DATABASE Programming & Design

Aiming High With Rapid Application Development

Balancing Data Access With Database Security

Building a Better Meta-Data Framework

Client/Server's Demon: Storage Management

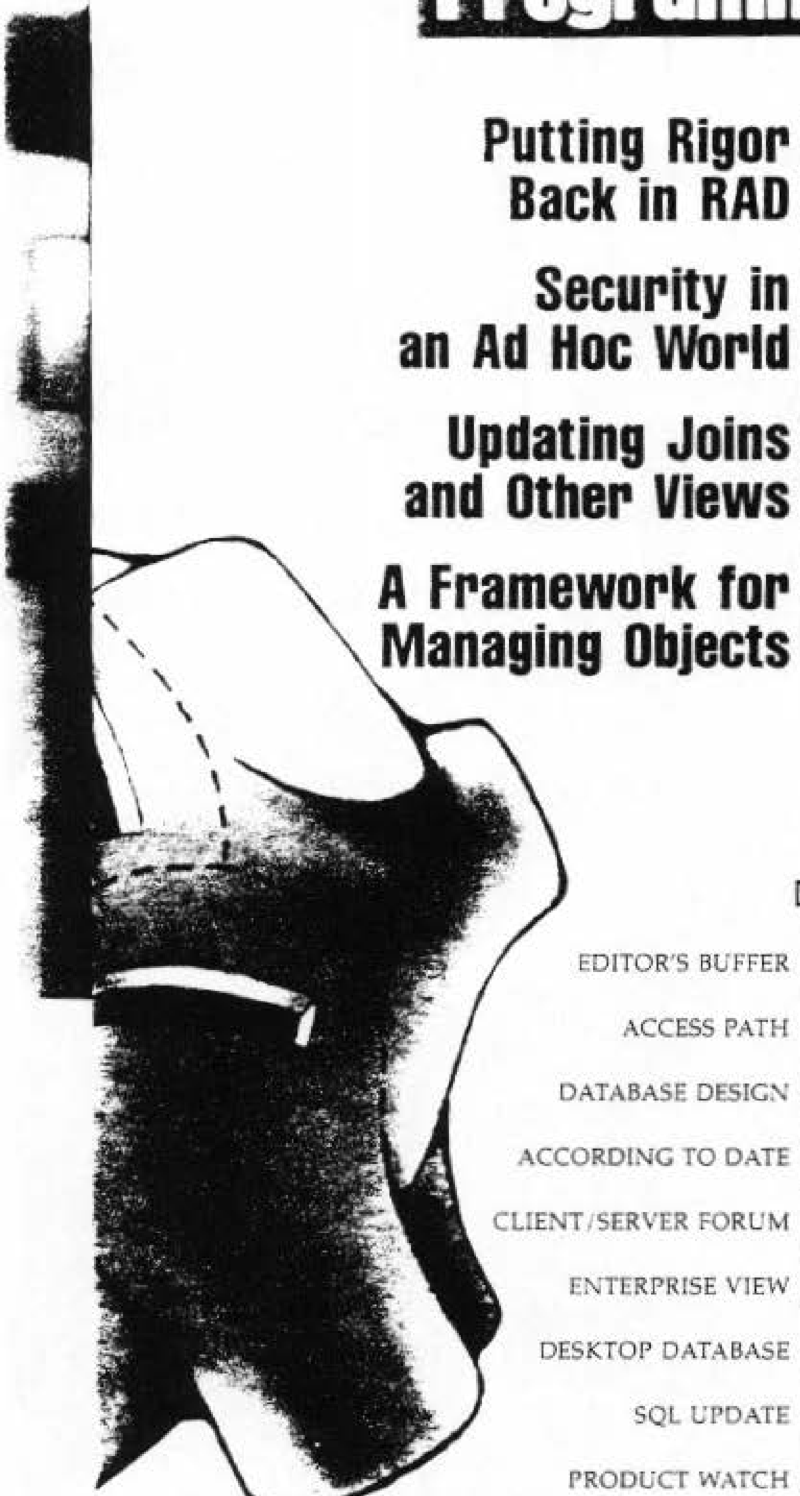
A MILLER FREEMAN PUBLICATION

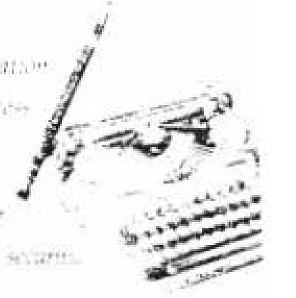
08
0 70989 35537 8

VOL. 7 NO. 8

DATABASE Programming & Design

AUGUST 1994





DATABASE PROGRAMMING & DESIGN (ISSN 0895-4518) is published monthly except in September, which is semi-monthly and contains the DATABASE PROGRAMMING & DESIGN Special Client/Server issue, by Miller Freeman, Inc., 600 Harrison St., San Francisco, CA 94107, (415) 905-2200. Please direct advertising and editorial inquiries to this address. For subscription inquiries, call (800) 289-0169 (outside U.S. (303) 447-9330). **SUBSCRIPTION RATE** for the U.S. is $47 for 13 issues (basic). Canadian/Mexican orders must be prepaid in U.S. funds with additional postage at $10 per year. Canadian GST Permit #124513185. All other countries outside the U.S. must be prepaid in U.S. funds with additional postage at $15 per year for surface mail or $40 per year for air mail. **POSTMASTER:** Send address changes to DATABASE PROGRAMMING & DESIGN, P.O. Box 53481, Boulder, CO 80322-3481. For subscriber service call toll-free (800) 289-0169 (in Colorado or outside the U.S. (303) 447-9330). Please allow six weeks for change of address to take effect. SECOND CLASS POSTAGE paid at San Francisco, CA 94107 and at additional mailing offices. **DATABASE PROGRAMMING & DESIGN** is a registered trademark owned by the parent company, Miller Freeman Inc. All material published in DATABASE PROGRAMMING & DESIGN is copyrighted © 1994 by Miller Freeman Inc. All rights reserved. Reproduction of material appearing in DATABASE PROGRAMMING & DESIGN is forbidden without permission. 16mm microfilm, 35mm microfilm, 105mm microfiche and article and issue photocopies are available from University Microfilms International, 300 N. Zeeb Rd., Ann Arbor, MI 48106 (313) 761-4700.

BY C. J. DATE AND DAVID McGOVERAN

Designers of database systems serving user-driven needs must take special care with view updating. Here is a systematic approach to a common problem

Updating Joins and Other Views

VIEW UPDATING HAS been the subject of considerable study for many years. In our June article,¹ we described an approach to the problem that seems more satisfactory (that is, more systematic and more robust) than previous proposals. In particular, we showed how our approach worked for union, intersection, and difference views. Our emphasis on those operators was deliberate—it was our feeling that union and the others, though important, had been largely neglected in most previous work. However, we clearly need to show how our approach applies to joins and other operators as well.

Note: The discussions of this article, like those in the June article, are quite informal. We intend to produce a formal description of our scheme as soon as time permits.

PRELIMINARIES

To begin the discussion, we should summarize the major points of the June article for easier reference.

The June article's major contribution was the identification of a series of principles that must be satisfied by any systematic view-updating mechanism. Of those principles, the first and overriding one follows:

1. A given row can appear in a given table only if that row does not cause the table predicate for that table to be violated. Moreover, this observation is just as true for a view as it is for a base table.

The *table predicate* for a given table is, loosely speaking, "what the table means"; it is the *update acceptability criterion* for that table. In other words, the table predicate for a given table constitutes the criterion for deciding whether or not some proposed update is in fact valid, or at least plausible, for that table. In the case of a base table, the table predicate is the logical AND of all column constraints and table constraints that apply to the base table in question. In the case of a derived table, the table predicate is derived in a straightforward way from the table predicate(s) for the table(s) from which the table in question is derived; for example, the table predicate PC for C = A INTERSECT B is (PA) AND (PB), where PA and PB are the table predicates for A and B, respectively.

The remaining principles follow:

2. All tables must be genuine relations (that is, duplicate rows are not allowed).

3. The updatability or otherwise of a given view is a semantic issue, not a syntactic one (that is, it must not depend on the particular form in which the view definition happens to be stated).

4. The view updatability rules must work correctly in the special case when the "view" is in fact a base table.

5. The rules must preserve symmetry where applicable.

6. The rules must take into account any applicable triggered actions, such as cascade DELETE.

7. For most purposes, UPDATE can be regarded as shorthand for a DELETE-then-INSERT sequence. Note, however, that no checking of table predicates is done "in the middle of" any given update; that is, the expansion of UPDATE is DELETE-INSERT-check, not DELETE-check-INSERT-check. Triggered actions are likewise never performed "in the middle of" any given update. Finally, some slight refinement is required to the shorthand in the case of projection views (see the section "Updating Projections" later for further discussion).

8. All update operations on views are implemented by the same kind of update operations on the underlying tables. That is, INSERTs map to INSERTs and DELETEs to DELETEs (we can ignore UPDATEs, thanks to the previous point).

9. The rules must be capable of recursive application.

10. The rules cannot assume that the database is well designed (though they might on occasion produce a slightly surprising result if the database is *not* well designed—a fact that can be seen in itself as an additional argument in support of good design).

11. If a view is updatable,

S

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

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

SP

S#	P#	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

FIGURE 1. *The suppliers-and-parts database (sample values).*

there should be no *prima facie* reason for permitting some updates but not others (for example, DELETEs but not INSERTs).

12. INSERT and DELETE should be inverses of each other, where possible.

We now discuss the application of these principles to updating views whose definition involves relational operations other than union, intersection, and difference. The major operators we consider are restriction, projection, extension, and join. We limit our attention to single-row updates for simplicity; however, we must first repeat the following remarks from our June article.

