obvious way to apply to multirelations as well as relations and tuples.

SELECTOR OPERATORS

The Third Manifesto [6] requires every value to be denotable by some literal, where a literal of type *T* is an invocation of some *selector* operator for type *T* in which each argument to the invocation is itself denoted by a literal in turn. Clearly, just as the relation selector invocation denoting relation *r* needs to specify—either explicitly or implicitly—both the heading and the body of *r,* so the multirelation selector invocation denoting multirelation *mr* needs to specify both the MR-heading and the MR-body of *mr*. So again we can follow the style of **Tutorial D:**

```
MULTIRELATION [ <MR-heading> ] { <tuple exp commalist> }
```

For example:

```
MULTIRELATION { S# S# , SNAME NAME , STATUS INTEGER , CITY CHAR }
  { TUPLE { S# S#('S3') , SNAME NAME('Blake') , STATUS 30 } ,
  { TUPLE { S# S#('S4') , SNAME NAME('Clark') , CITY 'London' } }
```

---

[8] RELATION(*mr*) is used in this chapter for expository purposes only. It couldn't be part of a language like **Tutorial D** that supports static type checking, because—assuming *mr* to be specifiable in such a language by means of an arbitrary multirelation expression—the heading of the prime participant wouldn't be known at compile time (in general). If the user knows that heading, however, then he or she can achieve the effect of RELATION(*mr*) by means of the operator PARTICIPANT FROM (see the section "Algebraic Operators," later).

Note in this example that (a) an explicit MR-heading has been specified, and (b) the specified MR-body contains two tuples with different headings. If no explicit MR-heading is specified, a default MR-heading equal to the set theory union of the headings of the specified tuples is specified implicitly.[9] (By definition, those tuples must be such that the default MR-heading is well defined; that is, they must be such that attributes with the same name are of the same type.) In the foregoing example, therefore, the explicit MR-heading could have been omitted, thus:

```
MULTIRELATION
  { TUPLE { S# S#('S3') , SNAME NAME('Blake') , STATUS 30 } ,
  { TUPLE { S# S#('S4') , SNAME NAME('Clark') , CITY 'London' } }
```

In general, any superset of the default MR-heading would be valid as an explicit MR-heading in a multirelation selector invocation. More generally, in fact, we can observe the following:

> If *MRB* is the MR-body of a multirelation with MR-heading *MRH,* then for every MR-heading MRH^+ that is a superset of *MRH* there is a multirelation with 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 MR-body of some multirelation with heading MRH^-, where MRH^- is the set theory difference between *MRH* and {*A*} (in that order).

It follows from these observations that (to spell the point out) distinct nonempty multirelations can have the same MR-body. For example, the MR-body of multirelation S as shown in Figs. 1 and 2 is also a possible MR-body for every multirelation whose MR-heading is a proper superset of the MR-heading of that multirelation S. This state of affairs contrasts sharply with that found in connection with relations, where two nonempty relations can (and do) have the same body only if they're in fact the very same relation.[10] *Note:* You might find this last claim surprising, especially if you're used to SQL; you might be thinking that surely two SQL tables can have the same set of rows and yet be distinct. Well, yes, indeed they can—but that's just another reason why SQL tables aren't relations. To be specific, rows in SQL tables, unlike tuples in relations, don't carry their heading around with them. (In fact, it would be more correct to say that rows in SQL tables, unlike tuples in relations, don't really *have* a heading.) As a consequence, and just by way of illustration, if SQL tables T1 and T2 both have just one column—of type INTEGER, say—but that column is called C1 in T1 but C2 in T2, then their bodies could indeed be equal in SQL, but the bodies of their relational analogs wouldn't be.

Consider now the empty MR-heading {}. There are just two multirelations with that MR-heading (i.e., just two multirelations of type MULTIRELATION {}): one whose MR-body contains exactly one tuple (necessarily the empty tuple), and one whose MR-body contains no tuples at all. I call them MR_DEE and MR_DUM, respectively. In other words, MR_DEE is MULTIRELATION{}{TUPLE {}} and MR_DUM is MULTIRELATION{}{}.

It should be clear that MR_DEE and MR_DUM are multirelational counterparts of TABLE_DEE and TABLE_DUM, respectively. In fact:

- TABLE_DEE is the sole participant, and thus the prime participant, in MR_DEE; so HAMONEP(MR_DEE) is TRUE, and RELATION(MR_DEE) is TABLE_DEE.

---

[9] Except in the case where the specified MR-body is empty, when the desired MR-heading must be specified explicitly.

[10] To state the matter precisely: Let the MR-heading *MRH1* of multirelation *mr1* be a proper subset of the MR-heading *MRH2* of multirelation *mr2*, but let the corresponding MR-bodies *MRB1* and *MRB2* be the same. Then (a) for all tuples *t, t* appears in *MRB1* if and only if it appears in *MRB2*; (b) for all relations *r1*, if *r1* is a participant in *mr1*, then *r1* is a participant in *mr2*; but (c) there exist relations *r2* such that *r2* is a participant in *mr2* and not a participant in *mr1* (though all such relations *r2* are empty).

- Likewise, TABLE_DUM is the sole participant, and thus the prime participant, in MR_DUM; so HAMONEP(MR_DUM) is also TRUE, and RELATION(MR_DUM) is TABLE_DUM.

COMPARISON OPERATORS

Let *mr, mr1,* and *mr2* be multirelations with MR-headings *MRH, MRH1,* and *MRH2,* respectively, and MR-bodies *MRB, MRB1,* and *MRB2,* respectively. If every participant in *mr1* is such that its body is a subset of the body of some participant in *mr2,* then *MRB1* is a subset of *MRB2*. Clearly, then, just as in **Tutorial D** we can write "*r1* ⊆ *r2*" (loosely, "*r1* is a subset of *r2*" or "*r1* is included in *r2*"), where *r1* and *r2* are relations, we can also write "*mr1* ⊆ *mr2*" (loosely, "*mr1* is a subset of *mr2*" or "*mr1* is included in *mr2*").

Now, in the relational case, *r1* and *r2* are required to be relations of the same type, but there's no need to impose an analogous rule in the multirelational case (in fact, it would be counterproductive to do so). We might perhaps require *MRH1* and *MRH2* not to be disjoint, but the advantages of doing so seem too slight to warrant such a rule. (If they *are* disjoint, then *mr1* ⊆ *mr2* will be TRUE if and only if *mr1* is empty.)

In the relational case, again, it's easy to see that the comparison operator "⊆" is logically sufficient.[11] But it isn't sufficient for all of the comparisons we might imagine for multirelations. For example, additional operators would be required to support tests such as the following, if there's any need for them:

- Is every tuple in *MRB1* a subtuple of some tuple in *MRB2?*
- Is every tuple of *MRB1* a supertuple of some tuple in *MRB2?*
- Is no tuple of *MRB1* a subtuple of some tuple in *MRB2?*
- Is no tuple of *MRB1* a supertuple of some tuple in *MRB2?*

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

ALGEBRAIC OPERATORS

In this rather lengthy section I define a number of read-only—in fact, algebraic—operators on multirelations, with an eye to what might be needed for practical purposes in a language like **Tutorial D**. However, I do not attempt, here, to develop a formal specification for an algebra of multirelations. Where applicable I do note some interesting algebraic properties, such as commutativity, associativity, and the like, but I make no attempt to identify a minimal or otherwise agreeable set of primitive operators.

Some of the operators have obvious relational counterparts. Where a relational counterpart exists, a concrete syntax might well use the same name for both the relational operator and its multirelational counterpart. I choose not to indulge in such operator name overloading here, however, because I think distinct names are preferable for expository purposes.

Notation: I assume again throughout this section (and I'll continue to make this assumption throughout the rest of the chapter, where appropriate) that *mr, mr1,* and *mr2* are multirelations with MR-headings *MRH, MRH1,* and *MRH2,* respectively, and MR-bodies *MRB, MRB1,* and *MRB2,* respectively. Also, for the sake of examples I take the names S and SP to refer not (as previously) to multirelations but, rather, to *multirelation variables* or multirelvars;[12] thus, for example, the expression "S" is now a *multirelvar reference,* and it denotes the

---

[11] Though for ergonomic reasons **Tutorial D** does also support the operators "⊂" ("is properly included in"), "⊇" ("includes"), "⊃" ("properly includes"), "=" ("equals"), and "≠" ("not equals"). Clearly, we would expect to see multirelation analogs of all of these operators to be supported as well, if multirelations are supported.

[12] Multirelvars are discussed in more detail in a series of sections toward the end of the chapter.

multirelation that's the current value of the multirelvar with that name. As stated earlier (in the section "What's a Multirelation?"), the type of multirelvar S is:

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

And the type of multirelvar SP is:

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

I'll also assume when I show sample results that the current values of these multirelvars are as shown in Figs. 1 and 2 (barring explicit statements to the contrary, of course).

Now I begin my discussion of the operators. The ones described in the two subsections immediately following ("Participant Extraction" and "Multiprojection") each yield a relation as their output; all of the rest yield a multirelation.

Participant Extraction

The PARTICIPANT FROM operator extracts a given participant (specified by means of its heading) from a given multirelation. For example:

```
PARTICIPANT { S# , P# } FROM SP
```

Result (a relation):

S#	P#
S2	P1
S4	P4

- **Definition (participant extraction):** Let *r* = PARTICIPANT {*A1,A2,...,An*} FROM *mr*. Then the heading *RH* of *r* is the subset of *MRH* specified by *A1, A2, ..., An,* and the body *RB* of *r* consists of just those tuples of *MRB* that have heading *RH*. Equivalently, *r* is that participant in *mr* whose heading is the subset of *MRH* specified by *A1, A2, ..., An*.

