BY CRAIG S. MULLINS

The conventional DB2 wisdom says that creating one view per base table is the way to go. But beware—that advice could spell trouble

One View Per Base Table?

Don't Do It!

RELATIONAL DBMSs in general, and DB2 specifically, provide a very useful feature for creating a virtual table—it's called a *view*. The usual recommendation is to create one view for each base table in a DB2 application system. To my mind, however, this is a highly dubious practice. The rationale for using that method is that it insulates application programs from database changes; so all programs will be written to access views instead of base tables. While this sounds like a good idea in principle, indiscriminate view creation can be dangerous.

WHY VIEWS?

Like all relational databases, an operation on a DB2 table produces another table. A view is a virtual table—a representation of the data stored in one or more tables. It's defined using the SELECT, PROJECT, and JOIN operations. Because a view is represented internally to DB2 by SQL commands, not by stored data, views can be defined using the same SQL statements that access data in base tables. The SQL comprising the view is executed only when the view is accessed. This allows the creation of logical tables that consist of a subset of columns from a base table. When the data in the underlying base tables changes, the changes are also reflected in any view that's derived from the base table. Views can also be created based on multiple tables using joins. Figure 1 gives an example of view creation and access.

Almost any SQL that can be issued natively can be coded into a view. There are exceptions, however: A view that contains the FOR UPDATE OF clause, an ORDER BY specification, or the UNION operation can't be defined.

Views can be accessed by SQL in the same way tables are. But again, strict rules exist about the types of views that can be updated. Figure 2 lists the restrictions on view updating.

Many DBAs, consultants, and analysts recommend creating one view per base table for new DB2 applications. These views consist of one SQL statement that retrieves all of the columns in the base table. Figure 3 gives an example of a base table and the view that would be created for it. Because this type of view doesn't break any of the rules for updatability, all SQL commands can be executed against it. What is the reason for this recommendation? Increased data independence.

I will present the arguments that proponents of the view method usually make for creating one view per base table.

ONE VIEW PER DB2 TABLE

☐ *If you add a column to a table, you don't have to change the program.*

The reasoning behind this assertion is that programs can be written so that they're independent of the table columns. If a program retrieves data using SELECT * or if it inserts rows, no knowledge of new columns would be required as long as the column is added correctly.

The SELECT * statement will return all the columns in a table. If a column is added to a table after the program is coded, the program will fail to execute. This happens because the variable necessary for storing the newly retrieved column will not be coded in the program. If the program were using a view, however, no failure would occur, because the view would

ARTWORK: TAYLOR GIBSON

only have the columns as previously defined.

If the program were coded to update views instead of base tables, the INSERT statement would continue to work, as well. However, the column that was added to the base table must allow default values. The default value could be either the null value or the DB2 default when a column is defined as NOT NULL WITH DEFAULT. The INSERT to the view would continue to work even though the view doesn't contain the new column. The row would be inserted and the new column would be assigned the appropriate default value.

☐ *If you remove a column from a table, you don't have to change the application program.*

When you remove a column, the table must be dropped and recreated without the column. You could recreate views that access the table being modified by substituting a constant value in place of the deleted column. Application programs that access the views will now return the constant in place of the column that was dropped.

☐ *If you split the table into two tables, you can change the view, thereby avoiding changing the program.*

Sometimes, you must split one DB2 table into two tables. This is usually done to increase efficiency of retrieval. For example, consider a table with 10 columns. Fifty percent of the queries against the table access the first six columns. The remaining 50 percent access the other four columns and the key column. By splitting the table, you'd improve access: one table containing the first six columns and the second table containing the remaining four columns and the key column.

If the programs were using a view, the view could be recoded to be a join of the two new tables. Then the programs would not have to be changed to reflect the modification; only the view would change.

☐ *If you combine two (or more) tables into one, you can change the view, and thereby avoid changing the program.*

This is the inverse of the previous situation. If two tables are almost always joined together, then you can increase efficiency by creating a "prejoined" table. You avoid the overhead incurred by joining the two tables. A straight SELECT can now be issued against the new table instead of a join.