"*Important caveat:* The reader must understand that considering single-row updates only is in fact an *over*simplification, and indeed a distortion of the truth. Relational operations are always set-at-a-time; a set containing a single row is merely a special case. What is more, a multirow update is sometimes *required* (that is, some updates cannot be simulated by a series of single-row operations). And this remark is true of both base tables and views, in general. Suppose [the employees table EMP is subject to the constraint that employees] E8 and E9 must have the same salary. Then a single-row UPDATE that changes the salary of just one of the two will necessarily fail.

"Since our objective in this article is merely to present an *informal* introduction to our ideas, we will (as stated) describe the update rules in terms of single-row operations. But the reader should not lose sight of the [foregoing] important caveat."

One final preliminary remark: We should make it clear at the outset that for the operators under consideration, our rules (or some of them, at any rate) will probably not look all that different from those found in other proposals—at least at first glance. Nevertheless, we claim that our rules are more systematic than—for example—those of the SQL standard, and in any case part of the point is that our rules must be seen as a package; that is, the rules for join (and so forth) discussed in this article cannot be separated from the rules for union and so forth discussed in our June article.

SUPPLIERS-AND-PARTS

All the examples in this article are based on the familiar suppliers-and-parts database.[1] Here is a simplified definition for that database:

```
CREATE DOMAIN S# ... ;
CREATE DOMAIN NAME ... ;
CREATE DOMAIN STATUS ... ;
CREATE DOMAIN CITY ... ;
CREATE DOMAIN P# ... ;
CREATE DOMAIN COLOR ... ;
CREATE DOMAIN WEIGHT ... ;
CREATE DOMAIN QTY ... ;

CREATE BASE TABLE S
   ( S# DOMAIN ( S# ),
      SNAME DOMAIN ( NAME ),
      STATUS DOMAIN ( STATUS ),
      CITY DOMAIN ( CITY ) )
PRIMARY KEY ( S# ) ;

CREATE BASE TABLE P
   ( P# DOMAIN ( P# ),
      PNAME DOMAIN ( NAME ),
      COLOR DOMAIN ( COLOR ),
      WEIGHT DOMAIN ( WEIGHT ),
      CITY DOMAIN ( CITY ) )
PRIMARY KEY ( P# ) ;

CREATE BASE TABLE SP
   ( S# DOMAIN ( S# ),
      P# DOMAIN ( P# ),
      QTY DOMAIN ( QTY ) )
PRIMARY KEY ( S#, P# )
FOREIGN KEY ( S# ) REFERENCES S
FOREIGN KEY ( P# ) REFERENCES P ;
```

Figure 1 shows a set of sample data values for this simplified database definition.

Note: Throughout this article we use uppercase letters A, B, ... from near the beginning of the alphabet to refer generically to tables, and uppercase letters X, Y, ... from near the end of the alphabet to refer generically to columns of such tables. The table predicates for A, B, ... are PA, PB, ..., respectively. We use lowercase letters a, b, ... to refer generically to rows of tables A, B, ..., respectively.

UPDATING RESTRICTIONS

First of all, note that the table predicate for the table resulting from the restriction operation (also known as a selection operation):

A WHERE *condition*

(where condition is, specifically, a restriction condition)[2] is

```
( PA ) AND ( condition )
```

For example, the table predicate for the restriction S WHERE CITY = 'London' is:

```
( PS ) AND ( CITY = 'London' )
```

where PS is the table predicate for the suppliers table, S. It follows that (for example) any row *r* presented for insertion into a view defined by means of this restriction must be such that the conditions PS(r) and r.CITY = 'London' both evaluate to *true*, or the INSERT will fail.[3]

Here then are the update rules for A WHERE *condition:*

☐ **INSERT:** The new row must satisfy both PA and *condition.* It is inserted into A.

☐ **DELETE:** The row to be deleted is deleted from A.

☐ **UPDATE:** The row to be updated must be such that the updated version satisfies both PA and *condition*. The row is deleted from A without performing any triggered actions or table predicate checks. The updated version of the row is then inserted into A.

Examples:

Let view LS be defined as S WHERE CITY = 'London'. Figure 2 shows a sample tabulation of this view, corresponding to the sample tabulation of S shown in Figure 1.

☐ An attempt to insert the row <S6,Green,20,London> into LS will succeed. The new row will be inserted into table S, and will therefore be effectively inserted into the view as well.

☐ An attempt to insert the row <S1,Green,20,London> into LS will fail, because it violates the table predicate for table S—specifically, it violates the uniqueness constraint on the primary key S.S# for table S.

☐ An attempt to insert the row <S6,Green,20,Athens> into LS will fail, because it violates the restriction condition CITY = 'London'.

☐ An attempt to delete the LS row <S1,Smith,20,London> will succeed. The row will be deleted from table S, and will therefore be effectively deleted from the view as well.

☐ An attempt to update the LS row <S1,Smith,20,London> to <S6,Green,20,London> will succeed. An attempt to update that same row <S1,Smith,20,London> to either <S2,Smith,20,London> or <S1,Smith,20,Athens> will fail—in the first case because it violates the primary key uniqueness constraint on table S, in the second case because it violates the restriction condition CITY = 'London'.

