Date & McGoveran: New Views on Updating Views • ERWin and R&R

JUNE 1994

$3.95
$4.95 Canadian

DATABASE Programming & Design

Managing Distributed Databases

AN ENTERPRISE VIEW

Comparing Connectivity Standards

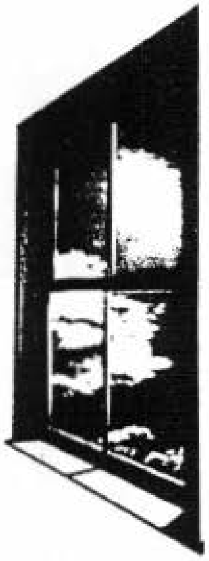
Client/Server Performance Modeling

06
0 71896 48946 0

VOL. 7 NO. 6

DATABASE Programming & Design

JUNE 1994



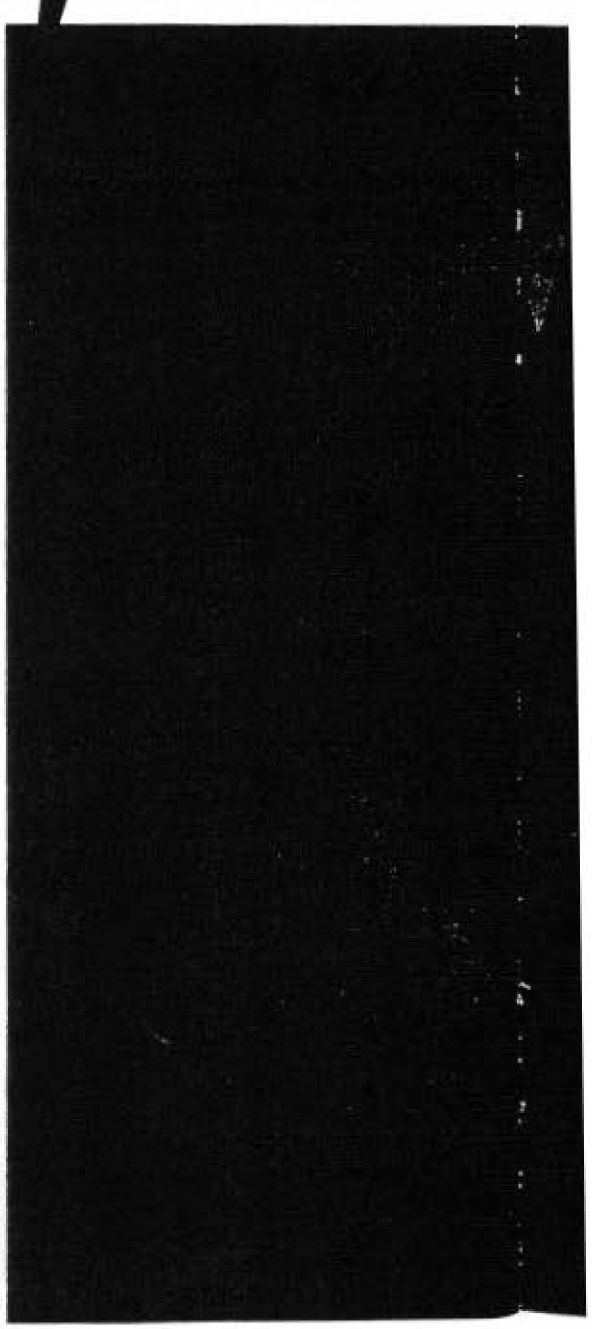
DEPARTMENTS

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 quickest 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

Though attention has been paid to some forms of view updating, it's time DBMS and SQL developers gave equal energy to other key forms

Updating Union, Intersection, and Difference Views



A VIEW IN A RELATIONAL system is a named table that is derived in some way from one or more underlying named tables—ultimately, from one or more underlying *base* tables. There is no need to go into details here on why view support is desirable: suffice it to say that it is (and always has been) an objective that, from the standpoint of data manipulation operations, views should behave as much like base tables as possible.