Note: In **Tutorial D,** wherever a commalist of attribute names can appear, denoting attributes of some relation *r,* that commalist can be preceded by ALL BUT to denote the attributes of *r* other than those mentioned. The same rule can obviously be used here, thereby allowing the example above to be expressed thus:

```
PARTICIPANT { ALL BUT QTY } FROM SP
```

Similar remarks apply, where appropriate, to all of the operators discussed in this chapter.

As noted earlier, if HAMONEP(*mr*) is TRUE and the heading of the prime participant is known, then PARTICIPANT FROM can be used to extract the prime participant from *mr*.

Multiprojection

Multiprojection is just a generalization of relational projection, but for clarity I won't use the relational projection syntax of **Tutorial D;** instead, I'll use a keyword, ONTO (since we often speak of projecting *onto* a given set of attributes). In essence, multiprojection works by first picking out just those tuples that have a specified set of

attributes (as well as others, possibly), and then projecting those tuples onto those attributes.[13] For example:

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

Or equivalently:

```
S ONTO { ALL BUT STATUS }
```

Result (a relation):

S#	SNAME	CITY
S1	Smith	London
S2	Jones	Paris
S5	Adams	Athens
S4	Clark	London

- **Definition (multiprojection):** Let *r* = *mr* ONTO {*A1,A2,...,An*}. Then the heading *RH* of *r* is the subset of *MRH* specified by *A1, A2, ..., An,* and the body *RB* of *r* is such that tuple *t* appears in *RB* if and only if *t* has heading *RH* and is a subtuple of some tuple in *MRB*. Equivalently, *RB* is the union of the bodies of the projections onto {*A1,A2,...,An*} of just those participants in *mr* that have heading some superset of *RH*.

Note that if HAMONEP(*mr*) is TRUE, then *mr* ONTO {*A1,A2,...,An*} = (RELATION(*mr*)) {*A1,A2,...,An*} if each of *A1, A2, ..., An* is an attribute of RELATION(*mr*), or an empty relation otherwise.

Now I turn to operators that return multirelations. Most of the operators I describe are multirelational counterparts of some familiar relational operator. I have little to say regarding their possible usefulness; I choose them rather for their teachability, on the grounds that—if multirelations turn out to be useful at all—the operators in question are likely to be as useful in the context of multirelations as their relational counterparts are in the context of relations.

MR-projection

MR-projection (MR_ONTO) applied to multirelation *mr* yields a multirelation *ms* formed by discarding all attributes not specified for retention. For example:

```
S MR_ONTO { STATUS , CITY }
```

Or equivalently:

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

Result (a multirelation):

[13] I follow *The Third Manifesto* here [5] here in taking projection to be an operator that can be applied to individual tuples as well as to relations.

STATUS	CITY
20	London
10	Paris
30	Athens
.....	.....
30	
.....	.....
	London
	Rome
.....	.....

By definition, this result has four participants ($4 = 2^2$), all of which happen to be nonempty (though one of them is TABLE_DEE and contains nothing but the empty tuple). It also happens to have the same cardinality as its input, but that's because no two tuples in that input have (a) the same combination of values for attributes STATUS and CITY, or (b) the same value for STATUS and no CITY attribute, or (c) the same value for CITY and no STATUS attribute.

Note: It's perhaps a trifle unfortunate that the operators ONTO and MR_ONTO have such similar names (multiprojection and MR-projection, respectively); it might be desirable to find a better name for at least one of them. At any rate, it's important to be clear over the logical differences between the two:

- First, ONTO ignores tuples that lack one or more of the specified attributes; MR_ONTO doesn't.
- Second, ONTO returns a relation; MR_ONTO returns a multirelation.

Here to illustrate these differences is the result (a relation) returned if we replace MR_ONTO by ONTO in the foregoing example—i.e., the result returned by S ONTO {STATUS, CITY}:

STATUS	CITY
20	London
10	Paris
30	Athens

- **Definition (MR-projection):** Let *ms* = *mr* MR_ONTO {*A1,A2,...,An*}. Then the MR-heading *MSH* of *ms* is the subset of *MRH* specified by *A1, A2, ..., An,* and the MR-body *MSB* of *ms* is such that tuple *t* appears in *MSB* if and only if *t* has heading some subset of *MSH* and is a subtuple of some tuple in *MRB*.

Note that if HAMONEP(*mr*) is TRUE, then HAMONEP(*ms*) is also TRUE, and RELATION(*ms*) is then some relational projection of RELATION(*mr*). Note too that TABLE_DEE is a participant in *ms* whenever the heading of some nonempty participant in *mr* contains none of the attributes *A1, A2, ..., An*. (This latter point was illustrated in the case of S MR_ONTO {STATUS,CITY}, because the input multirelation included a nonempty participant—namely, that whose MR-body contained just the tuple for supplier S7—whose heading contained neither of the attributes STATUS and CITY.)

MR-join

MR-join is the multirelational counterpart of relational join; it joins every participant in one operand to every participant in the other, and then returns the set theory union of those joins. For example:

```
S MR_JOIN SP
```

Result (a multirelation):

S#	SNAME	STATUS	CITY	P#	QTY
S1	Smith	20	London	P1	300
S1	Smith	20	London	P2	200
S2	Jones	10	Paris	P1	
S3	Blake	30		P2	200
S4	Clark		London	P4	

By the way, you might be thinking that joining every participant to every participant, as MR-join does, has the potential to produce a very large result—a kind of cartesian product of participants, as it were. As the example illustrates, however, it's likely in practice that the majority of those individual joins will yield empty results.

Here now is the definition:

- **Definition (MR-join):** Let *ms* = *mr1* MR_JOIN *mr2*. The MR-headings *MRH1* and *MRH2* must be such that their set theory union is an MR-heading; equivalently, each participant in *mr1* must be joinable with each participant in *mr2*. Then *ms* is the multirelation whose MR-heading *MSH* is the set theory union of *MRH1* and *MRH2* and whose MR-body *MSB* is the set of all tuples *t* such that *t* is the set theory union of a tuple of *MRB1* and a tuple of *MRB2*.

It follows from this definition that (as previously indicated) *p1* JOIN *p2* is a participant in *ms* if and only if *p1* is a participant in *mr1* and *p2* is a participant in *mr2*.

Now, it so happens in the example above that every tuple in the operand multirelations has a value for the sole common attribute (S#, in that example). Here by contrast is an example in which such is not the case:

```
( SP MR_ONTO { S# , QTY } )
  MR_JOIN
( SP MR_ONTO { P# , QTY } )
```

Result (a multirelation):

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

Note that if HAMONEP(*mr1*) and HAMONEP(*mr2*) are both TRUE, then HAMONEP(*ms*) is also TRUE. And if *mr1* and *mr2* have prime participants *p1* and *p2,* respectively, then RELATION(*ms*) = *p1* JOIN *p2*.

Like relational join, MR-join is both commutative and associative. It also has an identity value: viz., MR_DEE. As a consequence, an *n*-adic version of the operator, MR_JOIN {*mr1,mr2,...,mrn*}, can also be defined, thus: If *n* = 0, the result is MR_DEE; if *n* = 1, the result is *mr1;* otherwise, choose any two distinct multirelations from the set *mr1, mr2, ..., mrn* and replace them by their dyadic MR-join, and repeat this process until the set consists of just one multirelation *ms,* which is the overall result.

Unlike relational join, however, MR-join is not idempotent, even though S MR_JOIN S does happen to return its input given the sample values from Figs. 1 and 2. By way of a counterexample, let *mr* be the following multirelation:

```
MULTIRELATION { X INTEGER , Y INTEGER } { TUPLE { X 1 } ,
                                          TUPLE { Y 1 } }
```

Then the MR-body of *mr* MR_JOIN *mr* contains the tuple

```
TUPLE { X 1 , Y 1 }
```

in addition to the two tuples in the MR-body of *mr*.

Note: I considered defining a second kind of multirelation join in which only those participants that actually had values for the common attributes participated (if you see what I mean). But this operation can easily be expressed using MR_WHERE—see later—and MR_JOIN. The following example, a revised version of one of those given above, illustrates the point (the semantics of the "PRESENT{QTY}" construct are explained in the subsection "MR-restriction" but are in any case intuitively obvious):

```
( ( SP MR_ONTO { S# , QTY } ) MR_WHERE PRESENT { QTY } )
  MR_JOIN
( ( SP MR_ONTO { P# , QTY } ) MR_WHERE PRESENT { QTY } )
```

Result (a multirelation):

S#	P#	QTY
S1	P1	300
S1	P2	200
S3	P2	200

(No separator lines are shown because HAMONEP happens to be TRUE for this result.)

MR-union

Relational union requires its operands to be of the same type, but there's no need to impose such a rule on its multirelational counterpart MR_UNION; again, in fact, it would be counterproductive to do so. In other words, MR_UNION is very close to the conventional unfettered union of set theory. (An analogous remark applies to MR_INTERSECT and MR_MINUS also, q.v.) For example:

```
S MR_UNION SP
```

Result (a multirelation):

S#	SNAME	STATUS	CITY	P#	QTY
S1	Smith	20	London		
S2	Jones	10	Paris		
S5	Adams	30	Athens		
S3	Blake	30			
S4	Clark		London		
S6			Rome		
S7					
S1				P1	300
S1				P2	200
S3				P2	200
S2				P1	
S4				P4	

- **Definition (MR-union):** Let *ms* = *mr1* MR_UNION *mr2*. The MR-headings *MRH1* and *MRH2* must be such that their set theory union is an MR-heading. Then *ms* is the multirelation whose MR-heading *MSH* is the set theory union of *MRH1* and *MRH2* and whose MR-body *MSB* is the set theory union of *MRB1* and *MRB2*—i.e., *MSB* is the set of all tuples *t* such that *t* appears in *MRB1* or *MRB2* or both.