UPDATING PROJECTIONS

Again, we begin by considering the relevant table predicate. Let the columns of table A be partitioned into two disjoint groups, say X and Y. Regard each of X and Y as a single *composite* column. It is clear, then, that a given row <x> will appear in the projection A[X] if and only if some value y exists from the domain of Y-values such that the row <x,y> appears in A. Consider the projection of table S over S#, SNAME, and CITY. Every row <S#,sname,city> appearing in that projection is such that a status value status exists such that the row <S#,sname,status,city> appears in table S.

LS

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

FIGURE 2. *View LS (sample values)*

Here then are the update rules for A[X]:

☐ **INSERT:** Let the row to be inserted be <x>. Let the default value of Y be y.[7] (It is an error if no such default value exists, that is, if Y has "defaults not allowed.") The row <x,y> (which must satisfy PA) is inserted into A.

Note: Since candidate keys will usually (but not invariably)[4] have "defaults not allowed," a projection that does not include all candidate keys of the underlying table will usually not permit INSERTs.

☐ **DELETE:** All rows of A with the same X-value as the row to be deleted from A[X] are deleted from A.

Note: In practice, it will usually be desirable that X include at least one candidate key of A, so that the row to be deleted from A[X] is derived from exactly one row a of A exists. However, there is no logical reason to make this a hard requirement. (Analogous remarks apply in the case of UPDATE also.)

☐ **UPDATE:** Let the row to be updated be <x> and let the updated version be <x'>. Let a be a row of A with the same X-value x, and let the value of Y in row a be y. All such rows a are deleted from A without performing any triggered actions or table predicate checks. Then, for each such value y, row <x',y> (which must satisfy PA) is inserted into A.

Note: The "slight refinement" mentioned in Principle No. 7 in the "Preliminaries" section shows itself here. Specifically, observe that the final "INSERT" step in the UPDATE rule reinstates the previous *y*-value in each inserted row—it does *not* replace it by the applicable default value, as a stand-alone INSERT would.

SC

S#	CITY
S1	London
S2	Paris
S3	Paris
S4	London
S5	Athens

FIGURE 3. *View SC (sample values)*

Examples:

Let view SC be defined as S [S#, CITY].

Figure 3 shows a sample tabulation of this view, corresponding to the sample tabulation of S shown in Figure 1.

☐ An attempt to insert the row <S6,London> into SC will succeed, and will have the effect of inserting the row <S6,n,t,London> into table S, where n and t are the default values for columns S.SNAME and S.STATUS, respectively.

☐ An attempt to insert the row <S1,London> into SC will fail, because it violates the table predicate for table S—specifically, it violates the uniqueness constraint on the primary key S.S# for table S.

☐ An attempt to delete the row <S1,London> from SC will succeed. The row for S1 will be deleted from table S.

☐ An attempt to update the SC row <S1,London> to <S1,Athens> will succeed; the effect will be to replace the row <S1,Smith,20,London> in table S by the row <S1,Smith,20,Athens>—*not* by the row <S1,n,t,Athens>, please observe.

☐ An attempt to update that same SC row <S1,London> to <S2,London> will fail (why, exactly?).

Exercise for the reader: Consider the case in which the projection does not include a candidate key of the underlying table—for example, the projection of table S over STATUS and CITY.

UPDATING EXTENSIONS

The reader might perhaps be unfamiliar with the relational EXTEND operator. Here is a brief explanation.[3] Basically, EXTEND takes a specified table and—conceptually, at least—returns a new (derived) table that is similar to the original table but includes an additional column, values of which are obtained by evaluating some specified computational expression. For example, we might write:

```
EXTEND P ADD ( WEIGHT * 454 ) AS GMWT
```

Assuming sample values for table P as given in Figure 1, the re-

P#	PNAME	COLOR	WEIGHT	CITY	GMWT
P1	Nut	Red	12	London	5448
P2	Bolt	Green	17	Paris	7718
P3	Screw	Blue	17	Rome	7718
P4	Screw	Red	14	London	6356
P5	Cam	Blue	12	Paris	5448
P6	Cog	Red	19	London	8626

FIGURE 4. *An example of EXTEND*

sult of this expression is shown in Figure 4. (The assumption is that WEIGHT values are given in pounds; the expression WEIGHT * 454 will convert those weights to grams.) Note that there is an exact one-to-one correspondence between rows of the extension and rows of the underlying table.

In general, the table predicate PE for the table E that results from the extension operation:

```
EXTEND A ADD exp AS X
```

is:

```
PA ( a ) AND e.X = exp ( a )
```

Here e is a row of table E and a is the projection of that row e over all columns of A. In stilted English:

"Every row e in the extension is such that (a) the row a that is derived from e by projecting away the value e.X satisfies PA, and (b) that value e.X is equal to the result of applying the expression *exp* to that row a."