It is well known that view *retrieval* operations—SELECT operations, in SQL terms—are straightforward (at least theoretically, though SQL has had some nasty surprises in this area in the past). *Update* operations are a different matter, however. Indeed, the SQL standard, current SQL products, and even much of the research literature, have all treated view update in a fairly *ad hoc* manner. It is not at all unusual, for example, to find that a given DBMS will:

☐ Prohibit updates on a view that is logically updatable

☐ Permit updates on a view that is logically not updatable

☐ Implement view updates in a logically incorrect way

☐ Or (most likely in practice) do all of these things, depending on the circumstance.

The lack of a systematic approach to the problem is further illustrated by the undue emphasis that has historically been laid on restriction, projection, and join views.[4,12,13,15] Union, intersection, and difference views, which are—or should be—at least as important in practice, have received comparatively little attention.

We have recently developed a systematic and formal approach to the view updating problem. In this article, we present an informal introduction (emphasis on "informal") to that formal approach. We also illustrate the approach as it applies to union, intersection, and difference views specifically. Joins and other kinds of views will be covered in an article to appear in the August issue of *Database Programming & Design*.

ARTWORK BY STEVE KRONGARD/THE WEST © 1994

INTEGRITY CONSTRAINTS

Before we can get into the specifics of our approach, we need to lay some groundwork. The first point—and this one is **absolutely crucial**—is that ***every table has an interpretation*** or ***meaning***. In order to explain this point, we must first digress for a moment to consider the general issue of *integrity constraints*. For the purposes of this discussion, it is convenient to classify such constraints into three kinds, namely column constraints, table constraints, and database constraints,[17] as follows:

☐ A *column* constraint states that the values appearing in a specific column must be drawn from some specific domain. Consider the employees base table:

```
EMP { EMP#, ENAME, DEPT#, SALARY }
```

(see Figure 1 for a sample tabulation). The columns of this table are subject to the following column constraints:

```
e.EMP# IN EMP#_DOM
e.ENAME IN NAME_DOM
e.DEPT# IN DEPT#_DOM
e.SALARY IN US_CURRENCY_DOM
```

Here e represents an arbitrary row of the table and EMP#_DOM, NAME_DOM, and so on, are the names of the relevant domains. *Note:* Throughout this article we use a modified form of conventional SQL syntax, for reasons of simplicity and explicitness.

☐ A *table* constraint states that the rows of a specific table must satisfy some specific condition, where the condition in question refers *solely* to the table under consideration—that is, it does not refer to any other table, nor to any domain. For example, here are two table constraints for the base table EMP:

```
1. IF e.DEPT# = 'D1' THEN e.SALARY < 44K
2. IF e.EMP# = f.EMP#
              THEN e.ENAME = f.ENAME
              AND e.DEPT# = f.DEPT#
              AND e.SALARY = f.SALARY
```

The first of these constraints says that employees in department D1 must have a salary less than 44K. The second says that if two rows e and f have the same EMP# value,

Every table has an interpretation or meaning

then they must also have the same ENAME value, the same DEPT# value, and the same SALARY value—in other words, they must be the same row (this is just a longwinded way of saying that EMP# is a candidate key).

Aside: Note carefully that we talk of table constraints in general, not just base table constraints. The point is, *all* tables, base or otherwise, are subject to table constraints, as we will see later. *End of aside.*

☐ For the purposes of this article we define a database to be some user-defined (or DBA-defined) collection of named tables (base tables and/or views). A *database* constraint, then, states that the database in question must satisfy some specific condition, where the condition in question can refer to as many named tables as desired. Suppose the database containing base table EMP were extended in order to include a departments base table DEPT. Then the referential constraint from EMP to DEPT would be a database constraint (referring, as it happens, to exactly two base tables).

Here is another example of a database constraint (also referring to the same two base tables):

```
d.BUDGET > 2 * SUM ( e WHERE e.DEPT# =
                          d.DEPT#, SALARY )
```

Here d and e represent an arbitrary DEPT row and an arbitrary EMP row, respectively. The constraint says that every department has a budget that is at least twice the sum of all salaries for employees in that department.