It follows from this definition that *p1* UNION *p2* is a participant in *ms* if (but not only if) *p1* is a participant in *mr1, p2* is a participant in *mr2,* and *p1* and *p2* are of the same (relation) type. Also, if *p3* is a

participant in *mr1* such that no participant in *mr2* is of the same type as *p3,* then *p3* is a participant in *ms;* likewise, if *p4* is a participant in *mr2* such that no participant in *mr1* is of the same type as *p4,* then *p4* is a participant in *ms*. In fact, each participant in *ms* is at least one, and possibly more than one, of the following: a participant in *mr1,* a participant in *mr2,* or the relational union *p1* UNION *p2* of a participant *p1* in *mr1* and a participant *p2* in *mr2*.

Note that if HAMONEP(*mr1*) and HAMONEP(*mr2*) are both TRUE and have prime participants *p1* and *p2,* respectively, then HAMONEP(*ms*) is also TRUE and RELATION(*ms*) = *p1* UNION *p2*.

Like relational union, MR-union is both commutative and associative; it is also, unlike MR-join, idempotent. Also, unlike relational union, it has an identity value: viz., MR_DUM. As with MR-join, therefore, an *n*-adic version of the operator, MR_UNION {*mr1,mr2,...,mrn*}, can be defined, as follows: If *n* = 0, the result is MR_DUM; if *n* = 1, the result is *mr1;* otherwise, choose any two distinct multirelations from the set *mr1, mr2, ..., mrn* and replace them by their dyadic MR-union, and repeat this process until the set consists of just one multirelation *ms,* which is the overall result.[14]

MR-intersection

Relational intersection requires its operands to be of the same type, but MR-intersection does not. For example:

```
S MR_INTERSECT SP
```

Result (a multirelation):

S#

As you can see, this result is empty. But if multirelation SP were to include (say) an additional tuple with S# value S7 and no other attributes, then that tuple would be common to the two multirelation operands and would therefore appear in the result.

- **Definition (MR-intersection):** Let *ms* = *mr1* MR_INTERSECT *mr2*. The MR-headings *MRH1* and *MRH2* must be such that their set theory union is an MR-heading. Then *ms* is the multirelation whose MR-heading *MSH* is the set theory intersection of *MRH1* and *MRH2* and whose MR-body *MSB* is the set theory intersection of *MRB1* and *MRB2*—i.e., *MRB* is the set of all tuples *t* such that *t* appears in both *MRB1* and *MRB2*.

It follows from this definition that *p1* INTERSECT *p2* is a participant in *ms* if and only if *p1* is a participant in *mr1, p2* is a participant in *mr2,* and *p1* and *p2* are of the same (relation) type.

Note that if HAMONEP(*mr1*) and HAMONEP(*mr2*) are both TRUE, then HAMONEP(*ms*) is also TRUE. In fact, HAMONEP(*ms*) is TRUE in several other circumstances as well—e.g., when either *MRB1* and *MRB2* is empty, or more generally when *MRB1* and *MRB2* are disjoint (which they certainly are if one is empty).

Like relational intersection, MR-intersection is commutative, associative, and idempotent; however, whereas relational intersection is a special case of relational join, MR-intersection is not a special case of MR-join. As with MR-join, an *n*-adic version of the operator, MR_INTERSECT {*mr1,mr2,...,mrn*}, can be

---

[14] Of course, we can and do define an *n*-adic version of relational union, too. What we mean when we say relational union has no identity value is this: UNION (unlike MR_UNION) requires its operand relations all to be of the same type; as a consequence, there's no "universal" identity value for relational union in general. But there *is* an identity value for any given relation type—namely, the empty relation of that type.

defined, as follows: If $n = 1$, the result is *mr1;* if $n > 1$, choose any two distinct multirelations from the set *mr1, mr2, ..., mrn* and replace them by their dyadic MR-intersection, and repeat this process until the set consists of just one multirelation *ms,* which is the overall result. Note, however, that the case $n = 0$ is not allowed.[15]

MR-difference

Relational difference requires its operands to be of the same type, but MR-difference does not. For example:

```
S MR_MINUS SP
```

Result (a multirelation):

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

As you can see, this result is identically equal to the current value of multirelvar S. But if the current value of multirelvar SP were to include (say) an additional tuple with S# value S7 and no other attributes, then that tuple would be common to the two multirelation operands and would therefore not appear in the result.

- **Definition (MR-difference):** Let *ms* = *mr1* MR_MINUS *mr2*. The MR-headings *MRH1* and *MRH2* must be such that their set theory union is an MR-heading. Then *ms* is the multirelation whose MR-heading *MSH* is *MRH1* and whose MR-body *MSB* is the set theory difference between *MRB1* and *MRB2* (in that order)—i.e., *MSB* is the set of all tuples *t* such that *t* appears in *MRB1* and not in *MRB2*.

It follows from this definition that *p1* MINUS *p2* is a participant in *ms* if (but not only if) *p1* is a participant in *mr1, p2* is a participant in *mr2,* and *p1* and *p2* are of the same (relation) type. Also, if *p3* is a participant in *mr1* such that no participant in *mr2* is of the same type as *p3,* then *p3* is a participant in *ms*. In fact, each participant in *ms* is either or both of the following: a participant in *mr1,* or the relational difference *p1* MINUS *p2* between a participant *p1* in *mr1* and a participant *p2* in *mr2* (in that order).

Note that if HAMONEP(*mr1*) is TRUE, then HAMONEP(*ms*) is also TRUE. And if *mr1* has prime participant *p1,* then (a) if *mr2* has a participant *p2* of the same type as *p1,* then RELATION(*ms*) = *p1* MINUS *p2;* (b) otherwise, RELATION(*ms*) = *p1*.

Finally, if *mr1* and *mr2* are multirelations of the same type, then *mr1* MR_MINUS (*mr1* MR_MINUS

---

[15] Alternatively, we might introduce some syntactic mechanism in this case to specify the type of the result, in which case that result would be the universal multirelation of that type—i.e., the multirelation whose MR-body contains all possible tuples with MR-heading some subset of the MR-heading for that multirelation type. Even if we did, however, the implementation might very well want to outlaw (or at least flag) any expression requiring such a multirelation to be materialized.

mr2) is identically equal to *mr1* MR_INTERSECT *mr2*. However, it's easy to see that the same is not true, in general, if *mr1* and *mr2* are of different types.

MR-semijoin

Semijoin (MATCHING, in **Tutorial D**) takes two relational operands and returns those tuples from the first operand that are joinable with at least one tuple in the second, and the multirelational analog does essentially the same thing for multirelations. For example:

```
S MR_MATCHING SP
```

Result (a multirelation):

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	
S4	Clark		London

- **Definition (MR-semijoin):** Let *ms* = *mr1* MR_MATCHING *mr2*. The MR-headings *MRH1* and *MRH2* must be such that their set theory union is an MR-heading. Then *ms* is the multirelation whose MR-heading *MSH* is *MRH1* and whose MR-body *MSB* is the set of all tuples *t* such that *t* appears in *MRB1* and there exists a tuple in *MRB2* that is joinable with *t*.

It follows from this definition *mr1* MR_MATCHING *mr2* is identically equal to (*mr1* MR_JOIN *mr2*) MR_ONTO {*A1,A2,...,An*}, where *A1, A2, ..., An* are all of the attributes in *MRH1*.

Note that if HAMONEP(*mr1*) is TRUE, then HAMONEP(*ms*) is also TRUE. And if *mr1* has prime participant *p* and *mr2* has participants *p1, p2, ..., pn,* then RELATION(*ms*) = UNION {(*p* MATCHING *p1*),(*p* MATCHING *p2*),...,(*p* MATCHING *pn*)}.

Now, relational intersection is a special case of relational semijoin (i.e., *r1* MATCHING *r2* degenerates to *r1* INTERSECT *r2* if *r1* and *r2* are of the same type). But MR-intersection is not a special case of MR-semijoin (i.e., there exist multirelations *mr1* and *mr2* such that *mr1* MR_INTERSECT *mr2* and *mr1* MR_MATCHING *mr2* are both defined but yield different results, even if *mr1* and *mr2* are of the same type). The reason is that there might be a tuple *t* in *MRB1* that's not also in *MRB2* but is nevertheless joinable with some tuple in *MRB2,* in which case that tuple *t* appears in the result of *mr1* MR_MATCHING *mr2* but not in the result of *mr1* MR_INTERSECT *mr2*.

MR-semidifference

Semidifference (NOT MATCHING, in **Tutorial D**) takes two relational operands and returns those tuples from the first operand that aren't joinable with any tuples at all in the second, and the multirelational analog does essentially the same thing for multirelations. For example:

```
S NOT MR_MATCHING SP
```

Result (a multirelation):

S#	SNAME	STATUS	CITY
S5	Adams	30	Athens
S6			Rome
S7			

- **Definition (MR-semidifference):** Let *ms* = *mr1* NOT MR_MATCHING *mr2*. The MR-headings *MRH1* and *MRH2* must be such that their set theory union is an MR-heading. Then *ms* is the multirelation whose MR-heading *MSH* is *MRH1* and whose MR-body *MSB* is the set of all tuples *t* such that *t* appears in *MRB1* and there does not exist a tuple in *MRB2* that is joinable with *t*.

It follows from this definition that *mr1* NOT MR_MATCHING *mr2* is identically equal to *mr1* MR_MINUS (*mr1* MR_MATCHING *mr2*).

