The View Updating Problem:

Notes toward a Proposed Solution

offered as a basis for technical discussion

by

C. J. Date

Date of this WORKING DRAFT: June 12th, 2011

PREAMBLE :

Every scientific discipline has its share of unsolved problems … E.g.:

Mathematics:	Riemann Hypothesis
	/ still open after over 150 years */*
Computer science:	P = NP
	/ still open after some 40 years */*
Physics and cosmology:	"GUT"
	/ still no more than a search ??? */*
Database theory:	**View updating**
	/ not in the same league, but */*
	/ pragmatically important and of */*
	/ much theoretical interest */*

VIEW UPDATING TODAY IS VERY *AD HOC* :

- In the SQL standard
- In commercial SQL products
- In the technical literature (to some extent)

Quite usual to find that a given SQL DBMS will:

- Prohibit updates on logically updatable views
- Permit updates on logically nonupdatable views (?)
- Implement view updates in a logically incorrect way
- Do all of the above */* most likely in practice */*

Also, note the historical emphasis on restriction / projection / join views

We need to solve this problem !!!

THE SUPPLIERS-AND-PARTS DATABASE :

S

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

SP

SNO	PNO	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

P

PNO	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

BY THE WAY :

I do ***not*** use the term *database* to mean the DBMS !!!

DISCLAIMER :

The latest word … Certainly not the last!

I've been trying to get view updating "right" for several years

These slides represent my current thinking …*Where there are discrepancies with respect to things I've said earlier, what follows should be taken as superseding*

Constructive criticism is solicited (including implementation concerns, though I'm more interested in getting the model right first) … Still numerous loose ends and unanswered questions */* but I'm an optimist! */*

I assume you know *The Third Manifesto* and **Tutorial D** basics:

THE THIRD MANIFESTO :

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

- Proposal for future direction of data and DBMSs
- **D** = any language that conforms to *Manifesto* principles (generic name)
- **Tutorial D** = language used in *Manifesto* book and elsewhere as a basis for examples

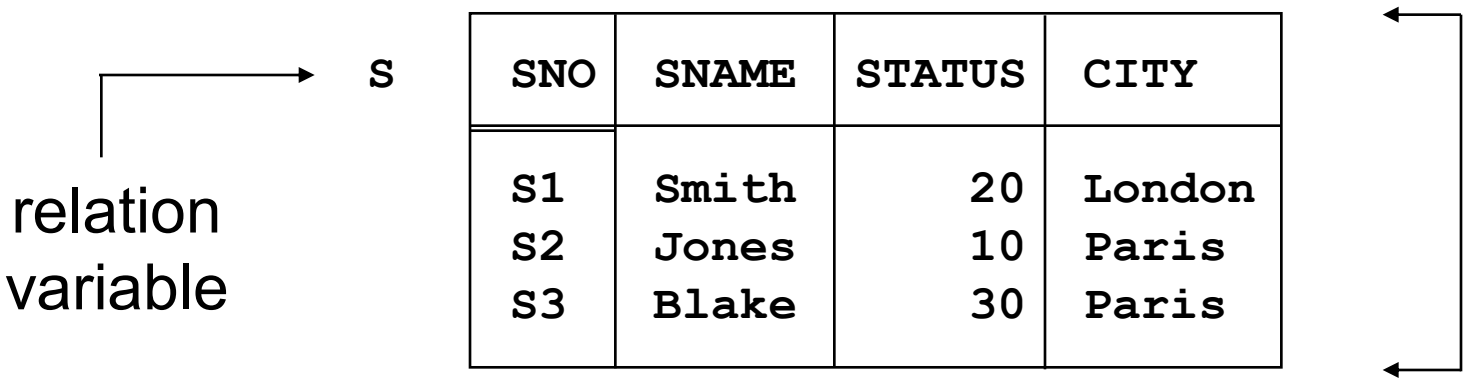
See www.thethirdmanifesto.com

I ASSUME YOU KNOW :

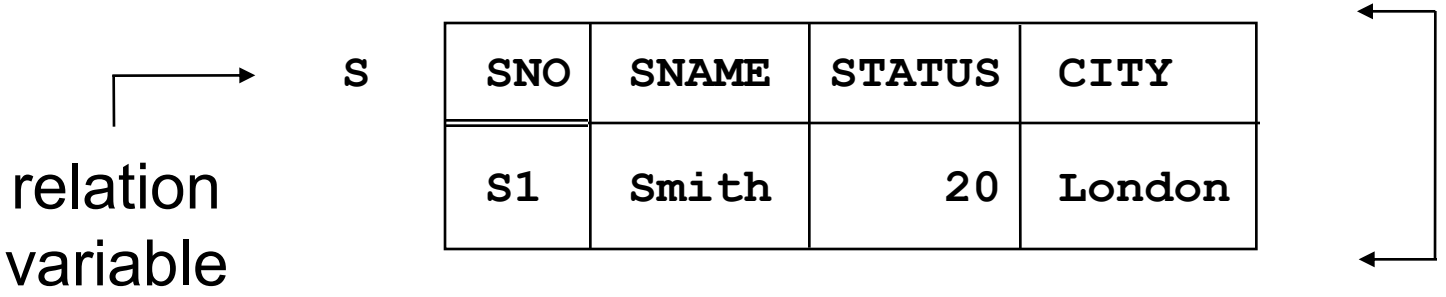
- There's a logical difference between relation *values* ("relations") and relation *variables* ("relvars")

- INSERT / DELETE / UPDATE are really shorthand for certain *relational assignments*

See next page for an illustration of both of these points




```
DELETE  S  WHERE CITY = 'Paris' ;
/* shorthand for: */
S  :=  S  MINUS ( S WHERE CITY = 'Paris' ) ;
```



I ASSUME YOU KNOW (cont.) :

- Assignment is *multiple,* in general: e.g.,

  DELETE … FROM S , DELETE … FROM SP ;

  / deliberately not quite valid* ***Tutorial D*** *syntax */*

  Shorthand for:

  S := … , SP := … ; */* semantically atomic */*

- Relvars are subject to *constraints* … Checked at *end of statement,* not deferred to end of transaction (i.e., checking is "immediate, not deferred")

  "The" relvar constraint for relvar *R* =
  logical AND of all constraints that mention *R*

I ASSUME YOU KNOW (cont.) :

- There's a logical difference* between *real* ("base") and *virtual* relvars ("views") … Views are ***relvars!***
- A view reference in a target position is really a *pseudovariable* reference (i.e., an operational expression appearing where a variable reference is expected): e.g.,

  DELETE *rx* FROM SSP ; */* assume SSP = S JOIN SP */*

  Shorthand for:

  (S JOIN SP) := (S JOIN SP) MINUS *rx* ;

  / not currently legal syntax in* ***Tutorial D*** **/*

  Henceforth "variable" includes pseudovariables as a special case

---

* *But not one that should affect the user!*

I ASSUME YOU KNOW (cont.) :

- There's a logical difference between database *values* and database *variables* ("databases") */* see next page */*
- Relvars are really "views" of the pertinent database!
- A relvar reference in a target position is really a pseudovariable reference, even if the reference is to a base relvar!

A QUOTE FROM *THE THIRD MANIFESTO* (slightly edited) :

The first version of this *Manifesto* drew a distinction between database values and database variables, analogous to the distinction between relation values and relation variables. It also introduced the term *dbvar* as shorthand for *database variable*. While we still believe this distinction to be a valid one, we found it had little direct relevance to other aspects of our proposals. We therefore decided, in the interests of familiarity, to revert to more traditional terminology.

Bad decision!

I ASSUME YOU KNOW (cont.) :

- Every relvar has a relvar *predicate* (user understood meaning of relvar in question): e.g.,

  Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY

- *The Closed World Assumption:* At any given time, relvar *R* contains all and only those tuples that satisfy the predicate for *R* at that time

  If a given tuple could appear but doesn't, we can assume it fails to satisfy the predicate

I ASSUME YOU KNOW (cont.) :

- There's a logical difference between relvar predicates and relvar constraints
- Constraints in **Tutorial D** typically, though not exclusively, take the form

  CONSTRAINT constraint name IS_EMPTY (*boolean expression*)

  E.g.:

```
CONSTRAINT …IS_EMPTY ( S WHERE STATUS < 1 )
CONSTRAINT … IS_EMPTY ( ( S JOIN P ) WHERE STATUS < 20
                                      AND PNO = 'P6' ) ;
```

- Familiar KEY and FOREIGN KEY syntax is effectively just shorthand

I ASSUME YOU KNOW (cont.) :

- MATCHING and NOT MATCHING are useful shorthands:

  r1 MATCHING *r2* = (*r1* JOIN *r2*) { *H1* }
  r1 NOT MATCHING *r2* = (*r1* MINUS (*r1* MATCHING *r2*))

  E.g.:

  S MATCHING SP

  / suppliers who supply at least one part */*

  S NOT MATCHING SP

  / suppliers who supply no parts at all */*

- EXTEND is a useful operator too. E.g.:

  EXTEND P : { GMWT := WEIGHT * 454 }

  / parts extended with their weight in grams */*

A PERSONAL BIBLIOGRAPHY :

"Updating Views" (1986): *Embarrassingly bad*

"Updating Union, Intersection, and Difference Views" and "Updating Joins and Other Views" (with David McGoveran, 1994): *Somewhat muddled, but (I believe) on the right lines*

"View Updating" (2006): *A little better*

"The Logic of View Updating" (2007): *A little better still*

"How to Update Views" (2010): *Still trying, but not there yet*

"Accessing and Updating Views and Relations [*sic!!*] in a Relational Database" (David McGoveran's patent, 2007)

CAVEAT :

As the previous page might suggest, this topic can be pretty confusing (and it's certainly quite controversial)

There's a *lot* of detail, and it's easy to lose your way in debates and discussions

In particular, it's easy to forget, especially in examples, which relvars are base ones and which ones are views

It's important to keep a clear head !!!