For example, the table predicate for the extension of table P shown in Figure 4 is:

"Every row <p#,pname,color,weight,city,gmwt> in the extension is such that (a) the row <p#,pname,color,weight,city> satisfies the table predicate for P, and (b) the value gmwt is equal to the value 454 * weight."

Here, then, are the update rules for E = EXTEND A ADD exp AS X:

☐ INSERT: Let the row to be inserted be e; e must satisfy PE. The row a that is derived from e by projecting away the value e.X is inserted into A.

☐ DELETE: Let the row to be deleted be e. The row a that is derived from e by projecting away the value e.X is deleted from A.

☐ UPDATE: Let the row to be updated be e and let the updated version be e'; e' must satisfy PE. The row a that is derived from e by projecting away the value e.X is deleted from A without performing any triggered actions or table predicate checks. The row a' that is derived from e' by projecting away the value e'.X is inserted into A.

Examples (refer to Figure 4):

☐ An attempt to insert the row <P7,Cog,Red,12,Paris,5448> will succeed, and will have the effect of inserting the row <P7,Cog,Red,12,Paris> into table P.

☐ An attempt to insert the row <P7,Cog,Red,12,Paris,5449> will fail (why?).

☐ An attempt to insert the row <P1,Cog,Red,12,Paris,5448> will fail (why?).

☐ An attempt to delete the row for P1 will succeed, and will have the effect of deleting the row for P1 from table P.

☐ An attempt to update the row for P1 to <P1,Nut,Red,10,Paris,4540> will succeed; the effect will be to replace the row <P1,Nut,Red,12,London> in table P by with the row <P1,Nut,Red,10,Paris>.

☐ An attempt to update that same row to a row for P2 (with all other values unchanged) or a row in which the GMWT value is not equal to 454 * WEIGHT will fail (in each case, why?).

UPDATING JOINS

Most previous treatments of the view update problem have argued that the updatability or otherwise of a given join depends, at least in part, on whether the join is one-to-one, one-to-many, or many-to-many. By contrast, we contend that joins are *always* updatable. Moreover, the update rules are identical in all three cases and are essentially quite straightforward.

What makes this claim plausible—startling as though it might seem at first sight—is the new perspective on the problem afforded by adopting the fundamental principle we stated at the beginning of the "Preliminaries" section. Broadly speaking, the overall objective of support for the view mechanism has always been to make views look as much like base tables as possible, and this objective is indeed a laudable one. With regard to view update specifically, however:

☐ It is usually assumed (implicitly) that it is always possible to update an individual row of a base table independently of all the other rows in that base table.

☐ By contrast, it is manifestly *not* always possible to update an individual row of a view independently of all the other rows in that view. For example, Codd shows that it is not possible to delete just one row from a certain join, because the effect would be to leave a table that "is not the join of any two tables whatsoever" (which means that the result could not possibly satisfy the table predicate for the view). And the approach to such view updates historically has always been to reject them altogether, on the grounds that it is impossible to make them look completely like base table updates.

Our approach is rather different. We recognize the fact that even with a base table, it is not always possible to update individual rows independently of all the rest. (Consider what would happen if the suppliers base table were subject to the constraint that either suppliers S1 and S4 both appear or neither of them does.) Typically, therefore, we accept those view updates that have historically been rejected, interpreting them in an obvious and logically correct way to apply to the underlying table(s); we accept them, moreover, in full recognition of the fact that updating those underlying tables might well have side-effects on the view. *These side-effects are, however, required in order to avoid the possibility that the view might violate its own table predicate.*

With that preamble out of the way, let us now get down into details. First, we will define our terms. Then we will present the update rules for joins. Then we will consider the implications of those rules for each of the three cases (that is, one-to-one, one-to-many, many-to-many) in turn.

First of all, we take the term "join" to mean *natural* join specifically. Let the columns of table A be partitioned into two disjoint groups, say X and Y. Likewise, let the columns of table B be partitioned into two disjoint groups, say Y and Z. Now suppose that the columns of Y (only) are common to the two tables, so that the columns of X are "the other columns" of A and the columns of Z are "the other columns" of B. Suppose also that corresponding columns of Y (that is, columns with the same name) are defined on the same domain. Finally, regard each of X, Y, and Z as a single *composite* column. Then the expression:

```
A JOIN B
```

yields a table with columns X,Y,Z consisting of all rows <x,y,z> such that the row <x,y> appears in A and the row <y,z> appears in B. The table predicate PJ for J = A JOIN B is thus:

```
PA ( a ) AND PB ( b )
```

where for a given row j of the join, a is the "A-portion" of j (that is, the row that is derived from j by projecting away the value j.Z) and b is the "B-portion" of j (that is, the row that is derived from j by projecting away the value j.X). In other words:

"Every row in the join is such that the A-portion satisfies PA and the B-portion satisfies PB."