Note that if HAMONEP(*mr1*) is TRUE, then HAMONEP(*ms*) is also TRUE. And if *mr1* has prime participant *p* and *mr2* has participants *p1, p2, ..., pn,* then RELATION(*ms*) = *p* NOT MATCHING *p1* NOT MATCHING *p2* ... NOT MATCHING *pn* (where the NOT MATCHINGs are evaluated in sequence left to right).

Now, relational difference is a special case of relational semidifference (i.e., *r1* NOT MATCHING *r2* degenerates to *r1* MINUS *r2* if *r1* and *r2* are of the same type). But MR-difference is not a special case of MR-semidifference (i.e., there exist multirelations *mr1* and *mr2* such that *mr1* MR_MINUS *mr2* and *mr1* NOT MR_MATCHING *mr2* are both defined but yield different results, even if *mr1* and *mr2* are of the same type). The reason is that there might be a tuple *t* in *MRB1* that's not also in *MRB2* but is nevertheless joinable with some tuple in *MRB2,* in which case that tuple *t* appears in the result of *mr1* MR_MINUS *mr2* but not in the result of *mr1* NOT MR_MATCHING *mr2*.

MR-restriction

Restriction is one of the simplest of the conventional relational operators. In **Tutorial D,** it's expressed as follows:

```
r WHERE cond
```

Here (a) *r* is a relation, represented by a relational expression of arbitrary complexity, and (b) *cond* is a restriction condition on *r*—i.e., it's a boolean expression in which every attribute reference identifies some attribute of *r* and there aren't any relvar references.[16] Each tuple *t* in *r* provides a value for each referenced attribute, thus allowing *cond* to be evaluated for that tuple, and that tuple *t* then appears in the body of the result if and only if the result of that evaluation is TRUE.

By contrast with the foregoing, multirelational restriction is a rather more complicated affair. The reason is that if *mr* is a multirelation, then (in general) some tuples in the MR-body *MRB* of *mr* will lack some of the attributes in the MR-heading *MRH* of *mr*. As a consequence, a simple expression such as

```
S MR_WHERE STATUS > 10
```

[16] Like many other languages, **Tutorial D** actually allows WHERE clauses to contain boolean expressions of arbitrary complexity (in particular, it allows them to contain relvar references). But if the boolean expression *cond* isn't of the particular simple form under discussion, then technically speaking the expression *r* WHERE *cond* doesn't represent a relational restriction as such.

will, typically, fail; to be specific, it'll fail at run time if the evaluation process encounters a tuple of S that lacks a STATUS attribute. (Of course, syntax of the even simpler form S WHERE STATUS > 10 would be doubly incorrect, because "S" isn't a relational expression but a multirelational one.)

Now, we could try adopting a rule to the effect that such tuples are simply ignored, but it's easy to see that such a rule quickly leads to worse problems. I'll just point out one such problem here (though there are countless others): The expression

```
S MR_WHERE NOT ( STATUS > 10 )
```

will presumably, in accordance with the hypothetical rule, also have to overlook tuples without a STATUS attribute—with the consequence that such tuples "fall through the cracks," as it were. Indeed, despite the fact that multirelations are quite definitely based on classical two-valued logic, the suggested rule seems to lead directly to just the kinds of difficulties that SQL's three-valued logic gets us into! And if there were no way around *that* problem, then the entire multirelations exercise would become rather pointless.

The most straightforward approach to avoiding such problems—though I feel bound to observe immediately that although the approach might be straightforward, the consequences in practice might not be—is to provide a means of testing, within an "MR_WHERE clause," for the presence of a particular attribute within a particular tuple. For example:

```
S MR_WHERE CASE
              WHEN PRESENT { STATUS } THEN STATUS > 10
              ELSE FALSE
           END CASE
```

Result (a multirelation):

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S5	Adams	30	Athens
....	.....	.....	.....
S3	Blake	30	

Explanation: Imagine the expression following MR_WHERE being evaluated for each tuple of S in turn, in some arbitrary sequence. For such a tuple *t,* PRESENT {STATUS} will evaluate to TRUE if and only if that tuple *t* actually has a STATUS attribute; thus, the CASE expression overall will evaluate to TRUE for just those tuples that do have a STATUS attribute with value greater than 10.

Note, incidentally, that the following syntax does *not* work:

```
S MR_WHERE PRESENT { STATUS } AND STATUS > 10
```

The reason is, of course, that AND is commutative; at run time, therefore, the system might legitimately try to evaluate the expression STATUS > 10 before it determines whether the tuple in question actually has a STATUS attribute. Testing for STATUS > 10 on a tuple with no STATUS attribute will raise an error.[17]

[17] Whether the compiler could detect the possibility that some attributes might be absent from some tuples, and could thus perhaps avoid certain run time errors, is an open question. See the section "Constraints" later in this chapter.

- **Partial definition (PRESENT):**[18] Consider the MR-restriction *mr* MR_WHERE *cond*. The construct PRESENT {*A1,A2,...,An*} is allowed to appear within *cond* wherever a boolean expression is allowed to appear, and that construct evaluates to TRUE for tuple *t* in *MRB* if and only if, for all *i* ($1 \leqslant i \leqslant n$), attribute *Ai* is present in *t*.

Note, therefore, that the construct PRESENT {...} is expressly defined in terms of some specific tuple of some specific multirelation. As a consequence, it can't appear in all possible contexts in which boolean expressions in general can appear, but only in certain specific contexts: namely, those in which the pertinent multirelation and pertinent tuple are well defined. (It's precisely for such reasons that I refer to PRESENT {...} as a "construct" and not an expression.[19]) Such contexts include MR_WHERE clauses in MR-restrictions; MR-extractions (see the next subsection); MR_WHERE clauses in DELETE and UPDATE (see the section "Update Operators"; and possibly others, if the need arises.

Now, sometimes we wish to test not for the presence, but rather the absence, of some particular attribute with respect to some tuple. Of course, we might simply use a negated form of PRESENT—for example:

```
S MR_WHERE NOT ( PRESENT { STATUS } )
```

But it's natural to provide a more direct formulation:

```
S MR_WHERE ABSENT { STATUS }
```

Result (a multirelation):[20]

S#	SNAME	STATUS	CITY
S4	Clark		London
S6			Rome
S7			

There are some traps for the unwary here, however. I've implied, in effect (though I didn't say as much explicitly), that PRESENT {*A1,A2,...,An*} is equivalent to

```
PRESENT { A1 } AND PRESENT { A2 } AND ...
                AND PRESENT { An } AND TRUE
```

It follows that NOT (PRESENT {*A1,A2,...,An*}) is equivalent to

```
NOT ( PRESENT { A1 } ) OR NOT ( PRESENT { A2 } ) OR ...
                       OR NOT ( PRESENT { An } ) OR FALSE
```

So if we define ABSENT {*A1,A2,...,An*} to be equivalent to NOT (PRESENT {*A1,A2,...,An*}), we could run into

---

[18] The definition is partial because PRESENT (and ABSENT, q.v.) can be used in other contexts as well, as we'll see in a moment.

[19] Actually such constructs are expressions, but they're so called *open* expressions, and the unqualified term *expression* is usually taken to mean one that's closed. See Chapter 11 for further explanation.

[20] As you can see, the heading of this result includes an attribute (STATUS) for which no result tuple has a value. To eliminate that attribute—which we would surely want to do in practice—we can use MR_WITH (see the next subsection) in place of MR_WHERE.

some problems. Thus, it seems preferable to define ABSENT {*A1,A2,...,An*} to be equivalent to

```
ABSENT { A1 } AND ABSENT { A2 } AND ... AND ABSENT { An } AND TRUE
```

Now ABSENT {*A1,A2,...,An*} is equivalent to NOT (PRESENT {*A1,A2,...,An*}) only in the special case when $n = 1$. So we have:

- **Partial definition (ABSENT):** Consider the MR-restriction *mr* MR_WHERE *cond*. The construct ABSENT {*A1,A2,...,An*} is allowed to appear within *cond* wherever a boolean expression is allowed to appear, and that construct evaluates to TRUE for tuple *t* in *MRB* if and only if, for all i ($1 \leqslant i \leqslant n$), attribute *Ai* is absent from *t*.

Note in particular that, perhaps a little counterintuitively, PRESENT {} and ABSENT {} both evaluate to TRUE for every tuple of every multirelation. By way of explanation, consider first the case of PRESENT {}. This construct does *not* mean no attributes are present; rather, it means the set of no attributes is a subset of the set of attributes that *are* present. Likewise, ABSENT {} means the set of no attributes is a subset of the set of attributes that are absent. Since the empty set is a subset of every set, the result follows.

Here now are some more examples of MR-restriction (syntax only; I leave it to you to work out what the results look like, given our usual sample values):

- ```
  S MR_WHERE S# ≠ S#('S2') AND S# ≠ S#('S4') AND
             CASE
                WHEN PRESENT { STATUS , CITY }
                     THEN STATUS = 20 OR CITY ≠ 'London'
                ELSE FALSE
             END CASE
  ```

- ```
  S MR_WHERE CASE
                WHEN PRESENT { STATUS , CITY } THEN STATUS = 20
                WHEN PRESENT { CITY } THEN CITY = 'London'
                ELSE FALSE
             END CASE
  ```

- ```
  S MR_WHERE CASE
                WHEN PRESENT { CITY } THEN CITY = 'London'
                WHEN PRESENT { STATUS , CITY } THEN STATUS = 30
                ELSE FALSE
             END CASE
  ```

- ```
  S MR_WHERE CASE
                WHEN PRESENT { CITY }
                     THEN CASE
                             WHEN ABSENT { STATUS } THEN TRUE
                             ELSE FALSE
                          END CASE
                WHEN ABSENT { CITY }
                     THEN CASE
                             WHEN ABSENT { SNAME } THEN FALSE
                             ELSE TRUE
                          END CASE
             END CASE
  ```

With all of the foregoing by way of preamble, the following definition of MR-restriction might come as something of an anticlimax:

- **Definition (MR-restriction):** Let *ms* = *mr* MR_WHERE *cond*. Then *ms* is the multirelation whose MR-heading is *MRH* and whose MR-body consists of just those tuples in *MRB* for which *cond* is TRUE.