I'll try not to get confused myself …
but You Have Been Warned

AND ANOTHER :

Much current DB literature talks about “materialized views,” which is a contradiction in terms, pretty much (whole point about views as far as RM is concerned is: They’re *virtual*, not materialized)

And then typically goes on to abbreviate “materialized view” to just *view* (!) ...

So ubiquitously, in fact, that the unqualified term *view* has come to mean, almost always, a “materialized view,” and we no longer have a good term for *view* in its original sense

I ***do*** use the unqualified term *view* in its original sense … I do *not* use it to mean a “materialized view” … In fact, I don’t use this latter term at all */* and neither should you! */*

VIEWS SERVE TWO DISTINCT PURPOSES :

1. User *U1* who defines view *V* is aware of the expression *x* that defines *V* … *U1* can use the name *V* wherever expression *x* is intended, but such uses are really just shorthand

   E.g., *U1* might have perception

   S *and* SP, *plus* V = S JOIN SP

   but *U1* knows these relvars aren't all independent

2. User *U2* who's merely informed that *V* is available for use is, or should be, typically *not* aware of the expression *x* … To *U2, V* should look just like a base relvar* … *This is the important case!*

   / though updates should behave the same way in both cases */*

---

* Implies "CREATE TABLE" vs. "CREATE VIEW" was a psychological mistake at the very least … What about **Tutorial D** ???

TWO IMPORTANT PRINCIPLES :

The Golden Rule:

No database is ever allowed to violate any constraint

/ and therefore: */*

No relvar is ever allowed to violate any constraint

/ applies to views in particular */*

The Assignment Principle:

After assignment of *v* to *V*, *v* = *V* must evaluate to TRUE

/ applies to variables in general, to relvars in particular, */*
/ and in fact to the entire database … */*
/ NB: often violated in SQL !!! */*

AND ANOTHER :

Updating in RM is set (or *relation*) at a time:

- INSERT inserts a set of tuples / DELETE deletes a set of tuples / UPDATE updates a set of tuples / *relational assignment assigns a relation to a relvar*
- We often talk of (e.g.) inserting an individual tuple … Such talk is convenient but sloppy
- Integrity checking mustn't be done till all updating has been done … Set level operations are **not** a sequence of tuple level operations!

STRUCTURE OF PRESENTATION :

- Preliminaries */* now done */*
- A motivating example
- Projection views
- Join views
- Extension and summarization views
- Intersection, union, and difference views
- Miscellaneous issues

MOTIVATING EXAMPLE :

```
/* suppliers base relvar */
VAR S BASE RELATION { SNO CHAR , SNAME CHAR ,
                      STATUS INTEGER , CITY CHAR }
        KEY { SNO } ;

/* London suppliers view—a virtual relvar */
VAR LS VIRTUAL ( S WHERE CITY = 'London' )
          KEY { SNO } ;

/* non London suppliers view—another virtual relvar */
VAR NLS VIRTUAL ( S WHERE CITY ≠ 'London' )
            KEY { SNO } ;
```

Note: **Tutorial D** does allow KEY specifications in view definitions, as in these examples … and note too that {SNO} in LS is a *foreign* key referring to {SNO} in S */* similarly for NLS */*

CRUCIAL OBSERVATION :

Instead of S being real (or base) and LS and NLS virtual, *we could make LS and NLS real and S virtual*—S is the union of restrictions LS and NLS, and mapping works both ways:

```
VAR LS BASE RELATION
    { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
      KEY { SNO } ;

VAR NLS BASE RELATION
    { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
      KEY { SNO } ;

VAR S VIRTUAL ( LS UNION NLS )
      KEY { SNO } ;
```

Designs are *information equivalent* */* see next page */* ...
So which relvars are base ones and which virtual is arbitrary (formally speaking, at least) ... Hence:

The Principle of Interchangeability: There must be no arbitrary and unnecessary distinctions between base and virtual relvars !!! Virtual relvars should "look and feel" just like base ones to the user ... E.g., with respect to:

- Having keys or not
- *Integrity in general*
- "Entity integrity"
- Tuple IDs

And we MUST be able to update views !!!

INFORMATION EQUIVALENCE :

Let *DBD1* and *DBD2* be (logical) DB designs

Let *db1* and *db2* be database values conforming to *DBD1* and *DBD2*, respectively

Let *M12* and *M21* be mappings that transform *db1* into *db2* and *db2* into *db1*, respectively

Then *db1* and *db2* are information equivalent (i.e., for every query on *db1* there exists a query on *db2* that yields the same result, and vice versa)

Now let the foregoing be true for all applicable *db1* and *db2* ...

Then *DBD1* and *DBD2* per se are information equivalent

WHAT'S MORE :

Let *DBD1* and *DBD2* be information equivalent

Let *DB1* and *DB2* be database variables conforming to *DBD1* and *DBD2*, respectively

Let *db1* and *db2* be the current values of *DB1* and *DB2*, respectively */* db1 and db2 are information equivalent */*

Let *U1* be an update on *DB1* that, given *db1,* yields *db1'*

Then there must exist an update *U2* on *DB2* that, given *db2,* yields *db2'*, such that *db1'* and *db2'* are information equivalent

In particular, foregoing applies if *DB1* is base relvars only and *DB2* is views only

TURNING THIS AROUND :

Let *DBD1* and *DBD2* ***not*** be information equivalent ...

Let *DB1* and *DB2* be database variables conforming to *DBD1* and *DBD2*, respectively

Let *db1* and *db2* be the current values of *DB1* and *DB2*, respectively

Then there'll be queries and updates on *db1* that have no counterpart on *db2* or vice versa ** speaking a trifle loosely **

LONDON vs. NON LONDON SUPPLIERS : A CLOSER LOOK

By *The Principle of Interchangeability,* the behavior of relvars LS and NLS (and S) mustn't depend on which if any are base relvars and which if any are views …So let's imagine they're all base relvars :

```
VAR S   BASE RELATION { ... } KEY { SNO } ;
VAR LS  BASE RELATION { ... } KEY { SNO } ;
VAR NLS BASE RELATION { ... } KEY { SNO } ;
```

Following constraints obviously apply:

```
CONSTRAINT … LS  =  S WHERE CITY = 'London' ;
CONSTRAINT … NLS =  S WHERE CITY ≠ 'London' ;
CONSTRAINT … IS_EMPTY ( NLS WHERE CITY  = 'London' ) ;
CONSTRAINT … IS_EMPTY ( LS  WHERE CITY  ≠ 'London' ) ;
```

Foregoing constraints imply certain *additional* ones:

```
CONSTRAINT … S  =  UNION { LS , NLS } ;
                   /* every tuple in S also appears in either LS or NLS  */

                   /* so design involves some redundancy … see later */

CONSTRAINT … DISJOINT { LS , NLS } ;
                   /* no tuple appears in both LS and NLS …          */

                   /* … hence UNION in previous constraint could be   */
                   /* replaced by D_UNION (disjoint union) … In fact:  */

CONSTRAINT … DISJOINT { LS { SNO } , NLS { SNO } } ;
                   /* no SNO appears in both LS and NLS              */

     /* DISJOINT { relation expression commalist } is proposed as an */
     /* extension to current Tutorial D                               */
```

DELETE OPERATIONS ???

Clearly we'll need some *compensatory actions* (actually *cascade delete rules*) to keep the relvars in synch ... */* see next page */*

ON DELETE *ls* FROM LS :
    DELETE *ls* FROM S ;

ON DELETE *nls* FROM NLS :
    DELETE *nls* FROM S ;

ON DELETE *s* FROM S :
    DELETE (*s* WHERE CITY = 'London') FROM LS ,
    DELETE (*s* WHERE CITY ≠ 'London') FROM NLS ;

Compensatory action: An update performed automatically by the system in addition to some requested update, with the aim of avoiding some integrity violation that might otherwise occur. Cascading a delete operation is a typical example.

/ we're used to this idea in connection with foreign keys, */*
/ but actually it applies to constraints in general */*

Compensatory actions should be specified declaratively, and in general users should be aware of them; that is, users should know when their update requests are shorthand for some more extensive set of actions */* at least if the results of those actions are visible to the user */* , for otherwise they might perceive a violation of *The Assignment Principle*.

A COUPLE OF ASIDES :

"Compensatory actions should be specified declaratively" ...