For example, the table predicate for the join of tables S and SP over S# follows:

"Every row <s#,sname,status,city,p#,qty> in the join is such that the row <s#,sname,status,city> satisfies the table predicate for S and the row <s#,p#,qty> satisfies the table predicate for SP."

Here, then, are the update rules for J = A JOIN B:

☐ **INSERT:** The new row j must satisfy PJ. If the A-portion of j does not appear in A, it is inserted into A.[7] If the B-portion of j does not appear in B, it is inserted into B.

Note: The specific procedural manner in which the previous rule is stated ("insert into A, then insert into B") should be understood purely as a pedagogical device;[7] it should not be taken to mean that the DBMS will execute exactly that procedure in practice. Indeed, the principle of symmetry—Number 5 from the "Preliminaries" section—implies as much, because neither A nor B has precedence over the other. Analogous remarks apply to all of the other rules below.

SSP

S#	SNAME	STATUS	CITY	P#	QTY
S1	Smith	20	London	P1	300
S1	Smith	20	London	P2	200
S1	Smith	20	London	P3	400
S1	Smith	20	London	P4	200
S1	Smith	20	London	P5	100
S1	Smith	20	London	P6	100
S2	Jones	10	Paris	P1	300
S2	Jones	10	Paris	P2	400
S3	Blake	30	Paris	P2	200
S4	Clark	20	London	P2	200
S4	Clark	20	London	P4	300
S4	Clark	20	London	P5	400

FIGURE 5. *View SSP (sample values).*

☐ **DELETE:** The A-portion of the row to be deleted is deleted from A, and the B-portion is deleted from B.

☐ **UPDATE:** The row to be updated must be such that the updated version satisfies PJ. The A-portion is deleted from A, without performing any triggered actions or table predicate checks, and the B-portion is deleted from B, again without performing any triggered actions or table predicate checks. Then, if the A-portion of the updated version of the row does not appear in A, it is inserted into A; if the B-portion does not appear in B, it is inserted into B.

Let us now examine the implications of the foregoing rules for the three different cases.

Case 1 (one-to-one): The term "one-to-one" here would more accurately be "(one-or-zero)-to-(one-or-zero)." In other words, a DBMS-known integrity constraint is in effect that guarantees that for each row of A at most one matching row is in B and vice versa. More precisely, the set of columns Y over which the join is performed must include a subset (not necessarily a proper subset) K, say, such that K is a candidate key for A and a candidate key for B.

Examples:

☐ For a first example, the reader is invited to consider the effect of these rules on the join of the suppliers table S to itself over supplier numbers (only).

☐ For a second example, suppose the suppliers-and-parts database includes another base table, SR (S#, REST), where S# identifies a supplier and REST identifies that supplier's favorite restaurant. Assume that not all suppliers in table S are represented in table SR. The reader is invited to consider the effect of these rules on the join of tables S and SR (over S#). What difference would it make if a given supplier could be represented in table SR and not in table S?

Case 2 (one-to-many): The term "one-to-many" here would more accurately be "(zero-or-one)-to-(zero-or-more)." In other words, a DBMS-known integrity constraint is in effect that guarantees that for each row of B at most one matching row is in A. Typically, what this means is that the set of "common columns" Y over which the join is performed must include a subset (not necessarily a proper subset) K, say, such that K is a candidate key for A and a matching foreign key for B.[10]

Examples:

Let view SSP be defined as S JOIN SP (this join is a foreign-key-to-matching-candidate-key join, of course). Sample values are shown in Figure 5.

☐ An attempt to insert the row <S4,Clark,20,London,P6,100> into SSP will succeed, and will have the effect of inserting the row <S4,P6,100> into table SP (thereby adding a row to the view).

☐ An attempt to insert the row <S5,Adams,30,Athens,P6,100> into SSP will succeed, and will have the effect of inserting the row <S5,P6,100> into table SP (thereby adding a row to the view).

☐ An attempt to insert the

SCP

S#	SNAME	STATUS	CITY	P#	PNAME	COLOR	WEIGHT
S1	Smith	20	London	P1	Nut	Red	12
S1	Smith	20	London	P4	Screw	Red	14
S1	Smith	20	London	P6	Cog	Red	19
S2	Jones	10	Paris	P2	Bolt	Green	17
S2	Jones	10	Paris	P5	Cam	Blue	12
S3	Blake	30	Paris	P2	Bolt	Green	17
S3	Blake	30	Paris	P5	Cam	Blue	12
S4	Clark	20	London	P1	Nut	Red	12
S4	Clark	20	London	P4	Screw	Red	14
S4	Clark	20	London	P6	Cog	Red	19

FIGURE 6. *The join of S and P over CITY.*

row <S6,Green,20,London,P6,100> into SSP will succeed, and will have the effect of inserting the row <S6,Green,20,London> into table S and the row <S6,P6,100> into table SP (thereby adding a row to the view).

Note: Suppose for the moment that it is possible for SP rows to exist without a corresponding S row. Suppose moreover that table SP already includes some rows with supplier number S6 (but not one with supplier number S6 and part number P1). Then the INSERT in the example just discussed will have the effect of inserting some additional rows into the view—namely, the join of the row <S6,Green,20,London> and those previously existing SP rows for supplier S6.