EMP	EMP#	ENAME	DEPT#	SALARY
	E1	Lopez	D1	25K
	E2	Cheng	D1	42K
	E3	Finzi	D2	30K
	E4	Saito	D2	45K

FIGURE 1. *Base table EMP (sample values).*

Aside: Unlike column constraints and base table constraints, which can always be checked *immediately* (that is, after each individual update operation), database constraints must—at least conceptually—be *deferred* (that is, checked at end-of-transaction). In practice, there will be many cases in which database constraints also can be checked immediately, but this type of "early" checking should be regarded principally as nothing more than an optimization. *End of aside.*

TABLE PREDICATES

Now we can get back to our discussion of what tables *mean*. As we stated previously, every table—be it a base table, a view, a query result, or whatever—certainly does have an associated meaning. And, of course, users must be aware of those meanings if they are to use the database effectively (and correctly). For example, the meaning of table EMP is something like the following:

"The employee with the specified employee number (EMP#) has the specified name (ENAME), works in the specified department (DEPT#), and earns the specified salary (SALARY). Furthermore, if the department number is D1, then the salary is less than 44K. Also, no two employees have the same employee number." (This statement is not very precise, but it will serve us for the moment.)

Formally, this statement is an example of a *predicate*, or truth-valued function—a function of four arguments, in this particular case. Substituting values for the arguments is equivalent to *invoking* the function (or "instantiating" the predicate), thereby yielding an expression that evaluates to either *true* or *false*. For example, the substitution:

```
EMP# = 'E1'
ENAME = 'Lopez'
DEPT# = 'D1'
SALARY = 25K
```

yields the value *true*. By contrast, the substitution:

```
EMP# = 'E1'
ENAME = 'Abbey'
DEPT# = 'D3'
```

What is the table predicate for a derived table?

```
SALARY = 45K
```

yields the value *false*. And at any given time, of course, the table contains exactly those rows that make the predicate evaluate to *true* at that time.

It follows from the foregoing that if (for example) a row is presented as a candidate for insertion into some table, the DBMS should accept that row only if it does not cause the corresponding predicate to be violated. More generally, the predicate for a given table represents the *criterion for update acceptability* for that table—that is, it constitutes the criterion for deciding whether or not some proposed update is in fact valid (or at least plausible) for the given table.

For it to be able to decide whether or not a proposed update is acceptable for a given table, therefore, the DBMS needs to be aware of the predicate for that table. Now, it is of course not possible for the DBMS to know *exactly* what the predicate is for a given table. In the case of table EMP, for example, the DBMS has no way of knowing *a priori* that the predicate is such that the row <E1,Lopez,D1,25K> makes it true and the row <E1,Abbey,D3,45K> does not; it also has no way of knowing exactly what certain terms appearing in that predicate (such as "works in" or "earns") really mean. However, the DBMS certainly *does* know a reasonably close approximation to that predicate. To be specific, it knows that, if a given row is to be deemed acceptable, all of the following must be true:

☐ The EMP# value must be a value from the domain of employee numbers.

☐ The ENAME value must be a value from the domain of names.

☐ The DEPT# value must be a value from the domain of department numbers.

☐ The SALARY value must be a value from the domain of U.S. currency.

☐ If the DEPT# value is D1 then the salary must be less than 44K.

☐ The EMP# value is unique with respect to all such values in the table.

In other words, for a base table such as EMP, the DBMS at least knows all of the integrity constraints (column and table constraints) that have been declared for that base table. Formally, therefore, we can *define* the (DBMS-understood) "meaning" of a given base table to be the logical AND of all column constraints and table constraints that apply to that base table (and it is this meaning that the DBMS will check whenever an update is attempted on the base table in question). For example, the formal meaning of base table EMP is the following:

```
e.EMP# IN EMP#_DOM AND
e.ENAME IN NAME_DOM AND
e.DEPT# IN DEPT#_DOM AND
e.SALARY IN US_CURRENCY_DOM AND
( IF e.DEPT# = 'D1' THEN e.SALARY < 44K )
AND
( IF e.EMP# = f.EMP#
            THEN e.ENAME = f.ENAME AND
                 e.DEPT# = f.DEPT# AND
                 e.SALARY = f.SALARY )
```

We will refer to this expression—let us call it PE—as *the table predicate* for base table EMP.