Note, however, that (to repeat) an error will occur if the evaluation of *cond* on some tuple involves an attempt to reference a nonexistent attribute (unless the reference appears in the context of PRESENT or ABSENT, of course). Note too that if HAMONEP(*mr*) is TRUE, then HAMONEP(*ms*) is also TRUE; and if *mr* has prime participant *p,* then RELATION(*ms*) is some relational restriction of *p*.

MR-extraction

Consider the following example:

```
( S MR_WHERE ABSENT { STATUS } AND PRESENT { CITY } )
                                        MR_ONTO { ALL BUT STATUS }
```

Result (a multirelation):

S#	SNAME	CITY
S4	Clark	London
....	.....	.....
S6		Rome

As you can see, what's happened here is that multirelation S has been restricted to just those tuples that don't contain a STATUS value but do contain a CITY value, and attribute STATUS has then been "projected away," as it were. Such a combination of operations seems likely to occur quite frequently, and so it seems worth considering a shorthand for it as suggested by the following example:

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

Recall now the participant extraction operator (PARTICIPANT FROM) discussed earlier in this section, which extracted a specified participant from a specified multirelation. The foregoing MR_WITH expression can also be regarded as performing an extraction operation of a kind, but it extracts not a specified participant but, rather, a specified multirelation. For that reason, I call it *MR-extraction*.

- **Definition (MR-extraction):** Let *ms* = *mr* MR_WITH ABSENT {*A1,A2,...,An*} PRESENT {*B1,B2,...,Bm*}. The sets {*A1,A2,...,An*} and {*B1,B2,...,Bm*} must be disjoint. Then *ms* is defined to be equal to the result of (*mr* MR_WHERE ABSENT {*A1,A2,...,An*} AND PRESENT {*B1,B2,...,Bm*}) MR_ONTO {ALL BUT *A1,A2,...,An*}. Equivalently, let *excl* and *incl* be the attributes specified by *A1, A2, ..., An* and *B1, B2, ..., Bm,* respectively; then the MR-heading *MSH* of *ms* is the set theory difference between *MRH* and *excl* (in that order), and the MR-body *MSB* of *ms* is such that tuple *t* appears in *MSB* if and only if *t* appears in *MRB* and has a heading that is both a superset of *incl* and a subset of *MSH*.

The ABSENT and PRESENT constructs can be specified in either order and need not both appear. Omitting PRESENT is equivalent to specifying PRESENT {}; omitting ABSENT is equivalent to specifying ABSENT {}. A couple of examples:

```
S MR_WITH PRESENT { S# , CITY }
```

Result (a multirelation):

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

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

Result (a multirelation):

S#	CITY
S6	Rome
....	.....
S7	

Note that if HAMONEP(*mr*) is TRUE, HAMONEP(*ms*) is also TRUE. HAMONEP(*ms*) is also TRUE whenever the set theory union of *incl* and *excl* is all of the attributes in *MRH* (in which case every tuple in the result certainly has the same heading: namely, the heading that consists of just the attributes of *incl*).

MR-extension

Relational extension is expressed in **Tutorial D** as follows:

```
EXTEND r : { X := exp }
```

Here (a) *r* is a relation, represented by a relational expression of arbitrary complexity; (b) *exp* is an expression in which attribute references identifying attributes of *r* are permitted; (c) *X* is the name of a new attribute;[21] and (d) tuple *t* appears in the result if and only if it's a tuple from *r* extended with a value for *X* that's computed by evaluating *exp* on *t*. Now, elsewhere this book proposes incorporating SUMMARIZE functionality into the relational EXTEND operator. I have not yet investigated the implications of that proposal for the multirelational analog of EXTEND; apart from such considerations, however, that analog is essentially straightforward. Here's an example:

```
MR_EXTEND ( S MR_WHERE PRESENT { STATUS } ) : { NT := STATUS + 10 }
```

Result (a multirelation):

[21] Usually, at any rate. I omit consideration of the case where it isn't for simplicity.

S#	SNAME	STATUS	CITY	NT
S1	Smith	20	London	30
S2	Jones	10	Paris	20
S5	Adams	30	Athens	40
S3	Blake	30		40

- **Definition (MR-extension):** Let *ms* = MR_EXTEND *mr* : {*X* := *exp*}. Then *ms* is the multirelation whose MR-heading is *MRH* extended with attribute *X* and whose MR-body consists of all tuples *t* such that *t* is a tuple of *MRB* extended with a value for attribute *X* that's computed by evaluating *exp* on that tuple of *r*. *Note:* In fact, this definition is essentially identical to the definition of relational extension, except that *mr* and *ms* are multirelations, not relations. Note too that if attribute *A* is referenced in *exp* but is absent from some tuple of *mr,* then attempting to evaluate *exp* on that tuple will give a run time error.[22]

Note that if HAMONEP(*mr*) is TRUE, then HAMONEP(*ms*) is also TRUE; and if *mr* has prime participant *p,* then RELATION(*ms*) is some relational extension of *p*.

As with relational extension, a multiple form of MR-extension can also be defined, but I omit the details here. An "outer" version might also be defined (see the subsection immediately following).

MR-renaming

At first sight, the multirelational analog of relational renaming appears quite straightforward. Here's an example:

```
S MR_RENAME { S# AS SNO }
```

Result (a multirelation):

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

However, one of the most vexing aspects of the whole multirelation idea is trying to decide when

[22] As noted earlier, whether the compiler could detect the possibility that some attributes might be absent from some tuples and thus avoid certain run time errors is an open question.

references to absent attributes (within specific tuples) should be treated as an error and when they should simply be ignored. In the case at hand (viz., MR_RENAME), it might seem harmless on the face of it just to ignore them. If we do, then, e.g., the following expression—

```
S MR_RENAME { STATUS AS XX }
```

—will yield:

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

In conventional relational algebra, however, the following identity holds:

```
R RENAME { A AS B }  ≡  ( EXTEND R : { B := A } ) { ALL BUT A }
```

So what about a multirelational counterpart of this identity? More specifically, what happens with the following expression, which ought perhaps to be equivalent to S MR_RENAME {STATUS AS XX}?

```
( MR_EXTEND S : { XX := STATUS } ) MR_ONTO { ALL BUT STATUS }
```

Answer: It fails as soon as a tuple is encountered in S without a STATUS attribute—see the definition of MR-extension in the previous subsection. So perhaps we should reconsider that definition, so that MR_EXTEND simply ignores tuples for which attributes mentioned on the right side of the assignment in braces are absent. If we do, then at least the multirelational counterpart of the foregoing identity will hold. But now consider the following example:

```
S MR_WHERE STATUS + 10 = i
```

(where *i* is some integer). Now, we've already agreed, for very good reasons, that this expression will fail as soon as a tuple is encountered in S without a STATUS attribute (see the subsection "MR-restriction," earlier). But this expression is, or at least ought to be, equivalent to the following one:

```
( MR_EXTEND S : { I := STATUS + 10 } ) MR_WHERE I = i
```

And this expression *doesn't* fail, if MR_EXTEND just ignores tuples as suggested above. So what should we do?

My own strong inclination, in situations like the one under consideration here, is to follow *The Principle of Cautious Design* [3] and to insist, even with MR_RENAME, that references to absent attributes be considered an error. If we take this path, then the example shown earlier—

```
S MR_RENAME { STATUS AS XX }
```

—will have to be rewritten thus:

```
( S MR_WHERE PRESENT { STATUS } ) MR_RENAME { STATUS AS XX }
  MR_UNION
( S MR_WHERE ABSENT { STATUS } )
```

I note in passing, however, that it would be possible to define a shorthand—we might perhaps call it *outer* MR_RENAME—for the foregoing combination of operations. Similarly, we might define "outer" shorthand versions of MR_EXTEND and MR_WHERE, if such operations were thought to be sufficiently useful.

Here then is my preferred definition for MR_RENAME:

- **Definition (MR-renaming):** The expression *mr* MR_RENAME {*A* AS *B*} is equivalent to

```
( MR_EXTEND mr : { B := A } ) MR_ONTO { ALL BUT A }
```

Note that if HAMONEP(*mr*) is TRUE, then HAMONEP(*ms*) is also TRUE; and if *mr* has prime participant *p,* then RELATION(*ms*) is some relational renaming of *p*.

As with relational renaming, a multiple form of MR-renaming can also be defined. I omit the details here.

Internal Join

The multirelation operators I've described in this section so far have all been counterparts of familiar relational operators. By contrast, the operators I describe in this subsection and the next don't have any relational counterpart. The operators in question both have to do with certain *canonical forms* for multirelations, canonical forms that might possibly be of use in connection with database design.

As a basis for illustrating the first of these operators, I'll use a different, and much simpler, value for multirelvar S (see Fig. 3 below). *Note:* Let me point out immediately that this multirelation would be prohibited as a value for S if certain obvious constraints were in effect for that multirelvar. See the section "Constraints," later.