☐ An attempt to insert the row <S4,Clark,20,Athens,P6,100> into SSP will fail (why?).

☐ An attempt to insert the row <S5,Adams,30,London,P6,100> into SSP will fail (why?).

☐ An attempt to insert the row <S1,Smith,20,London,P1,400> into SSP will fail (why?).

☐ An attempt to delete the row <S3,Blake,30,Paris,P2,200> from SSP will succeed, and will have the effect of deleting the row <S3,Blake,30,Paris> from table S and the row <S3,P2,200> from table SP.

☐ An attempt to delete the row <S1,Smith,20,London,P1,300> from SSP will "succeed" (see the following note) and will have the effect of deleting the row <S1,Smith,20,London> from table S and the row <S1,P1,300> from table SP.

Note: Actually, the overall effect of this attempted DELETE will depend on the foreign key delete rule from SP.S# to S.S#. If the rule is RESTRICT, the overall operation will fail. If it is CASCADE, it will have the side-effect of deleting all other SP rows for supplier S1 as well. Other possibilities are left as an exercise for the reader.

☐ An attempt to update the SSP row <S1,Smith,20,London,P1,300> to <S1,Smith,20,London,P1,400> will succeed, and will have the effect of updating the SP row <S1,P1,300> to <S1,P1,400>.

☐ An attempt to update the SSP row <S1,Smith,20,London,P1,300> to <S1,Smith,20,Athens,P1,400> will succeed, and will have the effect of updating the S row <S1,Smith,20,London> to <S1,Smith,20,Athens> and the SP row <S1,P1,300> to <S1,P1,400>.

☐ An attempt to update the SSP row <S1,Smith,20,London,P1,300> to <S6,Smith,20,London,P1,300> will "succeed" (see the following note) and will have the effect of updating the S row <S1,Smith,20,London> to <S6,Smith,20,London> and the SP row <S1,P1,300> to <S6,P1,300>.

Note: Actually, the overall effect of this attempted update will depend on the foreign key update rule from SP.S# to S.S#. The details are left as another exercise for the reader.

Case 3 (many-to-many): The term "many-to-many" here would more accurately be "(zero-or-more)-to-(zero-or-more)." In other words, no DBMS-known integrity constraint is in effect that guarantees that we are really dealing with a Case 1 or Case 2 situation.

Examples:

Let view SCP be defined as:

```
S JOIN P
```

(join of S and P over CITY—a many-to-many join). Sample values are shown in Figure 6.

☐ Inserting the row <S7,Brown,15,Oslo,P8,Wheel,White,25> will succeed, and will have the effect of inserting the row <S7,Brown,15,Oslo> into table S and the row <P8,Wheel,White,25,Oslo> into table P (thereby adding the specified row to the view).

☐ Inserting the row <S1,Smith,20,London,P7,Washer,Red,5> will succeed, and will have the effect of inserting the row <P7,Washer,Red,5,London> into the table P (thereby adding *two* rows to the view—the row <S1,Smith,20,London,P7,Washer,Red,5> (as specified) as well as the row <S4,Clark,20,London,P7,Washer,Red,5>).

☐ Inserting the row <S6,Green,20,London,P7,Washer,Red,5> will succeed, and will have the effect of inserting the row <S6,Green,20,London> into table S and the row <P7,Washer,Red,5,London> into table P (thereby adding *six* rows to the view).

☐ Deleting the row <S1,Smith,20,London,P1,Nut,Red,12> will succeed, and will have the effect of deleting the row <S1,Smith,20,London> from table S and the row <P1,Nut,Red,12,London> from table P (thereby deleting *four* rows from the view).

Further examples are left as an exercise for the reader.

CONCLUDING REMARKS

We have applied the systematic view updating scheme first described in our June article to the question of updating restriction, projection, extension, and join views. Regarding join views in particular, we offer the following additional comments:

☐ It is well-known that intersection is a special case of natural join. To be specific, if tables A and B are type-compatible,[11] the expressions A INTERSECT B and A JOIN B are semantically identical; they should thus display identical update behavior if treated as view definitions, and so they do (exercise for the reader).

☐ It is well known also that Cartesian product is a special case of natural join. To be specific, if tables A and B have no common columns at all, the expressions A TIMES B and A JOIN B are semantically identical; they should thus display identical update behavior if treated as view definitions, and so they do (another exercise for the reader).

☐ The reader will observe that we have said nothing regarding Θ-joins. The reason is, of course, that Θ-join is not a primitive operation; in fact, it is defined as a restriction of a Cartesian product. The update rules for Θ-join can therefore be derived from the rules for restriction and Cartesian product.