Aside: Incidentally, note how our previous remarks point out once again the fundamental importance of the relational *domain* concept. The relational vendors should be doing all that is within their power to incorporate proper domain support into their DBMS products. It is worth pointing out too that "proper domain support" here does *not* mean support for the very strange construct called "domains" in the SQL standard! *End of aside.*

So much for base tables. But what about *derived* tables—in particular, what about views? What is the table predicate for a derived table? Clearly, we need a set of rules such that if the DBMS knows the table predicate(s) for the input(s) to any relational operation, it can deduce the table predicate for the output from that operation. Given such a set of rules, the DBMS will then know the table predicate for all possible tables, and will thus be able to decide the acceptability or otherwise of an arbitrary update on an arbitrary table (derived or base).

It is in fact very easy to state such a set of rules—they follow immediately from the definitions of the relational operators. For example, if A and B are any two type-compatible tables and their respective table predicates are PA and PB, then the table predicate PC for table C, where C is defined as A INTERSECT B, is obviously (PA) AND (PB); that is, a row r will appear in C if and only if it appears in both A and B—that is, if and only if PA(r) and PB(r) are both *true*. So if, for example, we define C as a view and try to insert r into that view, r must satisfy both the table predicate for A and the table predicate for B, or the INSERT will fail (see the section "Updating Intersections and Differences" later in this article for further discussion).

Here is another example: The table predicate for the table that results from the *restriction* operation:

```
T WHERE condition
```

is (PT) AND (condition), where PT is the table predicate for T. For example, the table predicate for EMP WHERE DEPT# = 'D1' is:

```
( PE ) AND ( DEPT# = 'D1' )
```

where PE is the table predicate for EMP as defined earlier.

Stating the table predicates corresponding to the other relational operators is left as an exercise for the reader.

FURTHER PRINCIPLES

Several further principles must be satisfied by any systematic view updating mechanism. Space does not permit much elaboration on these principles here, but most of them are readily understandable on intuitive grounds anyway.

1. All tables must be genuine relations—that is, duplicate rows are not permitted.

2. 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. For example,

the following two view definitions are semantically identical:

```
CREATE VIEW V AS
EMP WHERE DEPT# = 'D1' OR SALARY > 33K ;

CREATE VIEW V AS
( EMP WHERE DEPT# = 'D1' ) UNION
( EMP WHERE SALARY > 33K ) ;
```

Obviously, both of these views should be updatable. The SQL standard, however, and most of today's SQL products, adopt the *ad hoc* position that the first is updatable and the second is not.

3. It follows from the previous point that the view updatability rules must work correctly in the special case when the "view" is in fact a base table. This is because any base table B is semantically indistinguishable from a view V that is defined as B UNION B, or B INTERSECT B, or B MINUS C (if C is another base table that has no rows in common with B), or B WHERE *true*, or any of several other expressions that are identically equivalent to just B. Thus, for example, the rules for updating a union view, when applied to the view V = B UNION B, must yield exactly the same result as if the updates had been applied directly to the base table B.

4. The rules must preserve symmetry where applicable. For example, the delete rule for an intersection view V = A INTERSECT B must not arbitrarily cause a row to be deleted from A and not from B, even though such a one-sided delete would certainly have the effect of deleting the row from the view. Instead, the row must be deleted from both A and B.

5. The rules must take into account any applicable triggered actions, such as cascade DELETE. *Note:* For numerous well-documented reasons, we would prefer such triggered actions to be specified *declaratively*, rather than procedurally. However, the view updating rules *per se*, do not impose any such requirement.

6. For reasons of simplicity among others, it is desirable to regard UPDATE as shorthand for a DELETE-then-INSERT sequence (that is, just as syntactic sugar), and we will so regard it later in this article. This shorthand is acceptable *provided* it is understood that:

The rules cannot assume that the database is well designed

☐ 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. The reason is, of course, that the DELETE portion might temporarily violate the table predicate while the UPDATE overall does not. Suppose table T contains exactly 10 rows, and consider the effect of "UPDATE row *r*" on T if T's table predicate says that T must contain at least 10 rows.