S

S#	SNAME	STATUS	CITY
S1	Smith		London
S3	Blake	30	
S1		20	
S3		30	
S1			London

Fig. 3: Another suppliers multirelation

As you can see, the three tuples for supplier S1 in the multirelation in Fig. 3 are joinable, and so are the two tuples for supplier S3. The *internal join* operation (INTRAJOIN) essentially just performs the corresponding tuple joins; that is, the expression

```
INTRAJOIN S
```

produces the following result (a multirelation):

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S3	Blake	30	

- **Definition (internal join):** Let *ms* = INTRAJOIN (*mr*). Then *ms* is a multirelation with MR-heading *MSH* equal to *MRH* and MR-body *MSB* defined as follows: Let *t1, t2, ..., tn* be a set of tuples within *MRB* such that *t1, t2, ..., tn* are (a) mutually joinable and (b) not joinable with any other tuple in *MRB*. Then *t* is a tuple within *MSB* if and only if it's the join of the tuples in some such set *t1, t2, ..., tn*.

Informally, *mr* and *ms* can be regarded as "information equivalent" (i.e., they effectively both represent the same set of propositions). However, they differ inasmuch as *ms* has the following property: If *t1* and *t2* are distinct tuples of *ms,* then they're not joinable. By contrast, *mr* doesn't necessarily have this property (in fact, if it does, then *mr* = *ms*). As a consequence, *mr* might involve some redundancy (in the case at hand, the fact that supplier S1 is located in London appears twice, and so does the fact that supplier S3 has status 30), while *mr* doesn't exhibit redundancy of this same kind. In other words, one of the effects of INTRAJOIN is to eliminate certain redundancies.

Note that if HAMONEP(*mr*) is TRUE, then HAMONEP(*ms*) is also TRUE; and if *mr* has prime participant *p,* then RELATION(*ms*) is equal to *p*. Note too that TABLE_DEE is a participant—in fact, the sole participant—in *ms* if and only if it's the sole participant in *mr*.

Internal Decomposition

Loosely speaking, internal decomposition (INTRADECOMPOSE) is a kind of inverse of internal join.[23] Here's an example:

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

Suppose S denotes the result of the internal join example from the previous section. Then this expression produces the following result (a multirelation):

S#	SNAME	STATUS	CITY
S1			
S3			
S1	Smith		
S3	Blake		
S1		20	
S3		30	
S1			London

[23] In fact, we might reasonably consider calling it "internal projection," and using syntax such as INTRA_ONTO to denote it.

- **Definition (internal decomposition):** Let *ms* = INTRADECOMPOSE *mr* ON {*A1,A2,...,An*}. Then *ms* is a multirelation with MR-heading *MSH* equal to *MRH* and MR-body *MSB* defined as follows: Tuple *t* appears in *MSB* if and only if (a) it's a subtuple of some tuple in *MRB* and (b) its heading includes the attributes *A1, A2, ..., An* and at most one additional attribute.

Note that if *t* appears in *MSB* and has some attribute *B* over and above *A1, A2, ..., An,* then the projection of *t* onto {*A1,A2,...,An*} also appears in *MSB*. Note too that *ms* and *mr* are information equivalent if and only if every tuple of *mr* has all of the attributes *A1, A2, ..., An*. Note finally that HAMONEP(*ms*) is TRUE if and only if *mr* is empty or the only nonempty participant in *mr* is the one whose heading consists precisely of attributes *A1, A2, ..., An*.

Other Operators

Tutorial D supports a variety of other relational operators in addition to the ones mentioned earlier in this section (examples include GROUP, UNGROUP, SUMMARIZE, and TCLOSE). However, I deliberately haven't yet defined multirelational analogs of those operators—but all of those operators except TCLOSE are defined in terms of ones for which I *have* defined multirelational analogs, and so it seems reasonable to assume that multirelation analogs could be defined for those operators as well if desired. I also deliberately haven't yet considered possible multirelation analogs of GROUP and UNGROUP involving multirelation valued attributes in place of relation valued attributes.

MULTIRELATION VARIABLES

The syntax for defining a multirelation variable, or multirelvar, can obviously follow the **Tutorial D** pattern for defining a relvar. Here's an example:

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

The type of this multirelvar is MULTIRELATION {S# S#, SNAME NAME, STATUS INTEGER, CITY CHAR}, but the MR_KEY specification (see the section "Constraints" immediately following) further constrains the values that can be assigned to it, just as a relational KEY specification further constrains the values that can be assigned to a relvar. Note, however, that (as we'll see) whereas every relvar has at least one key, not every multirelvar has an MR-key.

Note: If *MR* is a multirelvar, then its value at any given time (which is a multirelation, of course) has a set of participant relations, one for each subset of the MR-heading of *MR*. It's convenient to extend the "participant" terminology to multirelvars too in the obvious way (just as, in the relational context, we use terminology such as "the body of relvar *R,*" by which we really mean the body of the relation that's the current value of *R*).

CONSTRAINTS

Multirelvar constraints of arbitrary complexity can be formulated as boolean expressions with multirelational operands. As in the relational case, however, shorthands to address certain common requirements will probably be desirable in practice. I consider a few possibilities in this connection.

MR_IS_EMPTY

MR_IS_EMPTY(*mr*) is TRUE if and only if *MRB* is empty—in other words, if and only if, for every participant *p* in *mr,* IS_EMPTY(*p*) is TRUE (in which case *mr* has an empty prime participant). In other words, the expression

```
MR_IS_EMPTY ( mr )
```

is shorthand for the following expression:

```
mr MR_ONTO { } = MR_DUM
```

One particularly common application of MR_IS_EMPTY is likely to be in connection with *required attributes*—where an attribute is "required" if and only if it's required to be present in every tuple of the pertinent multirelvar. For example:

```
CONSTRAINT MRC1 MR_IS_EMPTY ( S MR_WHERE ABSENT { S# } ) ;
```

This constraint requires multirelvar S to have as its value at all times a multirelation in which attribute S# is present in every tuple. It might be nice to introduce a further shorthand that expresses the same constraint as part of the definition of the multirelvar in question, perhaps like this:

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

However, this further shorthand is clearly not adequate in itself to express all possible constraints of this same general nature, and additional shorthands based on it might well be desired. For example, suppose certain pairs of attributes are mutually exclusive; suppose, for example, that multirelvar S has an additional attribute, REASON, which has a value if and only the STATUS value is absent and indicates the reason for that absence:

```
CONSTRAINT MRC2 MR_IS_EMPTY
   ( S MR_WHERE PRESENT { STATUS , REASON }
              OR ABSENT { STATUS , REASON } ) ;
```

Observe in particular that MR_WITH isn't particularly helpful with examples like this one—a fact that might raise questions about the usefulness of that operator in general.

By way of a third example, consider the case of a soccer club. It has a fixture list. For each match in the fixture list, the result is eventually entered. So we might imagine a multirelvar, FIXTURE, whose nonempty participants at all times number no more than three: one for those matches that need to be scheduled but have no date assigned yet, one for those that are scheduled but haven't yet been played, and one for those that have been played.[24] Let FIXTURE have attributes GOALS_FOR and GOALS_AGAINST, with the obvious meanings. Clearly, whenever one of these attributes has a value for a particular match, then so must the other:

```
CONSTRAINT MRC3 MR_IS_EMPTY
( FIXTURE MR_WHERE NOT ( PRESENT { GOALS_FOR , GOALS_AGAINST } )
                  AND NOT ( ABSENT { GOALS_FOR , GOALS_AGAINST } ) ) ;
```

Clearly, such constraints will get increasingly complex as the number of attributes involved increases—suggesting, again, that further shorthands might be desirable.

MR-keys

I strongly suspect that if multirelvars are to be used at all, then they should always have at least one *MR-key* as a matter of good practice—meaning, to spell the point out, that no two distinct tuples in the same MR-body have the same value for the MR-key in question. (By *multirelvars* here, I mean multirelvars in the database, not necessarily ones that might exist from time to time merely to hold the result of some query.) Here's the

---

[24] Do you see any particular advantages here over a conventional three-relvar design?

definition:

- **Definition (MR-key):** Specifying MR_KEY {*A1,A2,...,An*} for multirelvar *MR* defines an MR-key for *MR*. It's equivalent to imposing the following constraint on *MR:*

```
MR_IS_EMPTY ( MR MR_WHERE NOT ( PRESENT { A1 , A2 , ... , An } ) )
AND COUNT ( MR ONTO { A1 , A2 , ... , An } ) = MR_COUNT ( MR )
```

Note that this definition implies that attributes *A1, A2, ..., An* are all required ones. Also, it's worth pointing out explicitly that the expression *MR* ONTO {*A1,A2,...,An*}—the argument expression in the COUNT invocation on the left side of the equality comparison—returns a relation, not a multirelation. However, the argument expression to the MR_COUNT invocation on the right side is a multirelvar reference (MR_COUNT is, of course, the multirelational analog of COUNT).

Now, I've already indicated that not every multirelvar has an MR-key. By way of example, consider the following multirelation (repeated from the discussion of internal decomposition in the previous section):

S#	SNAME	STATUS	CITY
S1			
S3			
....	.....	.....	.....
S1	Smith		
S3	Blake		
....	.....	.....	.....
S1		20	
S3		30	
....	.....	.....	.....
S1			London

Clearly, this multirelation could be the current value of some relvar; equally clearly, it doesn't satisfy any MR-key constraint at all.

In general, an MR-key has the same properties of uniqueness and irreducibility as relational keys do. However, note that "uniqueness" here means "uniqueness across several relations": namely, all of the relations that are participants in the multirelation currently assigned to the multirelvar with the MR-key in question. In the case of multirelvar S, for example, with its MR-key {S#}, no two tuples are ever allowed to have the same S# value, even if the tuples in question would have appeared in distinct participants.

Participant Keys

It seems natural to introduce another kind of key, one that's unique within participants but not necessarily across them, as it were. Here's the definition:

- **Definition (participant key):** Specifying PARTICIPANT KEY {*A1,A2,...,An*} for multirelvar *MR* defines a participant key for *MR*. Such a specification implies that attributes *A1, A2, ..., An* are all required. If *PK* is such a key, then every nonempty participant *p* in *MR* at any given time is such that (a) *PK* is a subset of the heading of *p* and (b) no two distinct tuples in *p* have the same value for *PK*.