If the application programs are using views in this instance, you could modify the views to be subsets of the new combination table. In this way you could avoid program changes.

☐ *Views provide a layer of protection between the application and the data.*

Sometimes people just feel safer using views instead of base tables.

```
ACCT Table                      Creating the ACCOUNT MANAGER View

ACCT_NO          CHAR(6)        CREATE VIEW userid.ACCT_MGR
ACCT_DESC        VARCHAR(40)    (ACCT_NO, MANAGER, NAME) AS
ACCT_MGR         CHAR(2)        SELECT ACCT_NO,ACCT_MGR,MGR_NAME
ACCT_TYPE        CHAR(1)        FROM userid.ACCT, userid.MGR
ACCT_OPEN_DT     DATE           WHERE ACCT_MGR = MGR_NO;

MGR Table                       Using the ACCOUNT MANAGER View

MGR_NO           CHAR(2)        SELECT ACCT_NO, MANAGER, NAME
MGR_NAME         VARCHAR(50)    FROM userid.ACCT_MGR;
```

Figure 1. *Creating and using views.*

THE REBUTTAL

For every reason given for creating one view per base table, a better case can be made for not doing so. The following list explains why the reasoning behind the points made in the previous section was unsound.

☐ *If you add a column to a table, you don't have to change the program.*

If you code your application programs properly, you won't have to change them if you add a new column. Proper program coding means coding all SQL statements with column names. If column names can be supplied in an SQL statement, then the columns should always be explicitly specified in the SQL statement. This applies in particular to the INSERT and SELECT statements and is true whether you're using views or base tables.

The SELECT * statement should never be permitted in an application program. Every DB2 manual and text issues this warning—and with good reason. All DB2 objects can be dropped and recreated or altered. If a DB2 object on which a program relies is modified, then a SELECT * in that program will cease to function.

This caveat does not change because you are using views. Even views can be dropped and recreated. If the program uses SELECT * on a view and the view has changed, it will not continue to work until it's modified to reflect the changes made to the view.

It would be a mistake to think that you'll never modify a view. You might want to for several reasons. Some companies establish a policy of keeping views in line with their base tables. This means the view changes when the table changes. Others use views for security. As security rules change, so will the views.

If you eliminate the SELECT * statement, you effectively eliminate this reason for using views. An INSERT statement will work against a base table if the column names are provided in the INSERT statement. As long as the new column will allow a default value, then the program will continue to work.

☐ *If you remove a column from a table, you don't have to change the application program.*

This is patently untrue. If you remove the column from the base table, you must remove it from the view. If you don't, and a constant is added to the view, then the view can no longer be updated. Also, all queries and reports will be returning a constant instead of the old column value, which will jeopardize the system's integrity.

Users must be able to rely on the data in the database. If con-

View Type	Restriction
Views that join tables	D U I
Views that use functions	D U I
Views that use DISTINCT	D U I
Views that use GROUP BY / HAVING	D U I
Views that contain derived data using arithmetic expression	1
Views that contain constants	1
Views that eliminate columns without default value	1

Restriction Legend

D = cannot delete
V = cannot update
I = cannot insert

Figure 2. *Non-updatable view types.*

```
Base Table

CREATE TABLE userid.BASE-TABLE
(COLUMN1  CHAR(10)  NOT NULL,
 COLUMN2  DATE      NOT NULL WITH
DEFAULT,
 COLUMN3  SMALLINT,
 COLUMN4  VARCHAR(50)
) IN DATABASE db-name;

Base View

CREATE VIEW userid.BASE-VIEW
(COL1, COL2, COL3, COL4) AS
 SELECT COLUMN1, COLUMN2, COLUMN3,
  COLUMN4
 FROM   userid.BASE-TABLE;
```

Figure 3. *One view per base table.*

stants are returned on screens and reports that users rely on, confusion will ensue. Also, if the data (that is now a constant) was used in any calculations, then those values are also unreliable. These unreliable calculation results could be generated and then inserted into the database, propagating bad data.