☐ Triggered actions are likewise never performed "in the middle of" any given update (in fact they are done at the end, immediately prior to the table predicate checking).

☐ The shorthand requires some slight refinement (beyond the scope of this article) in the case of projection views.

We remark that treating UPDATEs as DELETEs-then-INSERTs implies that we regard UPDATEs as replacing entire rows, *not* as replacing individual values within such a row.

7. 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). For suppose, contrariwise, that there is some kind of view—say a union view—for which (say) INSERTs map to DELETEs. Then it must follow that INSERTs *on a base table* must also sometimes map to DELETEs—because (as already observed under point 3) the base table B is semantically identical to the union view V = B UNION B. An analogous argument applies to every other kind of view also (restriction, projection, intersection, and so on). The idea that an INSERT on a base table might really be a DELETE we take to be self-evidently absurd; hence, it is our position that (to repeat) INSERTs map to INSERTs and DELETEs to DELETEs.

8. In general, the rules when applied to a given view V will specify the operations to be applied to the table(s) on which V is defined. And those rules must work correctly even when those underlying tables are themselves derived tables in turn. In other words, the rules must be capable of *recursive application*.

9. The rules cannot assume that the database is well designed (for example, fully normalized). However, 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. Later in this article, we will give some examples of such "slightly surprising results."

UPDATING UNIONS

The general principles articulated in the previous section apply to all kinds of updates on all kinds of tables. In particular, they apply to updates on joins, restrictions, projections, and so on. For the remainder of this article, however, we concentrate on the question of updates on unions, intersections, and differences specifically (unions in this section, intersections and differences in the next). We will begin with a few preliminary remarks.

1. We assume we are updating a table defined by means of an expression of the form A UNION B or A INTERSECT B or A MINUS B (as appropriate), where A and B are arbitrary relational expressions (that is, they are not necessarily base tables). A and B must be type-compatible.

2. The table predicates corresponding to A and B are PA and PB, respectively.

3. Several of the view update rules refer to the possibility of *side-effects*. Now, it is well known that side-effects are usually undesirable; the point is, however, that side-effects might be unavoidable if A and B happen to be overlapping subsets of the same underlying table, as will frequently be the case with union, intersection, and difference views.

4. We limit our attention to single-row updates only, for the sake of simplicity.

Important caveat: The reader must understand that considering single-row updates only is in fact an oversimplification, 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 table EMP includes two additional employees, E8 and E9, and is subject to the constraint that 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 previously stated important caveat.

Here then is the INSERT rule for A UNION B:

☐ The new row must satisfy PA or PB or both. If it satisfies PA, it is inserted into A (note that this INSERT might have the side-effect of inserting the row into B also). If it satisfies PB, it is inserted into B, unless it was inserted into B already as a side-effect of inserting it into A.

Note: The specific procedural manner in which the foregoing rule is stated ("insert into A, then insert into B") should be understood purely as a pedagogical device; it should not be taken to mean that the DBMS will execute exactly this procedure in practice. Indeed, the principle of symmetry—number 4 from the "Further Principles" section—implies as much, because neither A nor B has precedence over the other. Analogous remarks apply to all of the rules discussed in this article.

Explanation:

☐ The new row must satisfy at least one of PA and PB because

Relational operations are always set-at-a-time

otherwise it does not qualify for inclusion in A UNION B—that is, it does not satisfy the table predicate, viz. (PA) OR (PB), for A UNION B. (As an aside, we note also that the new row must not already appear in either A or B, because otherwise we would be trying to insert a row that already exists.)

☐ If the requirements of the previous paragraph are satisfied, the new row is inserted into whichever of A or B it logically belongs to (possibly both).

Examples:

Let view UV be defined as:

```
( EMP WHERE DEPT# = 'D1' ) UNION
( EMP WHERE SALARY > 33K )
```

Figure 2 shows a sample tabulation of this view, corresponding to the sample tabulation of EMP shown in Figure 1.