I don't want to get into a debate as to exactly what *declarative* and *procedural* mean … I'll say only that the compensatory actions we'll be talking about are exactly as declarative or procedural as statements in **Tutorial D** are in general

Also, it has been claimed that compensatory actions go beyond the purview of the relational model …

But view updating doesn't */* views are relvars! */* , and view updating *requires* compensatory actions by definition

Note carefully that a compensatory action that (a) is performed implicitly (i.e., without the user's explicit knowledge of the relevant rule), but (b) whose effect is visible to the user, should fail on a violation of *The Assignment Principle*

Note: *This state of affairs could arise in connection with implicit view updates caused by updates to the underlying base relvar(s) and/or by implicit base relvar updates caused by updates to views*

… though in fact I believe the foregoing won't ever happen, under the view updating proposals to be described

Aside:

If S, LS, and NLS are indeed all base relvars, then compensatory actions are exactly what's needed to *control the redundancy* (i.e., they're what cause the necessary *update propagation* to occur)

DELETE OPERATIONS *bis* :

```
ON DELETE ls FROM LS :          /* from LS to S */
    DELETE ls FROM S ;

ON DELETE nls FROM NLS :        /* from NLS to S */
    DELETE nls FROM S ;

ON DELETE s FROM S :            /* from S to LS and NLS */
    DELETE ( s WHERE CITY = 'London' ) FROM LS ,
    DELETE ( s WHERE CITY ≠ 'London' ) FROM NLS ;
```

These rules must be visible to user (assuming user sees all three relvars), because of *The Assignment Principle*

Note: Syntax for expository purposes only

A POSSIBLE SIMPLIFICATION :

DELETE rule from S to LS and NLS—

```
ON DELETE s FROM S :
    DELETE ( s WHERE CITY = 'London' ) FROM LS ,
    DELETE ( s WHERE CITY ≠ 'London' ) FROM NLS ;
```

—could be harmlessly simplified to just:

```
ON DELETE s FROM S :
    DELETE s FROM LS ,
    DELETE s FROM NLS ;
```

In what follows, I'll occasionally make use of the fact that an attempt to delete a nonexistent tuple or to insert an existing one has no effect */* but I'll come back to this point later */*

ANOTHER GENERAL POINT :

Again consider DELETE rule from S to LS and NLS:

```
ON DELETE s FROM S :
    DELETE ( s WHERE CITY = 'London' ) FROM LS ,
    DELETE ( s WHERE CITY ≠ 'London' ) FROM NLS ;
```

DELETE on S causes DELETEs to be performed on LS and NLS … Those DELETEs might cause further compensatory actions to be performed (and so on)

Original update plus all associated compensatory actions must all be logically considered part of the original updating statement */* statements are atomic */*

Cascading stops when a "fixpoint" is reached (?)

AND ANOTHER :

"Compensatory actions should be specified declaratively" … So I'm not talking about *triggers,* which are typically procedural … *Also:*

- DBMS can't determine triggers for itself, in general
  / if it could, triggers as such wouldn't be necessary! */*
- Triggers typically concealed from the user
  / and thus might violate The Assignment Principle */*
- Triggers typically violate set level nature of RM
- Triggers in general aren't *logically required*

But triggers might be used to implement compensatory actions, if the DBMS doesn't support such actions

INSERT OPERATIONS :

ON INSERT *ls* INTO LS :
    INSERT *ls* INTO S ; /* *cascade insert from LS to S* */

ON INSERT *nls* INTO NLS :
    INSERT *nls* INTO S ; /* *cascade insert from NLS to S* */

ON INSERT *s* INTO S :
    INSERT (*s* WHERE CITY = 'London') INTO LS ,
    INSERT (s WHERE CITY ≠ 'London') INTO NLS ;
                /* *cascade insert from S to LS and NLS* */

Note: Compensatory actions don't always have to be simple "cascades" /* *more complex examples later* */

WHAT ABOUT UPDATE ???

For S, LS, and NLS, can treat as DELETE + INSERT … E.g.:

1. UPDATE NLS WHERE SNO = ‘S2’ : { CITY := ‘Oslo’ } ;

   Deletes “old” tuple for S2 from NLS and therefore S
   Inserts “new” tuple for S2 into NLS and therefore S

WHAT ABOUT UPDATE ???

For S, LS, and NLS, can treat as DELETE + INSERT … E.g.:

1. UPDATE NLS WHERE SNO = ‘S2’ : { CITY := ‘Oslo’ } ;

   Deletes “old” tuple for S2 from NLS and therefore S
   Inserts “new” tuple for S2 into NLS and therefore S

2. UPDATE S WHERE SNO = ‘S2’ : { CITY := 'London' } ;

   Deletes “old” tuple for S2 from S and therefore NLS
   Inserts “new” tuple for S2 into S and therefore *LS* ...
   Loosely, tuple for S2 “migrates” from NLS to LS !!!

WHAT ABOUT UPDATE ???

For S, LS, and NLS, can treat as DELETE + INSERT … E.g.:

1. UPDATE NLS WHERE SNO = ‘S2’ : { CITY := ‘Oslo’ } ;

   Deletes “old” tuple for S2 from NLS and therefore S
   Inserts “new” tuple for S2 into NLS and therefore S

2. UPDATE S WHERE SNO = ‘S2’ : { CITY := 'London' } ;

   Deletes “old” tuple for S2 from S and therefore NLS
   Inserts “new” tuple for S2 into S and therefore *LS* ...
   Loosely, tuple for S2 “migrates” from NLS to LS !!!

3. UPDATE NLS WHERE SNO = ‘S2’ : { CITY := 'London' } ;

   Deletes “old” tuple for S2 from NLS and therefore S
   Tries to insert “new” tuple for S2 into NLS ... but fails
   (**Golden Rule** violation on NLS): *so UPDATE rejected*

HOWEVER :

For S, LS, and NLS, can treat UPDATE as DELETE + INSERT, as we've seen … But explicit UPDATE rules will sometimes be necessary. E.g., for S and SP */* *still assuming base relvars only* */* :

```
ON DELETE s FROM S :
    DELETE ( SP MATCHING s ) FROM SP ;  /* let's assume */

/* no "cascade insert" from S to SP … hence: */

ON UPDATE s { SNO := sno } IN S :
    UPDATE ( SP MATCHING s ) { SNO := sno } IN SP ;
```

Without such a rule, an UPDATE on S—e.g., UPDATE S WHERE SNO = 'S1' : { SNO := 'S8' }—could cause tuples to be lost from SP

AND NOW … THE POINT !!!

All of the foregoing applies effectively unchanged if some or all of the relvars concerned are views !!!

E.g., suppose S is a base relvar and LS and NLS are views (as in original example). Then:

1. Compensatory actions from S to LS and NLS:

   ON DELETE *s* FROM S :
   DELETE (*s* WHERE CITY = 'London') FROM LS ,
   DELETE (*s* WHERE CITY ≠ 'London') FROM NLS ;

   ON INSERT *s* INTO S :
   INSERT (*s* WHERE CITY = 'London') INTO LS ,
   INSERT (s WHERE CITY ≠ 'London') INTO NLS ;

   These happen "automatically" … but must be visible if user sees all three relvars

THE POINT (cont.) :

2. Compensatory actions from LS and NLS to S—i.e., *view updating rules*—are as follows:

ON DELETE *ls* FROM LS :
DELETE *ls* FROM S ;

ON DELETE *nls* FROM NLS :
DELETE *nls* FROM S ;

ON INSERT *ls* INTO LS :
INSERT *ls* INTO S ;

ON INSERT *nls* INTO NLS :
INSERT *nls* INTO S ;

These also happen “automatically” (with the obvious straightforward implementation) … but again rules must be visible if user sees all three relvars

Now consider a user who sees only views LS and NLS. That user:

1. Thinks of LS and NLS as base relvars (with key {SNO} in both cases)

2. Knows the corresponding predicates:

   LS: *Supplier SNO is under contract … and CITY is London*

   NLS: *Supplier SNO is under contract … and CITY isn't London*

3. Knows about the following constraints:

```
CONSTRAINT … IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
CONSTRAINT … IS_EMPTY ( LS  WHERE CITY ≠ 'London' ) ;
CONSTRAINT … DISJOINT { LS { SNO } , NLS { SNO } } ;
```

/ these constraints refer to relvars LS and NLS only, */*
/ not to relvar S (user doesn't even know S exists!) */*

4. Does *not* know about any compensatory actions (in either direction)

/ because those actions all refer to relvar S */*

All updates work exactly as expected !!!

Observe in particular that an update such as

```
UPDATE NLS WHERE SNO = 'S2' : { CITY := 'London' } ;
```

does ***not*** cause the tuple for S2 to "migrate" from NLS to LS !!!

/ note that such migration might have been expected,*/*
/ given traditional approaches to view updating */*

Instead, it fails on a violation of **The Golden Rule** on relvar (view) NLS

But what about a user who sees only view NLS (and thinks of it as a base relvar, and knows corresponding predicate) ???

```
VAR NLS … { … } KEY { SNO } ;
```

User knows about key constraint and this one (but no others):

```
CONSTRAINT … IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
```

This user mustn't be allowed to INSERT into this relvar, nor to UPDATE SNO in this relvar (because such operations might violate constraints of which this user is *and must be* unaware) … Reasonably, since we don't have information equivalence

Just like a user who sees relvar SP and not relvar S, in fact (where analogous prohibitions apply)

ASIDE :

Suppose we can DELETE but not INSERT on relvar *R* …

A base relvar that doesn't support INSERTs ??? …
In particular, can't DELETE tuple *t* from *R* and then immediately (re)INSERT tuple *t* into *R* ?!?