We have now discussed all of the operators that are usually regarded as part of the relational algebra except for RENAME, SUMMARIZE, and DIVIDE.[7] RENAME is trivial. The SUMMARIZE operation (in general) is not information-preserving—that is, there is no unambiguous reverse mapping from the result of a SUMMARIZE back to the original table. As a consequence, views whose definition involves a SUMMARIZE are (in general) not updatable. Finally, DIVIDE (like Θ-join) is not primitive, and hence the relevant update rules can be derived from those already given (specifically those for difference, projection, and Cartesian product); the details are left as yet another exercise for the reader, but we observe that in practice it seems likely that most division views will not be updatable at all (why, exactly?).

One final observation: Throughout this article as well as the June article, we have implied, but never quite stated explicitly, that the target of a given update operation need not be a named table (that is, a base table or a view), but can instead be *any arbitrary relational expression*. By way of illustration, suppose we have a view LSSP defined as:

```
( S WHERE CITY = 'London' ) JOIN SP
```

With our usual sample values, an attempt to insert the row <S6,Green,20,London,P6,100> into this view will succeed (it will have the effect of inserting the row <S6,Green,20,London> into table S and the row <S6,P6,100> into table SP). More precisely, the first of these two rows, <S6,Green,20,London>, will be inserted, not directly into base table S, but rather into the restriction S WHERE CITY = 'London'; the rule for inserting a row into a restriction will then come into play, with the desired final effect. The point is, however, that the target of the intermediate INSERT is represented by a restriction expression, not by a named relation.

It follows that there is no reason why the syntax of the usual INSERT, DELETE, and UPDATE operations need be limited (as it is in SQL today, for example) to designating the relevant target table by means of a table name. Rather, it should be extended to permit that target to be designated by means of an arbitrary relational expression. ▮▮▮▮

The authors would like to thank Hugh Darwen, Fabian Pascal, and Paul Winsberg for their helpful comments on earlier drafts of this article.

REFERENCES

1. Codd, E. "Recent Investigations in Relational Data Base Systems." *Proceedings of the IFIP Congress.* Stockholm, Sweden, August 1974.

2. Darwen, H. "Without Check Option." In C. Date and H. Darwen, *Relational Database Writings 1989-1991.* Addison-Wesley, 1992.

3. Date, C. *An Introduction to Database Systems.* 6th edition. Addison-Wesley, 1994 (to appear in September).

4. Date, C. "Notes Toward a Reconstituted Definition of the Relational Model Version 1 (RM/V1)." In C. Date and H. Darwen, *Relational Database Writings 1989-1991.* Addison-Wesley, 1992.

5. Date, C., and D. McGoveran. "Updating Union, Intersection, and Difference Views." *Database Programming & Design,* 7(6): 46-53, June 1994.

6. McGoveran, D. "Nothing from Nothing" (in four parts). *Database Programming & Design,* 6(12): 32-41, December 1993; 7(1): 54-61, January 1994; 7(2): 42-48, February 1994; 7(3): 54-63, March 1994).

7. In SQL, however, such an attempt will fail only if the CREATE VIEW statement explicitly includes the specification WITH CHECK OPTION—that is, by default, it will not fail. See reference [2] for some criticisms of this state of affairs.

8. A default value might be NULL in SQL. We do not wish to get sidetracked into a discussion of the problems of nulls here; for present purposes, it is irrelevant whether the default is null or something else. See reference [6] for further discussion.

9. Note that this INSERT might have the side-effect of inserting the B-portion into B also, as with INSERTs on, for example, unions or intersections (see reference [5]). Analogous remarks apply to the DELETE and UPDATE rules also; for brevity, we do not bother to spell out all such possibilities here.

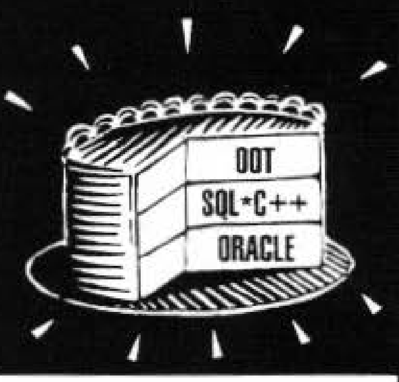
10. If this is in fact the case, and if that foreign key has "nulls not allowed," we can replace the phrase "zero or one" by "exactly one."

11. Type-compatibility is usually referred to as *union*-compatibility in the literature. We prefer our term for reasons that are beyond the scope of the present article.

C. J. Date is an independent author, lecturer, and consultant, specializing in relational database systems.

David McGoveran is president of Alternative Technologies (Boulder Creek, California), a relational database consulting firm founded in 1976.

SQL*C++™



Seamless integration of object oriented technology with ORACLE™

- Use existing ORACLE™ software
- Low cost/High success implementation method

Improve Application Speed...

- On existing or new applications
- In memory joins and queries maximize performance immediately

Powerful Class Libraries

- Allow OOT objects to be "derived"
- Beneficial to novice and expert

Proven Technology

- In use since 1992 at leading commercial and government clients

1-800-FON-MITI

FAX 310-424-9385

CIRCLE 12 ON READER SERVICE CARD