☐ Let the row to be inserted be <E5,Smith,D1,30K>. This row satisfies the table predicate for EMP WHERE DEPT# = 'D1' (though not the table predicate for EMP WHERE SALARY > 33K). It is therefore inserted into EMP WHERE DEPT# = 'D1'. Because of the rules regarding INSERT on a restriction (which are fairly obvious and are not spelled out in detail here), the effect is to insert the new row into the EMP base table.

☐ Now let the row to be inserted be <E6,Jones,D1,40K>. This row satisfies the table predicate for EMP WHERE DEPT# = 'D1' *and* the table predicate for EMP WHERE SALARY > 33K. It is therefore logically inserted into both. However, inserting the row into either of the two restrictions has the side-effect of inserting it into the other restriction anyway, so there is no need to perform the second INSERT explicitly.

Now suppose EMPA and EMPB are two distinct *base* tables, EMPA representing employees in department D1 and EMPB representing employees with salary > 33K (see Figure 3); suppose view UV is defined as EMPA UNION EMPB, and consider again the two sample INSERTs previously discussed. Inserting the row <E5,Smith,D1,30K> into view UV will cause this row to be inserted into base table EMPA, presumably as required. However, inserting the row <E6,Jones,D1,40K> into view UV will cause this row to be inserted into *both* base tables! This result is logically correct, although arguably counterintuitive (it is an example of what we called a "slightly surprising result" earlier). *It is our position that such surprises can occur only if the database is badly designed.* In particular, it is our position that a design that permits the very same row to appear in—that is, to satisfy the table predicate for—two distinct base tables is by definition a bad design. This (perhaps controversial!) position will be elaborated in an article to appear in the July issue of *Database Programming & Design.*

Aside: To pave the way for that discussion, readers might care to meditate on the fact that the two base tables EMPA and EMPB already both contain the row <E2, Cheng,D1,42K>. How did this state of affairs arise? *End of aside.*

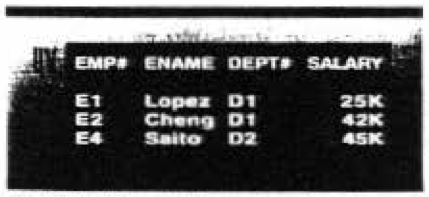
We turn now to the DELETE rule for A UNION B:

☐ If the row to be deleted appears in A, it is deleted from A (note that this DELETE might have the side-effect of deleting the row from B also). If it (still) appears in B, it is deleted from B.

Examples to illustrate this rule are left as an exercise for the reader. Note that (in general) deleting a row from A or B might cause a cascade DELETE or some other triggered action to be performed.

Finally, the UPDATE rule:

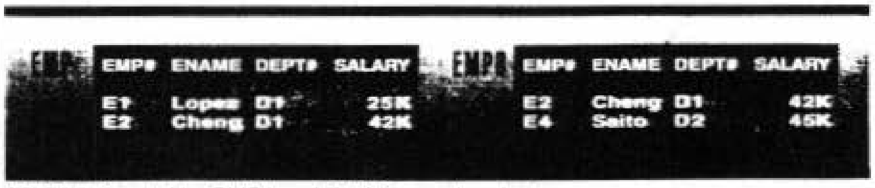
☐ The row to be updated



UV

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	25K
E2	Cheng	D1	42K
E4	Saito	D2	45K

FIGURE 2. *View UV (sample values).*



EMPA

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	25K
E2	Cheng	D1	42K

EMPB

EMP#	ENAME	DEPT#	SALARY
E2	Cheng	D1	42K
E4	Saito	D2	45K

FIGURE 3. *Base tables EMPA and EMPB (sample values).*

must be such that the updated version satisfies PA or PB or both. If the row to be updated appears in A, it is deleted from A *without* performing any triggered actions (cascade DELETE, and so on) that such a DELETE would normally cause, and likewise *without* checking the table predicate for A. Note that this DELETE might have the side-effect of deleting the row from B also. If the row (still) appears in B, it is deleted from B (again without any triggered actions or table predicate checks). Next, if the updated version of the row satisfies PA, it is inserted into A (note that this INSERT might have the side-effect of inserting the updated version into B also). Finally, if the updated version satisfies PB, it is inserted into B, unless it was inserted into B already as a side-effect of inserting it into A.