The removal of data from a database must be analyzed in the same manner as any change. Simply returning constants is no solution; it will cause more problems than it solves.

☐ *If you split the table into two tables, you can change the view, and thereby avoid changing the program.*

If a table needs to be split into two, you must have a good reason for doing so. As mentioned earlier, this decision is usually prompted by performance considerations. To increase efficiency you must change the underlying SQL to take advantage of the tables that have been split. Queries accessing columns that are only in one of the two new tables need to be modified to access only that table.

According to the reasoning of the view supporters, no changes will be made to programs. If no changes are made, then performance will actually suffer due to these changes. Think about it. The views are now joins instead of straight SELECTs. No SQL code has changed. Every straight SELECT is now doing a join, and we all know that joins are less efficient than a straight SELECT.

It's essential to understand that a change of this magnitude requires a thorough analysis of your application code. When table column definitions change, SQL and programs change as well; it can't be avoided. A trained analyst or DBA must analyze all of the application's SQL. This includes SQL in application PLANs, QMF queries, and dynamic SQL. Queries that access columns from both of the new tables need to be made into a join. You do not want to do indiscriminate joins, however. Queries that access columns from only one of the two tables must be recoded as a straight SELECT against that table to achieve the performance gain.

Also, any programs that update the view must be changed. Remember, views that join tables can't be updated.

If after investigating all of the queries you determine that some queries will require joining the two new tables, then you can create a view to accommodate those queries. The view can even have the same name as the old table to minimize the program changes that will be required. You can even give the two tables new names. The view will be created only when it's needed. This is a much more reasonable approach to change management.

It's also worth noting that a change of this magnitude is rarely attempted after an application has been moved to production. The people who recommend using views rarely consider this fact.

☐ *If you combine two or more tables into one, you can change the view and thereby avoid changing the program.*

If you simply combine the two tables into one and change the views to be subsets of the new prejoined table without changing the SQL, you will have once again degraded performance. Most queries need to access both tables. The queries that were joins are still joins, but now they're joining the new views. Remember that the views are just subsets of one table, so these queries are joining this one table to itself. This is usually less efficient than joining the two tables as they were previously defined.

Once again, you have to do a great deal of analysis before making a change of this magnitude. You must investigate all application SQL; if you determine that queries access only one of the two old tables, then views can be defined with the same name as the old tables. You can give the new prejoined table a new name, minimizing program modification.

☐ *Views provide a layer of protection between the program and the data.*

No, they don't. If you create one view for each base table, all types of SQL can be performed on the views. Update and retrieval SQL can be performed in the same manner on the views as it could on the base tables. There is no realistic basis for this reasoning whatsoever.

GENERAL VIEW RULES

To ensure a proper, reasonable approach to view creation, let me offer three basic rules.

The usage rule: Create a view only when it achieves a specific, rational goal. Each view must have

```
ACCT Table                      Dropping the ACCOUNT MANAGER View

ACCT_NO         CHAR(6)         DROP VIEW userid.ACCT_MGR;
ACCT_DESC       VARCHAR(40)
ACCT_MGR        CHAR(2)
ACCT_TYPE       CHAR(1)
ACCT_OPEN_DT    DATE            Recreating the ACCOUNT MANAGER View
                                CREATE VIEW userid.ACCT_MGR
                                (ACCT_NO, MANAGER, NAME, INIT) AS
MGR Table                        SELECT ACCT_NO, MGR_NO, MGR_NAME,
MGR_NO          CHAR(2)                 INITIAL
MGR_NAME        VARCHAR(50)      FROM  userid.ACCT, userid.MGR
INITIAL         CHAR(1)          WHERE ACCT_MGR = MGR_NO;
```

Figure 4. *Adding a column and changing a view.*

a specific usage before it's created. That usage must also be logical. There are three types of usage that are truly logical: security, access, and data derivation.