If such behavior is unacceptable, we'll have to allow INSERTs, on the understanding that the system will sometimes have to tell the user that he or she can't do a given INSERT simply "because I say so"

Note: Similar remarks apply (potentially) whenever it's the case that DELETE but not INSERT, or INSERT but not DELETE, is supported

WHAT I'M SUGGESTING :

Conclusions from motivating example are all very obvious …
But I claim that *thinking of views as base relvars "living alongside" the relvars in terms of which they're defined is a fruitful way to think about view updating in general*

1. View defining expressions imply constraints */* so note requirement for system to do **constraint inference** */*

2. Constraints—*often automatically*—imply compensatory actions

I'm *not* suggesting that the DBA should have to specify, explicitly, all of the various constraints and compensatory actions that apply in any given situation!

A NOTE ON TERMINOLOGY :

(Virtual) relvars LS and NLS in motivating example are "restriction views"

I.e., value of relvar LS (for example) at any given time *t* is a certain restriction of value of relvar S at that same time *t*

But restriction (like all operators of the relational algebra) applies to *relations,* not *relvars …*

However, talk of "restriction views," "projection views," "join views" (etc.) is conventional shorthand in this line of work

/ akin to the way we talk when we do normalization */*

MORE ON RESTRICTION VIEWS :

Base relvar S { SNO , SNAME , STATUS , CITY }
/ ignore attribute type specs for simplicity */*

Restriction views NLS and NPS ...

```
CONSTRAINT … NLS  =  S WHERE CITY  ≠ 'London' ;
CONSTRAINT … NPS  =  S WHERE CITY  ≠  'Paris' ;

CONSTRAINT … NLS WHERE CITY  ≠  'Paris' =
                    NPS WHERE CITY  ≠ 'London' ;
```

Views “overlap” … implying (automatic) additional cascades from one to the other ...

E.g., consider INSERTs (only, for simplicity):

INSERT RULES :

```
ON INSERT s INTO S :          /* implicit (happens automatically) */
     INSERT ( s WHERE CITY ≠ 'Paris' ) INTO NPS ,
     INSERT ( s WHERE CITY ≠ 'London' ) INTO NLS ;

ON INSERT nls INTO NLS :     /* view update rule */
     INSERT nls INTO S ,
     /* and therefore: */
     INSERT ( nls WHERE CITY ≠ 'Paris' ) INTO NPS ;

ON INSERT nps INTO NPS :  /* view update rule */
     INSERT nps INTO S ,
     /* and therefore: */
     INSERT ( nps WHERE CITY ≠ 'London' ) INTO NLS ;
```

So a user who sees views NLS and NPS (only), and thinks of them as base relvars, will see certain cascades from one to the other

A violation of *The Principle of Interchangeability* ???

Well … given that the foregoing "cascade insert" rules do apply when NLS and NPS are views, *they should apply when they're base relvars too !!!*

In fact, if they don't, then, e.g., inserting (S6,…,Oslo) into NLS only will lead to a violation of both **The Golden Rule** (the "overlapping" constraint) and *The Closed World Assumption*

/ I realize I'm advocating a somewhat novel approach here regarding */*

/ how relational systems really ought to behave … */*

STRUCTURE OF PRESENTATION :

- Preliminaries
- A motivating example */* restriction views */*
- Projection views
- Join views
- Extension and summarization views
- Intersection, union, and difference views
- Miscellaneous issues

PROJECTION VIEWS : EXAMPLE

Base relvar S { SNO , STATUS , CITY }
/ ignore attribute SNAME for simplicity */*

Projection views ST { SNO , STATUS } and SC { SNO , CITY }

```
CONSTRAINT … ST  =  S { SNO , STATUS } ;
CONSTRAINT … SC  =  S { SNO , CITY } ;
```

/ implicitly: */*

```
CONSTRAINT … IDENTICAL { S { SNO } , ST { SNO } , SC { SNO } } ;
CONSTRAINT … S  =  ST JOIN SC ;       /* nonloss decomposition! */
```

/ IDENTICAL { relation expression commalist } is proposed as an */*
/ extension to current **Tutorial D*** */

So consider a user who sees only views ST and SC. That user:

1. Thinks of ST and SC as base relvars (each with key {SNO})

2. Knows the corresponding predicates:

   ST: *Supplier SNO is under contract and has status STATUS*

   SC: *Supplier SNO is under contract and is located in city CITY*

3. Is aware of the following constraint:

   CONSTRAINT … IDENTICAL { ST { SNO } , SC { SNO } } ;

4. Compensatory actions … ???

Example strongly suggests that some updates, at least, must be *multiple assignments** … E.g.:

INSERT (S9,20) INTO ST , INSERT (S9,London) INTO SC ;

This one is clearly OK … But what about:

INSERT (S9,20) INTO ST ;

Golden Rule violated! … because ST { SNO } ≠ SC { SNO }
/ if INSERT allowed to succeed */*

Point is: No "cascade insert rule" from ST to S makes sense (no CITY value available)—*while some kind of "default values" mechanism might be built on top of the proposed view updating scheme, I'm not advocating such a mechanism here*

** Would be true even if ST and SC were base relvars!*

Note the implication:

If no compensatory action exists in some particular situation, then certain updates will fail on violations of **The Golden Rule**

It's *not* that certain views are intrinsically nonupdatable!

Rather, it's that certain updates fail on certain views *because they violate certain integrity constraints* */* or because they violate The Assignment Principle, possibly */*

Just as with base relvars, in fact … I believe *all* views are potentially updatable, modulo such violations

BACK TO THE EXAMPLE :

What about DELETE?

DELETE (S1,20) FROM ST , DELETE (S1,London) FROM SC ;

This one is *certainly* OK … But what about:

DELETE (S1,20) FROM ST ;

This one is *probably* OK … Depends on what compensatory actions are in effect ** see next few pages **

PROJECTION EXAMPLE (cont.) :

Compensatory actions:

ON INSERT *s* INTO S : */* implicit (happens automatically) */*
INSERT *s* { SNO , STATUS } INTO ST ,
s { SNO , CITY } INTO SC ;

ON INSERT *st* INTO ST , *sc* INTO SC : */* view updating rule */*
INSERT (*st* JOIN *sc*) INTO S ;

Notes:

1. *st* {SNO} and *sc* {SNO} must be equal
2. User who sees just ST or just SC can't do INSERTs (or UPDATEs on SNO)

PROJECTION EXAMPLE (cont.) :

```
ON DELETE s FROM S :  /* implicit (happens automatically) */
      DELETE ( ST MATCHING s ) FROM ST ,
                   ( SC MATCHING s ) FROM SC ;

ON DELETE st FROM ST :  /* view updating rule */
      DELETE ( S MATCHING st ) FROM S ,
      /* and therefore: */
     DELETE ( SC MATCHING st ) FROM SC ;

ON DELETE sc FROM SC :  /* view updating rule */
      DELETE ( S MATCHING sc ) FROM S ,
      /* and therefore: */
     DELETE ( ST MATCHING sc ) FROM ST ;
```

NOTE :

First of the foregoing DELETE rules—

```
ON DELETE s FROM S :
    DELETE ( ST MATCHING s ) FROM ST ,
           ( SC MATCHING s ) FROM SC ;
```

—can equivalently be stated thus:

```
ON DELETE s FROM S :
    DELETE s { SNO , STATUS } FROM ST ,
           s { SNO , CITY } FROM SC ;
```

In what follows, I'll tend to favor this latter (possibly clearer) style, where applicable

PROJECTION EXAMPLE (cont.) :

```
ON UPDATE s { SNO := sno , STATUS := t , CITY := c } IN S :
     UPDATE ( ST MATCHING s ) { SNO := sno , STATUS := t } IN ST ,
     UPDATE ( SC MATCHING s ) { SNO := sno , CITY := c } IN SC ;

ON UPDATE st { SNO := sno , STATUS := t } IN ST :
     UPDATE ( S MATCHING st ) { SNO := sno , STATUS := t } IN S ,
     /* and therefore: */
     UPDATE ( SC MATCHING st ) { SNO := sno } IN SC ;

ON UPDATE sc { SNO := sno , CITY := s } IN SC :
     UPDATE ( S MATCHING sc ) { SNO := sno , CITY := c } IN S ,
     /* and therefore: */
     UPDATE ( ST MATCHING sc ) { SNO := sno } IN ST ;
```

NET OF THE FOREGOING :

Reprise: Consider user who sees only views ST and SC, and thinks of them as base relvars ...

That user is aware of following constraint (as well as predicates* and key constraints for ST and SC):

```
CONSTRAINT … ST { SNO } = SC { SNO } ;

/* hence INSERTs must be multiple */
```

That user is also aware of following compensatory actions:

---

* Supplier SNO is under contract and has status STATUS;
supplier SNO is under contract and is in city CITY

NET OF THE FOREGOING (cont.) :

ON DELETE *st* FROM ST :
    DELETE (SC MATCHING *st*) FROM SC ;

ON DELETE *sc* FROM SC :
    DELETE (ST MATCHING *sc*) FROM ST ;

ON UPDATE *st* { SNO := *sno* } IN ST :
    UPDATE (SC MATCHING *st*) { SNO := *sno* } IN SC ;

ON UPDATE *sc* { SNO := *sno* } IN SC :
    UPDATE (ST MATCHING *sc*) { SNO := *sno* } IN ST ;

All updates work exactly as expected !!!