This UPDATE rule essentially consists of the DELETE rule followed by the INSERT rule, except that (as indicated) no triggered actions or table predicate checks are performed after the DELETE (any triggered actions associated with the UPDATE are conceptually performed after all deletions and insertions have been done, just prior to the table predicate checks).

It is worth pointing out that one important consequence of treating UPDATEs in this fashion is that a given UPDATE can effectively cause a row to move from one table to another. Given the database of Figure 3, for example, updating the row <E1,Lopez,D1,25K> within view UV to <E1,Lopez,D2,40K> will delete the existing row for Lopez from EMPA and insert the updated row for Lopez into EMPB.

UPDATING INTERSECTIONS AND DIFFERENCES

Here now are the rules for updating A INTERSECT B. This time we simply state the rules without further discussion (they follow the same general pattern as the union rules). Again, examples to illustrate the various cases are left as an exercise for the reader.

☐ **INSERT:** The new row must satisfy both PA and PB. If it does not currently appear in A, it is inserted into A (note that this INSERT might have the side-effect of inserting the row into B also). If it (still) does not appear in B, it is inserted into B.

☐ **DELETE:** The row to be deleted is deleted from A (note that this DELETE might have the side-effect of deleting the row from B also). If it (still) appears in B, it is deleted from B.

☐ **UPDATE:** The row to be updated must be such that the updated version satisfies both PA and PB. The row is deleted from A without performing any triggered actions or table predicate checks (note that this DELETE might have the side-effect of deleting it from B also); if it (still) appears in B, it is deleted from B, again without any triggered actions or table predicate checks. Next, if the updated version of the row does not currently appear in A, it is inserted into A (note that this INSERT might have the side-effect of inserting the row into B also). If it (still) does not appear in B, it is inserted into B.

And here are the rules for updating A MINUS B:

☐ **INSERT:** The new row must satisfy PA and not PB. 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 PA and not PB. 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.

Few DBMS products support updates on union, intersection, and difference views

CONCLUDING REMARKS

We have described a systematic approach to the view updating problem in general, and have applied this approach to the question of updating union, intersection, and difference views in particular. A critical aspect of our approach is that 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, and this observation is just as true for a view as it is for a base table. In other words, the table predicate for a given table represents the *criterion for update acceptability* for that table.

Regarding the rules for union, intersection, and difference views specifically, we think that it is worth calling out the following desirable properties of our approach explicitly:

1. Each kind of view supports all three update operations (INSERT, UPDATE, and DELETE). By contrast, other proposals allow, for example, DELETE but not INSERT on a union view,[1,4] implying that the user might be able to delete a row from a given view and then not be able to insert that very same row back into that very same view.

2. Certain important equivalences are preserved. For example, the expressions A INTERSECT B and A MINUS (A MINUS B) are semantically identical and should thus display identical update behavior if treated as view definitions, and so they do (exercise for the reader!).

3. For union and difference, INSERT and DELETE are always inverses of each other; however, for intersection they might not be (quite). For instance, if A and B are distinct base tables, inserting row r into V = A INTERSECT B might cause r to be inserted into A only (because it is already present in B); subsequently deleting r from V will now cause r to be deleted from both A and B. (On the other hand, deleting r and then reinserting it will always preserve the *status quo*.) However, it is once again our position that such an asymmetry can arise only if the database is badly designed (in particular, if the design permits the very same row to satisfy the table predicate for two distinct base tables). We will discuss this question in our article next month.

Of these properties, note that numbers 1 (support for all three update operations) and 3 (INSERT and DELETE are inverses of each other) might be regarded as two more principles that a systematic view updating mechanism really ought to satisfy if possible. Number 2 (certain equivalences preserved) is in fact a special case of the second of the principles already stated in the section "Further Principles" earlier.