Views created to provide security on tables will effectively create a logical table that's a subset of rows, columns, or both, from the base table. By eliminating restricted columns from the column list and providing the proper predicates in the WHERE clause, you can create views to allow access to only those portions of a table that each user is allowed to access.

Views created for access reasons should guarantee efficient access to the underlying base table by specifying indexed columns and proper join criteria. By coding views to always specify columns that are indexed in the WHERE clause, you gain efficient access. Coding join logic into a view also increases access efficiency, because the join will always be done properly. A proper join is done by coding the WHERE clause to compare the primary key of one table to the foreign key of another.

The third type of view that provides a valid usage is one that contains data-derivation formulas. An example is a view containing a column named TOTAL_SALARY, which is created by selecting SALARY+COMMISSION.

Any other usages for views should be scrupulously analyzed. Chances are, they're not good usages at all.

The proliferation avoidance rule: Do not needlessly proliferate DB2 objects. Every DB2 object that is created constitutes additional entries in the DB2 catalog. Creating needless views clutters the catalog. The larger the DB2 catalog tables become, the less efficient your entire DB2 system will be.

The synchronization rule: Keep all views logically pure by synchronizing them with the underlying base tables.

Whenever you change a base table, you should analyze all views that are dependent on that base table to determine if the change will affect them. All views should remain logically pure. The view was created for a specific reason (see the usage rule). It should therefore remain useful for that specific purpose. You can accomplish this only by ensuring that all subsequent changes that are pertinent to a specified usage are made to all views that satisfy that usage.

For example, say you have a view that satisfies an access usage. Figure 1 shows a view that satisfies a join between a table of accounts and a table of managers. If a column is added to the MGR table specifying the manager's initial, it should also be added to the ACCT_MGR view, because it's pertinent to that view's specific use: to provide information about each account's manager. Figure 4 shows the steps necessary to drop and recreate the view.

The synchronization rule requires that you institute strict change-impact-analysis procedures. Every table change should follow these procedures. Simple SQL queries can help with change-impact analysis. These queries should pinpoint QMF queries, application plans, and dynamic SQL users who could be affected by specific changes. Figure 5 lists queries that will assist in the change-impact-analysis process. Always execute these queries to determine what views might be impacted by changes to base tables.

```
To find all views dependent on the
table to be changed:

  SELECT DCREATOR, DNAME
  FROM   SYSIBM.SYSVIEWDEP
  WHERE  BCREATOR = 'Table Creator'
  AND    BNAME    = 'Table Name';

To find all QMF queries that access
the view:

  SELECT DISTINCT OWNER, NAME, TYPE
  FROM   Q.OBJECT-DATA
  WHERE  APPLDATA LIKE '%VIEW Name%';

To find all plans that are dependent
on the view:

  SELECT DNAME
  FROM   SYSIBM.SYSPLANDEP
  WHERE  BCREATOR = 'VIEW Creator'
  AND    BNAME    = 'VIEW Name';

To find all potential dynamic SQL
users:

  SELECT GRANTEE
  FROM   SYSIBM.SYSTABAUTH
  WHERE  TCREATOR = 'VIEW Creator'
  AND    TTNAME   = 'VIEW Name';
```

Figure 5. *Change-impact-analysis queries.*

By following these three rules, you'll establish a sound framework for view creation in your organization.

IN SUMMARY

There's no adequate reason for enforcing a strict rule of one view per base table for DB2 application systems. Indeed, the evidence recommends against using views in this manner. The one-view theory is rooted in the mistaken assumption that applications can be totally ignorant of underlying changes to the database. Change-impact analysis must always be done when tables are modifed. Failure to do so means your applications will perform poorly. The idea that views eliminate the need for change-impact analysis is a myth that needs to be debunked. Otherwise, you'll encounter major performance problems. ||||

Craig S. Mullins is a database and systems administrator specializing in DB2 at Mellon Bank in Pittsburgh. He is also a cofounder and vice president of ASSET Inc., a customized-software and technical consulting firm.