But what about a user who sees only view ST (and thinks of it as a base relvar, and knows corresponding predicate) ???

VAR ST … { … } KEY { SNO } ;

As noted earlier, this user probably shouldn't be allowed to INSERT into this relvar, nor to UPDATE SNO in this relvar (because such operations might violate constraints of which this user is unaware) … though DELETEs might be supported, as already explained … */* all reasonable, given lack of information equivalence … I won't keep on saying this */*

Again like a user who sees relvar SP and not relvar S, where analogous prohibitions apply

AND WHAT IF …

… the projection view fails to retain some key of the underlying relvar ???

E.g., V = S { STATUS , CITY }

INSERT: Must fail (no sensible compensatory action)

DELETE: *Might* define a "cascade delete" rule but probably not a good idea

UPDATE: Might not *too* unreasonably define a "cascade update" rule */* e.g., suppose {CITY} → {STATUS} holds */*

And what about a user who sees LS { ALL BUT CITY } ??? … DELETE and UPDATE (not on SNO!) probably supported, INSERT probably not

For simplicity, let's agree to ignore UPDATE rules from this point forward, unless there's some special point to be made regarding them

STRUCTURE OF PRESENTATION :

- Preliminaries
- A motivating example
- Projection views
- Join views
- Extension and summarization views
- Intersection, union, and difference views
- Miscellaneous issues

JOIN VIEWS : EXAMPLE 1

Base relvars ST { SNO , STATUS } and SC { SNO , CITY }
/ ignore SNAME for simplicity */*

Join view S { SNO , STATUS , CITY } */* "one to one" join */*

CONSTRAINT … S = ST JOIN SC ;

Suppose this design is supposed to be information equivalent to one in which S is a base relvar and ST and SC are projection views. Then:

CONSTRAINT … ST = S { SNO , STATUS } ;
CONSTRAINT … SC = S { SNO , CITY } ;
CONSTRAINT … IDENTICAL { S { SNO } , ST { SNO } , SC { SNO } } ;

Note: I'm NOT going to wind up with a requirement that the DBMS be able to infer these additional constraints for itself !!!

JOIN VIEWS : EXAMPLE 1 (cont.)

Compensatory actions (view updating rules) from S to ST and SC:

```
ON  INSERT s  INTO S :
      INSERT s { SNO , STATUS }  INTO ST ,
                 s { SNO , CITY }  INTO SC ;

ON  DELETE s FROM S :
      DELETE s { SNO , STATUS } FROM ST ,
                 s { SNO , CITY } FROM SC ;
```

These rules are are the same as the (implicit) rules from S to ST and SC in the (first, and most important) projection example

/ this shouldn't be a surprise, given information equivalence */*

JOIN VIEWS : Example 1 (cont.)

So consider a user who sees only view S. That user:

1. Thinks of S as a base relvar (with key {SNO})

2. Knows the corresponding predicate:

   Supplier SNO is under contract, has status STATUS, and is located in city CITY

3. Knows nothing of any other constraints or associated compensatory actions

4. Can indeed behave as if join view S were a base relvar

JOIN VIEWS : EXAMPLE 1a

Now suppose a supplier can appear in (base relvar) ST but not SC or vice versa … i.e., some suppliers have a status but no city or vice versa ** is join still "one to one"? **

Can directly insert into ST or SC without inserting into the other
… Implicit INSERT actions from ST and SC to view S are:

```
ON INSERT st  INTO ST :
    INSERT ( st JOIN SC ) INTO S ;

ON INSERT sc  INTO SC :
    INSERT ( sc JOIN ST ) INTO S ;
```

But we're more interested in INSERTs on view S:

JOIN VIEWS : EXAMPLE 1a (cont.)

Think of join view S as a base relvar … INSERT rule has to be:

```
ON INSERT s INTO S :
    INSERT s { SNO , STATUS } INTO ST ,
           s { SNO , CITY } INTO SC ;
```

Because cascading to just one of ST and SC (or neither!) would—in general—cause INSERT on S to violate *The Assignment Principle* on join view S */* though cascades will sometimes be no ops … i.e., cascaded inserts are really "insert unless already present" */*

So the INSERT rule is as for the "information equivalence" case ("join views Example 1")

JOIN VIEWS : EXAMPLE 1a (cont.)

As for DELETE: *Can directly delete from ST or SC without deleting from the other* … Implicit DELETE actions from ST and SC to view S are:

```
ON DELETE st  FROM ST :
    DELETE ( S MATCHING st ) FROM S ;

ON DELETE sc  FROM SC :
    DELETE ( S MATCHING sc ) FROM S ;
```

But we're (much!) more interested in DELETEs on view S:

JOIN VIEWS : EXAMPLE 1a (cont.)

Might consider any of the following as a possible DELETE rule on S:

1. ON DELETE *s* FROM S :
   DELETE *s* { SNO , STATUS } FROM ST ;

2. ON DELETE *s* FROM S :
   DELETE *s* { SNO , CITY } FROM SC ;

3. ON DELETE *s* FROM S :
   DELETE *s* { SNO , STATUS } FROM ST ,
   DELETE *s* { SNO , CITY } FROM SC ;

The classic “ambiguity” issue, in fact … This is exactly what critics complain about !!!

JOIN VIEWS : EXAMPLE 1a (cont.)

I claim we should go with Option 3 … Advantages:

- Symmetry */* see next page … avoids arbitrary choice … */*
  / avoids possible DBA involvement */*

- Symmetry with respect to INSERT rule

- Single universal rule for "delete via join" */* as we'll see */*

- Options 1 and 2 are safe only if it's definitely the case that a supplier can have a status but no city or vice versa … The DBMS can't know such is the case if all it knows is that S = ST JOIN SC!
  / see next page but one */*

REGARDING SYMMETRY :

Try to treat symmetrically what is symmetrical, and do not destroy wantonly any natural symmetry.

—G. Polya, *How to Solve It*

We expect that any symmetry found in the data and condition of the problem will be mirrored by the solution … Symmetry should result from symmetry.

—G. Polya, *Mathematical Discovery: On Understanding, Learning, and Teaching Problem Solving*

JOIN VIEWS : EXAMPLE 1a (cont.)

Possible assumptions on the part of the DBMS:

a. Suppliers can have a status but no city or vice versa

b. Suppliers must have both

Which assumption is safer ???

Consider DELETE on join view S under assumption a. … We could:

0. Reject the DELETE entirely (“insufficient information”)
1. Delete from just ST
2. Delete from just SC
3. Delete from both ST and SC

WHICH OPTION MAKES BEST SENSE ???

0. DELETE will fail when it could have succeeded

   And *should* have succeeded, if b. is in fact the case

1. and 2. How decide ???

   Might need DBA involvement … Might imply

   ST JOIN SC ≠ SC JOIN ST !!!

   And DELETE will fail but should have succeeded if b. is in fact the case

3. Avoids foregoing problems … and is the only sensible option (in fact, the logically correct option) if b. is the case

JOIN VIEWS : EXAMPLE 1a (cont.)

So I like Option 3 !!!

Conclusion: Go with Option 3 (and, effectively, assumption b.) …

So the DELETE rule is as for the "information equivalence" case ("join views Example 1")

/ this DOESN'T mean the DBMS has to infer those */*
/ additional constraints mentioned under Example 1 */*
/ … in fact, they might or might not apply */*

Note: Can avoid deleting from both, if desired, by not deleting from S in the first place—in particular, by not giving the user DELETE rights over the join view

/ yes, I know this is an invocation of The Groucho Principle, */*
/ which is why I don't insist on it */*

TALKING OF PRINCIPLES …

The Principle of Insufficient Reason: No [solution] should be favored of eligible possibilities among which there is no reason to choose.

—G. Polya, *Mathematical Discovery: On Understanding, Learning, and Teaching Problem Solving*

Some might use this principle to argue for choosing Option 0

But I believe I *have* given reasons—good and sufficient reasons—for choosing Option 3

ONE LAST POINT :

Suppose we could tell the DBMS, somehow, that assumption a. is in fact the *case* */* i.e., suppliers can have a status but no city or vice versa */*

At least Option 3 still works, even so

Aside: If suppliers MUST have either a status or a city but not both, the join is always empty … in which case DELETE is always a no op, and INSERT always fails or is a no op

JOIN VIEWS : EXAMPLE 2

Base relvars S { SNO , CITY } and P { PNO , CITY }
/ ignore other attributes for simplicity */*

Join view SCP { SNO , CITY , PNO } */* "many to many" join */*

```
CONSTRAINT … SCP = S JOIN P ;
CONSTRAINT … SCP = SCP { SNO , CITY } JOIN SCP { PNO , CITY } ;
```

Suppose this design is supposed to be information equivalent to one in which SCP is a base relvar and S and P are projection views. Then:

```
CONSTRAINT … S = SCP { SNO , CITY } ;
CONSTRAINT … P = SCP { PNO , CITY } ;
```

Again, however, I'm not going to wind up with a requirement that the DBMS be able to infer these additional constraints for itself

JOIN VIEWS : EXAMPLE 2 (cont.)