Finally, we note that (of course) few DBMS products today support any kind of updates at all on union, intersection, and difference views. It is our hope that this article can serve as a guideline to be followed (a) by the vendors in adding the necessary support to their products, (b) by the SQL standards committees in their efforts to develop the next iteration of the SQL standard known informally as "SQL3." In the meantime, DBAs and application programmers who must develop workaround solutions (using, perhaps, stored or triggered procedures) to the problems caused by the current lack of support would be well advised to adhere to the principles described in this article. ▮

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

REFERENCES

1. International Organization for Standardization (ISO). *Database Language SQL*. Document ISO/IEC 9075: 1992. Also available as American National Standards Institute (ANSI) Document ANSI X3.135-1992.

2. Chamberlin, D., J. Gray, and I. Traiger. "Views, Authorization, and Locking in a Relational Data Base System." Proceedings NCC 44, Anaheim, California, AFIPS Press, May 1975.

3. Codd, E. F. *The Relational Model for Database Management: Version 2*. Addison-Wesley, 1990.

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

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

6. Date, C. "A Matter of Integrity" (in three parts), *Database Programming & Design*, 6(10): 15-18, October 1993; 6(11): 15-18, November 1993; 6(12): 19-21, December 1993.

7. Date, C. "A Contribution to the Study of Database Integrity," *Relational Database Writings 1985-1989*. Addison-Wesley, 1990.

8. Date, C. "Updating Views," *Relational Database: Selected Writings*. Addison-Wesley, 1986.

9. Date, C., and H. Darwen. *A Guide to the SQL Standard*, 3rd edition. Addison-Wesley, 1993.

10. Date, C., and C. White. *A Guide to DB2*, 4th edition. Addison-Wesley, 1992.

11. Dayal, U., and P. Bernstein. "On the Correct Translation of Update Operations on Relational Views." ACM TODS 7, Number 3, September 1982.

12. Furtado, A., and M. Casanova. "Updating Relational Views." In *Query Processing in Database Systems* (eds. W. Kim, D. Reiner, and D. Batory). Springer Verlag, 1985.

13. Goodman, N. "View Update Is Practical." *InfoDB*, 5(2), Summer 1990.

14. Keller, A. "Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins." Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Portland, Oregon, March 1985.

15. McGoveran, D., and C. Date. *A Guide to Sybase and SQL Server*. Addison-Wesley, 1992.

16. Stonebraker, M. "Implementation of Views and Integrity Constraints by Query Modification." Proceedings of the ACM SIGMOD International Conference on Management of Data, San Jose, California, May 1975.

17. This classification is slightly different (but not dramatically so) from that previously given by Date in references [6] and [7].

18. 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 discussion.

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.

CLIENT/SERVER WITHOUT TIERS!

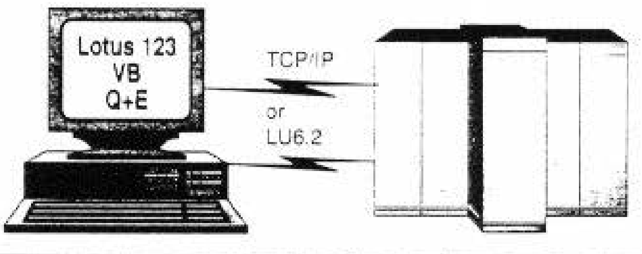
Direct Access to Mainframe Data from a User's Desktop Using TCP/IP or LU6.2

No GATEWAYS

No CICS

Direct Access to DB2 Data   Direct Access to MVS RPCs

Direct Access to the IMS Message Queue and Databases



Your desktop's applications (Lotus 123®, PowerBuilder®, Visual Basic® and more) will run seamlessly and transparently with data retrieved from your enterprise's DB2, IMS and VSAM databases.

With Shadow Direct from NEON Systems, applications connect directly to the host without any intermediate gateways. And, without going through CICS or any other network transaction manager. It's client/server without the tiers!

Shadow Direct also provides:

- Comprehensive optimization of all components for ODBC.
- Diagnostic, monitoring, and control facilities needed for transaction processing.

For more information about Shadow Direct from NEON Systems, call 713-975-3563.

NEON
Systems, Inc.

CIRCLE 16 ON READER SERVICE CARD