Note that if *MK* is an MR-key for *MR,* then *MK* is a participant key for *MR* a fortiori. However, if *PK* is a participant key for *MR,* then *PK* isn't necessarily an MR-key for *MR*.

Foreign MR-keys

The relational concept of foreign keys and, more generally, the relational *inclusion dependency* concept (see Chapter 13 of the present book) will need multirelational counterparts. For example, in the suppliers-and-shipments database (with sample values as shown in Figs. 1 and 2), multirelvars S and SP are subject to the constraint that, at all times, the MR-body of the MR-projection of SP onto {S#} is a subset of the MR-body of the MR-projection of S onto {S#}:

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

Clearly, we might want to define a shorthand according to which a specification of the form

```
FOREIGN MR_KEY { S# } REFERENCES S
```

could appear as part of the definition of multirelvar SP.

MR-DNF

An MR-DNF constraint (DNF = "decomposed normal form"—see the section immediately following) is a constraint on a multirelvar to the effect that the only participants allowed to be nonempty are those whose headings have no more than one attribute in addition to those of a participant key (the same participant key in every such participant).

NORMAL FORMS

As I mentioned a few pages back, there are two obvious canonical forms, or normal forms, that can be defined for multirelvars. I'll call them MR-JNF and MR-DNF, for joined normal form and decomposed normal form, respectively. They aren't mutually exclusive, by the way (i.e., a given multirelvar can be in both at the same time). Their definitions are simple:

- **Definition (MR-JNF):** Multirelvar *MR* is in MR-JNF if and only if it's subject to an MR-key constraint.
- **Definition (MR-DNF):** Multirelvar *MR* is in MR-DNF if and only if it's subject to an MR-DNF constraint.

If multirelvar *MR1* has a participant key *PK,* then its MR-JNF equivalent *MR2,* with MR-key *PK,* can be obtained by the following assignment:

```
MR2 := INTRAJOIN MR1 ;
```

Likewise, its MR-DNF equivalent *MR3* can be obtained by the following assignment:

```
MR3 := INTRADECOMPOSE MR1 ON { PK } ;
```

Both normal forms prevent "accidents" such as the following (an impossible value for multirelvar S, because the two tuples effectively contradict one another):[25]

[25] The point is, stating the constraints to prevent such accidents in the case of a multirelvar not in one of the two normal forms is a rather tricky business.

S#	SNAME	STATUS	CITY
S1	Smith		London
....	.....	.....	.....
S1			Paris

With reference to this particular example, enforcing MR-JNF for S prevents the appearance of two or more tuples for the same supplier; enforcing MR-DNF prevents the appearance of two or more CITY values for the same supplier. More generally, the two normal forms both prevent two or more tuples from appearing in the same multirelvar at the same time with the same value for the same MR-key value and with different values for some other attribute. Equivalently, for every pair of participants (*p1* and *p2,* say) in the multirelvar in question, no tuple in *p1* has the same value for the same MR-key value as more than one tuple—or, in the case of MR-JNF, any tuple—in *p2* at any given time.

Note: As you might have realized, there are some interesting parallels to be observed between multirelvars in either of the two normal forms, on the one hand, and relvars obtained by means of the decomposition approach [1] on the other. To be specific, the various constraints (on keys, to be precise) that relvars obtained by means of decomposition are required to satisfy are very similar to the constraints on MR-keys and participant keys that apply when either MR-JNF or MR-DNF is in effect.

Now, it's easy to see that a multirelvar that's in neither MR-JNF nor MR-DNF (such as a single multirelvar constituting the entire database!) would be subject to all sorts of difficulties in connection with update operations. In practice, therefore, I think the only multirelvars (base or virtual) that might not be in one of those two normal forms would be ones used for holding query results. For that reason, at this time I discuss, in the section immediately following, only such update operators as might usefully be defined under the assumption that one of those normal forms is in effect.

UPDATE OPERATORS

Assignment is defined for variables of all types, and multirelvars are no exception; I've already given a couple of examples.[26] As usual, however, certain shorthands are likely to be desirable in practice. The shorthands in question are very similar to the familiar (relational) INSERT, DELETE, and UPDATE shorthands of **Tutorial D**.

MR-INSERT

- **Definition (MR_INSERT):** Let *MR* be a multirelvar and let *mr* be a multirelation with MR-heading some subset of that of *MR*. Then MR_INSERT *MR mr* is equivalent to

  ```
  MR := MR MR_UNION mr ;
  ```

For example:

[26] Note, however, that whereas relational assignment requires the source relation and target relvar to have the same heading (in other words, to be of the same type), multirelational assignment requires only that the source multirelation have a heading that's some subset of that of the target multirelvar.

```
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'       } } ;
```

Result (assuming for simplicity that multirelvar S was empty before the MR_INSERT):

S#	SNAME	STATUS	CITY
S2	Jones		Paris
S3	Blake		
S3			Paris

This result violates both MR-JNF and MR-DNF. In general, therefore, if the target multirelvar *MR* is required to be in MR-JNF, we will probably want a shorthand for the following:

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

Similarly, if *MR* is required to be in MR-DNF, then we will probably want a shorthand for the following:

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

However, I offer no suggestions for such shorthands at this time.

Other possibilities that might be worth considering include:

- Allowing the *mr* operand to be a relation instead of a multirelation
- Supporting a multirelational analog of **Tutorial D**'s D_INSERT (in which case we'd presumably need a multirelational analog of **Tutorial D**'s D_UNION as well)

MR-DELETE

Note: For completeness, analogs of the first (less commonly used) form of **Tutorial D**'s DELETE, as well as its I_DELETE, might be desirable in addition to the following, but I omit consideration of those possibilities here.

- **Definition (MR_DELETE):** Let *MR* be a multirelvar and let *cond* be a boolean expression. Then MR_DELETE *MR* MR_WHERE *cond* is equivalent to

```
MR := MR MR_WHERE NOT ( cond ) ;
```

For example:

```
MR_DELETE S MR_WHERE CASE
                        WHEN PRESENT { CITY } THEN CITY = 'Paris'
                        ELSE FALSE
                    END CASE ;
```

Note the need to be able to use the PRESENT and ABSENT constructs once again. In the example, the effect is

to delete every tuple with an attribute named CITY whose value is Paris (loosely speaking). Tuples without a CITY attribute are *not* deleted.

Here's another example:

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

This MR_DELETE is equivalent to:

```
S := S MR_WHERE NOT ( ABSENT { SNAME , STATUS } ) ;
```

Observe in particular that it's *not* equivalent to:

```
S := S MR_WHERE PRESENT { SNAME , STATUS } ;
```

No special varieties of MR_DELETE are needed for normal form preservation.

MR-UPDATE

I omit a precise definition of MR_UPDATE; the details are tedious, though essentially straightforward, and the effect is (I hope) intuitively obvious.[27] Here's an example:

```
MR_UPDATE ( S MR_WHERE S# = S#('S1') AND PRESENT { STATUS } ) :
                                                { STATUS := 10 } ;
```

No special varieties of MR_UPDATE are needed for normal form preservation.

VIRTUAL RELVARS AND MULTIRELVARS

This subject needs further investigation, but one observation can be made right away. Clearly, if *A1, A2, ..., An* are attributes of multirelvar *MR,* then a virtual relvar—not multirelvar—*PV* can be defined over the participant in *MR* whose heading is just those attributes:

```
VAR PV VIRTUAL ( PARTICIPANT { A1 , A2 , ... , An } FROM MR ) ;
```

Certain updates on *MR* can now be expressed in terms of updates on *PV*. As I more or less suggested earlier (in the section "What's a Multirelation?"), such virtual relvars might thus provide the basis for a mapping from a database design based on multirelvars to a relational design based on the proposals of either reference [1] or reference [13].

INTERPRETATION

Consider the multirelation MSC shown in Fig. 4, which is intended to be a sample value for a multirelvar MSCV of the following multirelation type:

```
MULTIRELATION { S# S# , CITY CHAR }
```

Since MSCV is of degree two, every multirelation (including the one in Fig. 4 in particular) that's a possible value of that multirelvar necessarily has four participants. For the sake of the example—but completely arbitrarily—I've chosen to make each of the participants in the multirelation in the figure nonempty. (One of them is TABLE_DEE, represented in the figure by a row with a vacant space in every attribute position.)

Now let *s#* and *c* be range variables, ranging over the set S# (the underlying type for attribute S#) and the

---

[27] Though we might need an additional form of MR_UPDATE for adding attributes to, and/or removing attributes from, specified tuples (please forgive the loose manner of speaking here). Again I omit the details.

set CHAR (the underlying type for attribute CITY), respectively. I'll consider the four participants one at a time. (You can interpret the discussion that follows in terms of the multirelation in Fig. 4, but of course it's meant to apply to every multirelation that's a possible value for multirelvar MSCV.)

MSC

S#	CITY
S1	London
S2	Paris
....	....
S3	
S4	
....	....
	Athens
....	....

Fig. 4: Multirelation MSC

First of all, the body of the participant with heading {S#,CITY} consists of just those tuples of the form <*s#,c*> that satisfy predicate *p1*(*s#,c*), where *p1* is some predicate with parameters *s#* and *c*. (Intuitively, we might expect predicate *p1* to be *Supplier s# is located in city c,* but its exact form is irrelevant here.) So we can say the participant with heading {S#,CITY} is defined by the following domain calculus expression:[28]

```
<s#,c> : p1(s#,c)
```

Similarly, the defining expression for the participant with heading {S#} is

```
<s#>   : p2(s#)
```

for some predicate *p2* with sole parameter *s#*. Likewise, the defining expression for the participant with heading {CITY} is

```
<c>    : p3(c)
```

for some predicate *p3* with sole parameter *c*. And the defining expression for the participant with heading {} is

```
<>     : p4()
```

for some predicate *p4* with no parameters at all (in other words, *p4* is in fact a proposition).