Compensatory actions (implicit) from S and P to SCP:

ON INSERT *s* INTO S : INSERT (*s* JOIN P) INTO SCP ;
ON INSERT *p* INTO P : INSERT (S JOIN *p*) INTO SCP ;

E.g., INSERT (S9,London) INTO S : INSERT (S9,London,*p*) INTO SCP
for all parts *p* in London

ON DELETE *s* FROM S :
DELETE (SCP MATCHING *s*) FROM SCP ;
ON DELETE *p* FROM P :
DELETE (SCP MATCHING *p*) FROM SCP ;

E.g., DELETE (S1,London) FROM S : DELETE (S1,London,*p*) FROM SCP
for all parts *p* in London … */* etc. */*
/ see later for an explanation of that "etc."! */*

JOIN VIEWS : EXAMPLE 2 (cont.)

Compensatory actions (view updating rules) from SCP to S and P:

```
ON INSERT scp  INTO SCP :
    INSERT scp { SNO , CITY } INTO S ,
    INSERT scp { PNO , CITY } INTO P ;
```

/ INSERT into S or P might be a no op … but in general will cause */*
/ further INSERTs into SCP (example on next page) */*

Loosely: Inserting a tuple *t* into SCP causes the appropriate projection of *t* to be inserted into S (unless it already exists),* and similarly for P

Same as the INSERT rule for a one to one join view !!!

---

* In particular, unless it already exists in SCP (loosely speaking)

E.G. :

INSERT (S9,London,P8) INTO SCP ;

inserts (S9,London) and (P8,London) into S and P, respectively, and

(S9,London,P1)	(S1,London,P8)
(S9,London,P4)	(S4,London,P8)
(S9,London,P6)	(S9,London,P8)

into SCP … Original INSERT causes compensatory action for INSERT on SCP to be invoked, which in turn causes actions for INSERT on S and P to be invoked

Doesn't violate *The Assignment Principle,* just so long as pertinent INSERT rule is visible …

```
ON INSERT scp  INTO SCP :
    INSERT ( scp { SNO , CITY } JOIN SCP { CITY , PNO } ) INTO SCP ,
    INSERT ( scp { PNO , CITY } JOIN SCP { CITY , SNO } ) INTO SCP ;
```

This rule is the INSERT rule for relvar SCP, as seen by a user who knows only about SCP and thinks of it as a base relvar …
It's a logical consequence of the rules from S and P to SCP and vice versa

JOIN VIEWS : EXAMPLE 2 (cont.)

By analogy, the foregoing INSERT rule suggests the following as the appropriate DELETE rule:

Deleting a tuple *t* from SCP causes the appropriate projection of *t* to be deleted from S unless it exists elsewhere in SCP (loosely speaking), and similarly for P ** might cause further DELETEs on SCP **/

The DELETE rule for a one to one join view is a degenerate case

Algorithmic definition */* for S only … P is analogous */* :

temp := SCP MINUS *scp* ; */* "delete" specified tuples from SCP */*
S := S MINUS SCP { SNO , CITY } ; */* keep S's with no SCP's at all */*
S := S UNION *temp* { SNO , CITY } ; */* keep S's with at least one SCP */*

Note: MINUS in first line here could be NOT MATCHING

JOIN VIEWS : EXAMPLE 2 (cont.)

Declarative version */* for both S and P */* :

```
ON DELETE scp FROM SCP :
    DELETE SCP { SNO , CITY } FROM S ,
    INSERT ( SCP MINUS scp ) { SNO , CITY } INTO S ,
    DELETE SCP { PNO , CITY } FROM P ,
    INSERT ( SCP MINUS scp ) { PNO , CITY } INTO P ;
```

/ might be a neater way to state this rule … note that it relies on */*
/ careful definition of multiple assignment with repeated targets */*

Examples on next page

So the rules, for both INSERT and DELETE, are essentially as for the one to one case (Example 1 and 1a) !!!

SCP

SNO	CITY	PNO
S1	London	P1
S1	London	P4
S1	London	P6
S2	Paris	P2
S2	Paris	P5
S3	Paris	P2
S3	Paris	P5
S4	London	P1
S4	London	P4
S4	London	P6

Deleting (S1,London,P1)
is a no op … *Not* a violation
of *The Assignment Principle*

Deleting (...,Paris,...)
will delete S2 and S3 from S,
P2 and P5 from P, and all
Paris tuples from SCP …
Not a violation of *The*
Assignment Principle

Aside:

Actually there *IS* a neater way to state the DELETE rule from SCP to S and P:

```
ON  DELETE scp FROM SCP :
    DELETE ( ( S MATCHING SCP )
                 NOT MATCHING ( SCP MINUS scp ) ) FROM S ,
    DELETE ( ( P MATCHING SCP )
                 NOT MATCHING ( SCP MINUS scp ) ) FROM P ;
```

This formulation does *not* rely on “careful definition of multiple assignment with repeated targets” */* see next page */*

Consider the multiple assignment:

R1 := *r1* , *R2* := *r2* , … , *Rn* := *rn* ;

Semantics */* slightly simplified … see below */* :

1. Evaluate source expressions *r1, r2, …, rn*
2. Execute individual assignments "simultaneously"
3. Do integrity checking

But if *R1* and *R2* are the same (*R*, say):

WITH (*R* := *r1*) : *R* := *r2* , … , *Rn* := *rn* ;

Example illustrates "careful definition of multiple assignment with repeated targets" … *and note that R could be a view*

So consider a user who sees only view SCP. That user:

1. Thinks of SCP as a base relvar (with key {SNO,PNO})

2. Knows the corresponding predicate:

   Supplier SNO is under contract and is located in the same city, CITY, as part PNO

3. Knows about the following constraint:

```
CONSTRAINT …                                    /* SCP not in 4NF! */
        SCP  =  SCP { SNO , CITY } JOIN SCP { PNO , CITY } ;
```

4. Knows about the following compensatory actions:

```
ON INSERT scp  INTO SCP :
     INSERT ( scp { SNO , CITY } JOIN SCP { CITY , PNO } ) INTO SCP ,
     INSERT ( scp { PNO , CITY } JOIN SCP { CITY , SNO } ) INTO SCP ;

ON DELETE scp FROM SCP :
     DELETE ( ( SCP MATCHING scp { SNO , CITY } )
                          NOT MATCHING ( SCP MINUS scp ) ) FROM SCP ,
     DELETE ( ( SCP MATCHING scp { PNO , CITY } )
                          NOT MATCHING ( SCP MINUS scp ) ) FROM SCP ;
```

These rules are logical consequences of the rules from S and P to SCP and vice versa … in fact, they're required in order to ensure that the MVDs {CITY} $\rightarrow\rightarrow$ {SNO} | {PNO} hold in SCP

JOIN VIEWS : EXAMPLE 2a

Now suppose a city can appear in S but not P or vice versa (i.e., some cities have a supplier but no part or vice versa) … What happens to the compensatory actions ???

I'll skip the detailed analysis (it's analogous to that for Example 1a) … The message is:

The rules are the same as those for Example 2

/ see earlier */*

JOIN VIEWS : EXAMPLE 3

Base relvars S { SNO , CITY } and SP { SNO , PNO , QTY }

Join view SSP { SNO , CITY , PNO , QTY } /* *"one to many" join* */

```
CONSTRAINT … IS_EMPTY ( SP NOT MATCHING S ) ; /* foreign key */
CONSTRAINT … SSP  =  S JOIN SP ;
CONSTRAINT … SP = SSP { SNO , PNO , QTY } ;
CONSTRAINT … SSP { SNO , CITY } KEY { SNO } ; /* {SNO} → {CITY} */
CONSTRAINT … SSP { SNO , PNO , QTY } KEY { SNO , PNO } ;
```

/ KEY specifications in CONSTRAINT statements proposed as an */*
/ extension to current* ***Tutorial D*** **/*

If this design is supposed to be information equivalent to one in which SSP is a base relvar and S and SP are views, then:

```
CONSTRAINT … S  =  SSP { SNO , CITY } ;
```

But let's *not* assume this!

JOIN VIEWS : EXAMPLE 3a

So a supplier can be represented in S and not SP...
Compensatory actions (implicit) from S and SP to (SP and) SSP:

ON INSERT *s* INTO S :
    do nothing to SP ;
ON INSERT *sp* INTO SP :
    INSERT (S JOIN *sp*) INTO SSP ;

ON DELETE *s* FROM S :
    DELETE (SP MATCHING *s*) FROM SP ,
    DELETE (SSP MATCHING *s*) FROM SSP ;
ON DELETE *sp* FROM SP :
    DELETE (SSP MATCHING *sp*) FROM SSP ;

But we're interested in INSERTs and DELETEs on view SSP:

JOIN VIEWS : EXAMPLE 3a (cont.)

```
ON INSERT ssp  INTO SSP :
    INSERT ssp { SNO , CITY } INTO S ,
    INSERT ssp { SNO , PNO , QTY } INTO SP ;
```

Inserting a tuple *t* into SSP causes the appropriate projection of *t* to be inserted into S (unless it already exists), and similarly for SP

```
ON DELETE ssp FROM SSP :
    DELETE ( ( S MATCHING SSP )
                NOT MATCHING ( SSP MINUS ssp ) ) FROM S ,
    DELETE ssp { SNO , PNO , QTY } FROM SP ;
```

Deleting a tuple *t* from SSP causes the appropriate projection of *t* to be deleted from S (unless it exists elsewhere in SSP), and similarly for SP

Same as for other join view cases !!!

JOIN VIEWS : Example 3a (cont.)

So consider a user who sees only view SSP. That user:

1. Thinks of SSP as a base relvar (with key {SNO,PNO} and FD {SNO} → {CITY})

2. Knows the corresponding predicate:

   Supplier SNO is under contract, is located in city CITY, and supplies part PNO in quantity QTY

3. Knows nothing of any other associated constraints or compensatory actions

4. Can indeed behave as if join view SSP were a base relvar*

---

* *Except for INSERT (S5,London,…) … ???*

So my claim is: Foregoing proposals represent a single uniform approach that works for all join views !!!

Makes no difference whether the join is one to one, many to many, or one to many, nor whether designs are information equivalent */* though many to many case does require certain compensatory actions to be visible to the user */*

To summarize: Generic rules for view *A* JOIN *B* …

INSERT : INSERT *A* (sub)tuples if they don't already exist,
INSERT *B* (sub)tuples if they don't already exist

DELETE : DELETE *A* (sub)tuples if they don't exist elsewhere,
DELETE *B* (sub)tuples if they don't exist elsewhere

STRUCTURE OF PRESENTATION :

- Preliminaries
- A motivating example
- Projection views
- Join views
- Extension and summarization views
- Intersection, union, and difference views
- Miscellaneous issues

EXTENSION VIEWS :

Base relvar P { PNO , WEIGHT }

Extension view PX { PNO , WEIGHT , GMWT }

```
CONSTRAINT … PX = EXTEND P : { GMWT := WEIGHT * 454 } ;
```

Compensatory actions:

```
ON INSERT p INTO P :
    INSERT ( EXTEND p : { GMWT := WEIGHT * 454 } ) INTO PX ;

ON DELETE p FROM P :
    DELETE ( EXTEND p : { GMWT := WEIGHT * 454 } ) FROM PX ;
        /* or ( PX MATCHING p ) … */
```

(cont.)

EXTENSION VIEWS (cont.) :

```
ON  INSERT px  INTO PX :
      /* tuples in px must satisfy GMWT = WEIGHT * 454 */
      INSERT px { PNO , WEIGHT } INTO P ;

ON  DELETE px  FROM PX :
      DELETE px { PNO , WEIGHT } FROM P ;
```

EXTENSION VIEWS (cont.) :

So consider a user who sees only view PX. That user:

1. Thinks of PX as a base relvar (with key {PNO})

2. Knows the corresponding predicate:

   Part PNO is of interest, has weight WEIGHT, and weight in grams GMWT

3. Knows that GMWT = WEIGHT * 454, updates on WEIGHT will "cascade" to GMWT, and GMWT itself is nonupdatable (?)

4. Can indeed behave as if extension view PX were a base relvar, albeit one with a computed or virtual attribute (?)

SUMMARIZATION VIEWS :

Base relvars S { SNO , CITY } and SP { SNO , PNO , QTY }

Summarization view STQ { SNO , TQ } … KEY { SNO }
/ supplier nos and total shipment quantities */*

```
CONSTRAINT … S { SNO } = STQ { SNO } ;

CONSTRAINT … STQ = SUMMARIZE SP PER ( S { SNO } ) :
                                    { TQ := SUM ( QTY ) } ;
```

/ Or (and I prefer this style): */*

```
CONSTRAINT … STQ = EXTEND S { SNO } : { TQ := SUM ( !!SP , QTY ) } ;
```
/ "!!SP" is an image relation reference */*

SUMMARIZATION VIEWS (cont.) :

DELETE rules from S and SP to STQ */* *implicit* */* ,
also from S to SP :

```
ON DELETE s FROM S :
    DELETE ( SP MATCHING s ) FROM SP ,
    DELETE ( STQ MATCHING s ) FROM STQ ;

ON DELETE sp FROM SP :
     DELETE ( STQ MATCHING sp ) FROM STQ ,
     INSERT ( EXTEND sp { SNO } :
                          { TQ := SUM ( !!SP , QTY ) } ) INTO STQ ;

/* if tuple for supplier sno is deleted from SP, tuple for sno will */
/* be deleted from STQ; remaining shipments for sno will then */
/* be resummarized and result inserted into STQ                */
```

SUMMARIZATION VIEWS (cont.) :

INSERT rules from S and SP to STQ */* *implicit* */* :

```
ON INSERT s INTO S :
     INSERT ( EXTEND s { SNO } : { TQ := 0 } ) INTO STQ ;

ON INSERT sp INTO SP :
      DELETE ( STQ MATCHING sp ) FROM STQ ,
      INSERT ( EXTEND sp { SNO } :
                              { TQ := SUM ( !!SP , QTY ) } ) INTO STQ ;

/* interestingly (but not unreasonably), the INSERT rule from SP to STQ */
/* is, formally, the same as the DELETE rule from SP to STQ !!!         */
```

SUMMARIZATION VIEWS (cont.) :

But what about compensatory actions (view updating rules) from STQ to S and SP ???

```
ON DELETE stq FROM STQ :
    DELETE ( S MATCHING stq ) FROM S ,
    /* and therefore */
    DELETE ( SP MATCHING stq ) FROM SP ;

ON INSERT stq INTO STQ :
    ?????
```

No obvious INSERT rule applies, so INSERTs on view STQ most likely won't be supported … More precisely, such INSERTs will fail on a **Golden Rule** violation … Except possibly if TQ = 0 ???

/ might allow UPDATEs on SNO (but not TQ) in STQ */*

STRUCTURE OF PRESENTATION :

- Preliminaries
- A motivating example
- Projection views
- Join views
- Extension and summarization views
- Intersection, union, and difference views
- Miscellaneous issues

INTERSECTION VIEWS :

Recall relvars (views?) NLS and NPS, with key constraints plus:

```
CONSTRAINT … NLS  =  S WHERE CITY  ≠ 'London' ;
CONSTRAINT … NPS  =  S WHERE CITY  ≠  'Paris' ;

CONSTRAINT … NLS WHERE CITY  ≠  'Paris'
                 =  NPS WHERE CITY  ≠ 'London' ;
```

Let view V = NLS INTERSECT NPS ...

Compensatory actions:

```
ON  INSERT v  INTO V :
      INSERT v  INTO NLS ,
      INSERT v  INTO NPS ;

ON DELETE v FROM V :
      DELETE v FROM NLS ,
      DELETE v FROM NPS ;
```

/ essentially as for JOIN … same arguments apply */*

UNION VIEWS :

Recall relvars (views?) LS and NLS, with key constraints plus :

CONSTRAINT … IS_EMPTY (NLS WHERE CITY = 'London') ;

CONSTRAINT … IS_EMPTY (LS WHERE CITY ≠ 'London') ;

CONSTRAINT … S = UNION { LS , NLS } ;

CONSTRAINT … DISJOINT { LS { SNO } , NLS { SNO } } ;

Let view V1 = LS UNION NLS */* *disjoint union* */* ...

UNION VIEWS (cont.) :

Compensatory actions:

ON DELETE *v* FROM V1 :

DELETE (*v* WHERE CITY = 'London') FROM LS ,

DELETE (*v* WHERE CITY ≠ 'London') FROM NLS ;

ON INSERT *v* INTO V1 :

INSERT (*v* WHERE CITY = 'London') INTO LS ,

INSERT (*v* WHERE CITY ≠ 'London') INTO NLS ;

"Disjointness" constraint implies that each tuple of *v* will be deleted from or inserted into exactly one of LS and NLS

UNION VIEWS (cont.) :

Let view V2 = NLS UNION NPS ** overlapping union **

Compensatory actions:

```
ON  DELETE v  FROM V2 :
      DELETE ( v  WHERE CITY  ≠ 'London' ) FROM NLS ,
      DELETE ( v  WHERE CITY  ≠  'Paris' ) FROM NPS ;

ON  INSERT v  INTO V2 :
      INSERT ( v  WHERE CITY  ≠ 'London' ) INTO NLS ,
      INSERT ( v  WHERE CITY  ≠  'Paris' ) INTO NPS ;
```

"Overlap" constraint implies that tuples of *v* for which the city is neither London nor Paris *must* be deleted from or inserted into both NLS and NPS ** else violation of **The Golden Rule** and The Closed World Assumption **

UNION VIEWS (cont.) :

In fact …

Compensatory actions with respect to union are exactly the same as those for restriction, mutatis mutandis

And this state of affairs shouldn’t come as a surprise !!!
(Cf. join vs. projection)

UNION VIEWS (cont.) :

Base relvars FRIEND { NAME } and CONTACT { NAME }

Let view V3 = FRIEND UNION CONTACT

/ overlapping union (?) */*

Compensatory actions:

ON DELETE *v* FROM V3 :
DELETE (*v* [*WHERE TRUE*]) FROM FRIEND ,
DELETE (*v* [*WHERE TRUE*]) FROM CONTACT ;

ON INSERT *v* INTO V3 :
INSERT (*v* [*WHERE TRUE*]) INTO FRIEND ,
INSERT (*v* [*WHERE TRUE*]) INTO CONTACT ;

Hmmm …

UNION VIEWS (cont.) :

Ambiguity again ??? … Again, this is the sort of thing critics complain about

But note that the union *loses information* … V3 isn't information equivalent to relvars FRIEND and CONTACT taken in combination … e.g., there are queries that can be done on the latter and not on the former

So if you want to do INSERTs on a design that loses information, I don't think you should complain about what happens

The rules are explicit and well defined … If you don't like what they do, then don't invoke them !!!