So what's the predicate for multirelvar MSCV overall? Well, let MSCH and MSCB be the MR-heading and current MR-body, respectively, of that multirelvar, and let *t* be a tuple in MSCB. Clearly, *t* has a heading that's some subset of MSCH. Let *H1, H2, H3,* and *H4,* denote the four subsets of MSCH, thus:

```
H1 = { S# , CITY }

H2 = { S# }

H3 = { CITY }
```

[28] The domain calculus is slightly better suited to my purpose here than the possibly more familiar tuple calculus.

```
H4 = { }
```

Then we can say that the predicate for MSCV overall—let's call it MSCP—looks something like this:

```
IF heading(t) = H1 THEN p1(t) AND
IF heading(t) = H2 THEN p2(t) AND
IF heading(t) = H3 THEN p3(t) AND
IF heading(t) = H4 THEN p4(t)
```

Observe that:

a. This predicate is a conjunction of implications.

b. For any given tuple *t* with heading *Hi* for some *i* ($1 \leqslant i \leqslant 4$), exactly one of those implications has an antecedent ("the IF part") that evaluates to TRUE.

c. Each of the other three implications has an antecedent that evaluates to FALSE and thus evaluate to TRUE overall.

d. For that tuple *t,* in other words, the predicate reduces to just *pi*(*t*).

Note: We can simplify the formulation of this predicate slightly by introducing the notation *PS*(MSCH) to denote the power set of MSCH:

```
FORALL Hi ∈ PS(MSCH) ( IF heading(t) = Hi THEN pi(t) )
```

A final observation: Consider a conventional relational database *RDB* containing just four relvars *RV1, RV2, RV3,* and *RV4,* with relvar predicates *p1, p2, p3,* and *p4,* respectively. Then what we might call "the database predicate" for database *RDB* will again be essentially just predicate MSCP—which is surely just as we should expect.

POTENTIAL APPLICATIONS

Since the first publication of *The Third Manifesto* in early 1995 [2], we've seen a gratifying amount of interest—and it's still growing—in the idea of providing an interface to existing SQL databases that conforms to the proposals of the *Manifesto* (i.e., is truly relational). Now, the tables shown throughout this chapter, depicting multirelations, could alternatively be understood as depicting SQL tables, with nulls occupying the vacant spaces. It seems, therefore, that there's a straightforward mapping between SQL tables and multirelations. The existence of such a mapping creates an obvious opportunity to provide an alternative language for operating on SQL tables. The operators described in the present chapter are, unlike SQL's operators, based firmly on classical logic and set theory. As a result, they should be significantly easier than SQL's operators to teach, learn, and use, and they should also be capable of serving as a bridge to true relations and *their* operators.

In my opinion, the main application for multirelation operators is likely to be in connection with constraints that would be needed if the database were to contain multirelvars. The usefulness of those operators for query purposes is, I think, somewhat debatable; to be specific, I think their comparative complexity makes them more subject to misinterpretation than their relational counterparts (even though, to repeat, I do think they're simpler than their SQL counterparts).

That said, however, I note that the often perceived requirement for "relational" *outer* operations (especially outer join) can be addressed by means of MR-union in particular. In particular, for a certain kind of report—one commonly required in practice—MR-union might be more suitable than relational join. Such requirements have occasionally given rise to suggestions that "relational" operators might be needed that yield sets of relations instead of just a single relation as their result. Speaking a trifle loosely, the key difference between such suggestions and the multirelation approach seems to be this (though I readily admit the difference in question

might be more apparent than real):

- In those suggestions, the result relations are effectively *elements* of those result sets.
- In the multirelation approach, by contrast, relations—that is, participant relations in a multirelation—are *subsets* of that multirelation, not elements of it.

To illustrate, here's what McGoveran has to say regarding such matters in reference [13] (italics and boldface in the original):

> [This 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 the "restrict and project 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.**

Here's an example of what I mean when I claim that MR-union can be used to address the "outer join" problem. Let S, SP, and SPJ be relvars (not multirelvars) for suppliers, shipments of parts, and shipments of parts to projects, respectively. Relvar S has key {S#}; relvar SP has key {S#,P#}; and relvar SPJ has key {S#,P#,J#}. Now suppose we want a report showing (a) suppliers in supplier number order, each such supplier being followed by (b) a list of parts in part number order, showing parts shipped by that supplier, each such part being followed by (c) a list of projects in project number order showing the projects that supplier supplies that part to. The multirelation denoted by the following expression provides all of the information needed for that report:

```
MR_UNION { MULTIRELATION ( S ) ,
           MULTIRELATION ( SP ) ,
           MULTIRELATION ( SPJ ) }
```

The tuples resulting from evaluation of this expression would have to be processed in a suitable sequence in order to meet McGoveran's requirement of "grouping like rows together." Perhaps that sequence could be specified like this:

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

The semantics of such a specification would have to be defined in such a way that (among other things) each S tuple comes immediately before its first matching SP tuple if any, and each SP tuple comes immediately before its first matching SPJ tuple if any. Of course, if the intuitively obvious foreign key constraints from SPJ to SP and from SP to S don't apply, then the report might display some anomalies.

SOME OUTSTANDING QUESTIONS

As noted near the beginning, this chapter is meant as a kind of discussion paper, not as any kind of definitive statement. Certainly there are quite a few loose ends to be tidied up. And I feel bound to say that, if the scheme overall is seen principally as an approach to the "missing information" problem—a problem to which, I say again, purely relational solutions already exist—then it does seem to involve a degree of complexity out of proportion to the problem it's meant to address. In particular, the possibility that certain attributes might be absent from certain tuples leads to a lot of complexity in connection with MR_WHERE clauses and related matters.

On the other hand, if the approach is felt to be worth pursuing, then there are many topics that need further investigation. Here are some of them (in no particular order):

- Is there a better way of dealing with absent attributes (in MR_WHERE clauses in particular)?
- Do we need additional multirelation comparison operators?
- Do we need additional constraint shorthands?
- What about the possibility of compensatory actions, such as cascade delete?
- What issues are raised by the possibility of defining virtual relvars on multirelvars?
- What about the possibility of virtual multirelvars?
- Can we pin down the specifics of the putative mapping between relations and multirelations in more detail?
- What are the implications for multirelations of the inheritance model as defined in reference [7]—especially with respect to "specialization by constraint"?
- How do aggregation and summarization work with multirelations?
- What's the relationship, if any, between multirelations and support for temporal data as described in reference [8]?
- Can we define a formal multirelation algebra?
- Can the compiler determine when attributes might be absent from one or more tuples in the results of arbitrary multirelational expressions?
- Can the compiler perform any other constraint inferencing—for example, determining that some MR-key constraint applies to the result of some multirelational expression?
- More generally, what problems, if any, can be solved with multirelations that can't be solved without them? (We think: None.)

ACKNOWLEDGMENTS

I'd like to thank Chris Date for his careful review of several earlier drafts of this chapter. Adrian Hudnott also reviewed an earlier draft and gave me useful comments. Dennis Ashley provided me with references [9] and [11], both of which are mathematical treatises that use the term *multirelation* (though it's not clear to me whether their use of the term refers to exactly the same concept, nor how close either of them is to the concept I've defined here).

REFERENCES AND BIBLIOGRAPHY

1. Hugh Darwen: "How to Handle Missing Information Without Using Nulls" (presentation slides), *www.thethirdmanifesto.com* (May 9th, 2003; revised May 16th, 2005). See also Chapter 23 of the present book.
2. Hugh Darwen and C. J. Date: "*The Third Manifesto,*" *ACM SIGMOD Record 24,* No. 1 (March 1995).
3. C. J. Date: "The Principle of Cautious Design," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).
4. C. J. Date: "The Closed World Assumption," in *Logic and Databases: The Roots of Relational Theory.* Victoria, B.C.: Trafford Publishing (2007).
5. C. J. Date: *The Relational Database Dictionary, Extended Edition.* Berkeley, Calif.: Apress (2008).

6. C. J. Date and Hugh Darwen: *Databases, Types, and the Relational Model: The Third Manifesto* (3rd edition). Boston, Mass.: Addison-Wesley (2006). See also Chapter 1 of the present book.

7. C. J. Date and Hugh Darwen: "The Inheritance Model" (Chapter 19 of the present book).

8. C. J. Date, Hugh Darwen, and Nikos A. Lorentzos: *Temporal Data and the Relational Model*. San Francisco, Calif.: Morgan Kaufmann (2003).

9. Roland Fraïssé and Norbert Sauer: *Theory of Relations.* New York, N.Y.: Elsevier Science (2000).

10. Maurice Gittens: "On Logical Mistakes and *The Third Manifesto*" (English version of an article that appeared in Dutch in *DB/M Magazine,* No. 2. Array Publications, Netherlands, April 2007). See Chapters 4-9 of the present book (Chapter 9 in particular is relevant to the theme of the present chapter).

11. Wim H. Hesselink: "Multirelations Are Predicate Transformers," *http://www.cs.rug.nl/~wim/pub/ whh318.pdf* (February 23rd, 2004).

12. Adrian Larner: "A New Model of Data," *http://www.btinternet.com/~adrian.larner/ database.htm* (undated).

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

14. Fabian Pascal: "The Final Null in the Coffin," *http://www.dbdebunk.com/publications.html* (September 2004).