DIFFERENCE VIEWS :

Consider FRIEND { NAME } and CONTACT { NAME } again

Let view V = FRIEND MINUS CONTACT

Compensatory actions:

```
ON INSERT v INTO V :
    INSERT v INTO FRIEND ,
    DELETE v FROM CONTACT ;   /* might be a no op */

ON DELETE v FROM V :
    DELETE v FROM FRIEND ,
    INSERT v INTO CONTACT ;   /* might be a no op */
```

If “hidden” side effects on CONTACT aren’t acceptable, a user who sees just view V probably won’t be able to do INSERTs or DELETEs on V

STRUCTURE OF PRESENTATION :

- Preliminaries
- A motivating example
- Projection views
- Join views
- Extension and summarization views
- Intersection, union, and difference views
- Miscellaneous issues

1. A GENERAL OBSERVATION

Note that :

1. The problem of updating base relvars appropriately to support requested updates on views is exactly the same problem as …

2. The problem of updating stored data appropriately to support requested updates on base relvars !!!

Just shows up at different point in overall system architecture

We must solve this problem, otherwise we have to give up on the goal of data independence

AND ANOTHER :

For views where INSERT and DELETE are both supported, operations are inverses of each other (speaking a trifle loosely)

I.e., INSERT *t* + DELETE *t* and
DELETE *t* + INSERT *t*

both preserve the status quo

Note, however, that INSERT and DELETE aren't *always* inverses, even on base relvars … Consider, e.g., INSERT *t* + DELETE *t* on base relvar S, if *t* = (S1,…,London)

2. RELATIONAL ASSIGNMENT

It's easy to prove that any given relational assignment is equivalent to one of the form

R := (*r* MINUS *d*) UNION *i* ;

where

r = old value of *R*
d = tuples to be deleted
i = tuples to be inserted

and

r and *i* are disjoint
d $\subseteq$ *r*
d and *i* are unique and disjoint

So the original assignment is equivalent to:

INSERT *i* INTO *R* , DELETE *d* FROM *R* ;

Or to:

INSERT *i* INTO (*R* MINUS *d*) ;

Or to:

DELETE *d* FROM (*R* UNION *i*) ;

And so we don't need special rules for assignment

IN FACT :

Tutorial D provides two variants on INSERT and DELETE:

- D_INSERT: Inserting a tuple that already exists is an error
- I_DELETE: Deleting a tuple that doesn't exist is an error

So the original assignment is equivalent to:

D_INSERT *i* INTO *R* , I_DELETE *d* FROM *R* ;

What about D_INSERT and I_DELETE on a view ??? Appeal to *The Principle of Interchangeability* …

3. RELATION CONSTANTS

So far I've tacitly been assuming that all views are defined in terms of relation *variables* specifically …

What about a view defined—partly—in terms of a relation *constant* (or equivalent literal) ??? E.g.:

```
V = S JOIN
        RELATION { TUPLE { SNO 'S1' , PNO 'P1' , QTY 300 } ,
                                   ..........................................  }
```

How does updating work ??? *More study required (but I don't expect any showstoppers)*

4. SYNTAX NOT SEMANTICS ???

I'm on record as claiming that:

> *The semantics of view updating should not depend on the particular syntactic form in which the view definition in question happens to be expressed*

I.e., if expressions *rx* and *ry* denote the same relation at all times, then views *VX* (with defining expression *rx*) and *VY* (with defining expression *ry*) should exhibit identical behavior

But is this claim reasonable ??? ... Consider the following example:

RELVARS S AND SP (simplified) :

S

SNO
S1
S2
S5

SP

SNO	PNO
S1	P1
S1	P2
S1	P3
S2	P2

Let *rx* and *ry* be “SP” and “S JOIN SP”, respectively

DELETE (S2,P2) FROM *VY* (= S JOIN SP) …
Deletes (S2,P2) from SP and (S2) from S

DELETE (S2,P2) FROM *VX* (= SP) …
Deletes (S2,P2) from SP, leaves S unchanged (!?!)

/ both delete just* (S2,P2) *from the view, of course */*

RELVARS S AND SP (simplified) :

S

SNO
S1
S2
S5

SP

SNO	PNO
S1	P1
S1	P2
S1	P3
S2	P2

Let *rx* and *ry* be “SP” and “S JOIN SP”, respectively

INSERT (S4,P1) INTO *VY* (= S JOIN SP) …
Inserts (S4,P1) into SP and (S4) into S

INSERT (S4,P1) INTO *VX* (= SP) …
Fails on FK violation (!?!)

MORE EXAMPLES :

1. If *R1* and *R2* are of the same type, the following identity holds:

   R1 UNION (*R1* INTERSECT *R2*) = *R1*

   But updating via LHS will affect *R2*, while updating via RHS won't … ???

2. If *R1* and *R2* are of the same type, the following identity holds:

   R1 INTERSECT *R2* = *R1* MINUS (*R1* MINUS *R2*)

   But updating via LHS will affect *R1*, while updating via RHS won't … ???

SYNTAX NOT SEMANTICS (cont.) :

Such issues are very vexing !!!

But maybe we should be asking different questions ...

> When things get too complicated, it sometimes makes sense to stop and wonder: Have I asked the right question?
>
> —Enrico Bombieri, Fields Medalist and IBM von Neumann professor of mathematics at Institute for Advanced Study, Princeton

Aside: The relational model is a prize example of precisely this point!

SYNTAX NOT SEMANTICS (cont.) :

If *rx* and *ry* are equivalent, following are certainly true:

- Value of *VX* at time *t* = value of *VY* at time *t*
- Evaluating query *Q* on *VX* at time *t* = evaluating query *Q* on *VY* at time *t*
- For every update *UX* on DB of which *VX* is a part, there exists an update *UY* on DB of which *VY* is a part such that effect on *VX* of executing *UX* at time *t* = effect on *VY* of executing *UY* at time *t* */* and vice versa */*

But I see no reason why following has to be true too:

SYNTAX NOT SEMANTICS (cont.) :

Effect on *VX* of executing update *U* on *VX* at time *t* =
effect on *VY* of executing update *U* on *VY* at time *t* … ???

Here are some cogent reasons why this equation isn't necessarily valid:

1. There's a logical difference—in fact, a *semantic* difference—between (read-only) expressions and pseudovariable references

2. Either of *rx* and *ry* can mention a relvar that the other doesn't ** this happens in all of the examples … could the phenomenon occur even if it doesn't happen ??? **

SYNTAX NOT SEMANTICS (cont.) :

S

SNO
S1
S2
S5

SP

SNO	PNO
S1	P1
S1	P2
S2	P1
S2	P2

P

PNO
P1
P2
P4

VX = (S JOIN SP) JOIN P

VY = (S JOIN P) JOIN SP

DELETE (S1,P1) FROM *VX* : deletes (S1,P1) from SP (only)

DELETE (S1,P1) FROM *VY* : ditto !!!

/* I thought this might be a case where the phenomenon could */
/* occur even if neither of *rx* and *ry* mentions a relvar the other */
/* one doesn't, but I was wrong */

SYNTAX NOT SEMANTICS (cont.) :

Maybe the notion of information equivalence needs to be refined somewhat ???

- Information equivalence of *db1* and *db2* is a matter of *values*
- Information equivalence of *DB1* and *DB2* is a matter of *variables*

Logical difference !!!

SYNTAX NOT SEMANTICS (cont.) :

Foregoing logical difference is reminiscent of the distinction we draw in our inheritance model between value and variable substitutability

Another possibly relevant observation from our inheritance model:

Even if *R1* and *R2* are of the same type, the declared type of *R1* MINUS (*R1* MINUS *R2*) isn't, in general, the same as the declared type of *R1* INTERSECT *R2*

SYNTAX NOT SEMANTICS (cont.) :

So … I regard it as an interesting exercise to determine exactly when two expressions that are "read equivalent" can be treated as "update equivalent" also

Also: Does update equivalence imply read equivalence?

I suspect *The Principle of Orthogonal Design* might be relevant to these questions

Loosely: Avoid designs in which, if a given tuple appears in the DB at all, it must appear in more than one place */* like many of our examples, in fact! */*

5. USING RELVAR PREDICATES

Placemarker …

See Appendix C of "How to Update Views" (in *Database Explorations: Essays on The Third Manifesto and Related Topics,* by C. J. Date and Hugh Darwen, Trafford, 2010)

Briefly describes an idea due to David McGoveran involving (a) informing DBMS of relvar predicates and (b) appealing to those predicates explicitly in updates

Further details beyond the scope of this discussion

6. CONFORMANCE

Given that ...

1. Views are relvars

2. Relvars are variables

3. Variables are updatable (assignable to)

… can an implementation be said to be in conformance with the prescriptions of *The Third Manifesto* if it fails to support view updating ???

STRUCTURE OF PRESENTATION :

- Preliminaries
- A motivating example
- Projection views
- Join views
- Extension and summarization views
- Intersection, union, and difference views
- Miscellaneous issues