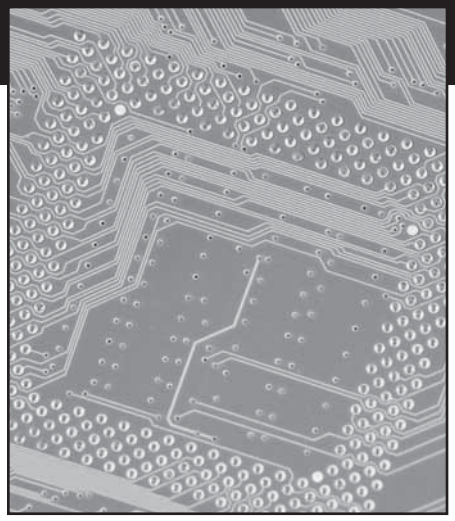

# A View Review

BY CRAIG S. MULLINS

One of the most fertile grounds for disagreement between DB2 professionals is the appropriate usage of views. The manner in which views can be utilized to provide the greatest benefit can be a very controversial issue. Some analysts promote the liberal creation and usage of views, whereas others preach a more conservative approach.

When properly implemented and managed, views can be fantastic tools that help to ease data access and simplify development. Although views are simple to create and implement, few organizations take a systematic and logical approach to view creation. And therein lies the controversy. A strategic and reasonable policy guiding the creation and maintenance of views is required to avoid a muddled and confused mish-mash of view usage. Basically, views are very useful when implemented wisely, but can be an administrative burden if implemented without planning.

## VIEW OVERVIEW

Before discussing a proper view implementation strategy, let's review the basics of views.

All SQL access to a DB2 table results in another table. This is a requirement of the relational model. A view can be considered to be a logical table. A view is a "logical" representation of data that is "physically" stored in other tables (and perhaps other views as well).

Views are defined using SQL and are represented internally to DB2 by SELECT statements, not by stored data. The SQL inside the view is executed only when the view is accessed and views can be accessed by SQL in the same way that tables are – using SQL. Certain limitations on data modification exist depending upon the type of view, though. Views that join tables, use functions, specify DISTINCT, or use GROUP BY and HAVING may not be updated, inserted to or deleted from. Additionally, inserting is prohibited for the following types of views:

- views using constants,
- views having columns with derived data in the SELECT-list, and
- views that do not contain all columns defined as NOT NULL from the tables from which they were defined.

Almost any SQL that can be issued natively can be coded into a view. There are a few exceptions, though. A view cannot be defined that contains any of the following clauses:

- FOR UPDATE OF
- ORDER BY
- OPTIMIZE FOR n ROWS

As of DB2 Version 7, one of the biggest limitations to view usage was removed, namely the ability to specify UNION and UNION ALL in the view definition. Prior to V7, UNION was not permitted in a view definition. This capability greatly expands the functionality of views and the administrative options available when creating tables. For example, it is now possible to "partition" using a view by creating several, separate tables that "look the same," and then creating a view that unions them together – in essence creating a pseudo-partitioned "table" using views.

## VIEW IMPLEMENTATION RULES

After you understand view mechanics, you should develop guidelines for view creation in order to limit administrative burden. The following rules can be used to ensure that views are created in a responsible and useful manner at your shop. These rules were developed over a number of years as a result of reviewing implementations and working with views in many different environments. There are three basic view implementation rules:

- The View Usage Rule,
- The Proliferation Avoidance Rule, and
- The View Synchronization Rule.

These rules define the parameters for efficient and useful view creation. Following them will result in a DB2 shop implementing views that are

effective, minimize resource consumption, and have a stated, long-lasting purpose.

There are likely more uses for views than are presented here, so do not needlessly worry if you do not see your favorite use for views covered in this article.

## THE VIEW USAGE RULE

The first rule is the view usage rule. Simply stated, your view creation strategy should be goal-oriented. Views should be created only when they achieve a specific, reasonable goal. Each view should have a specific application or business requirement that it fulfills before it is created. That requirement should be documented somewhere, preferably in a data dictionary or possibly as a remark in the DB2 Catalog.

Although this rule seems obvious, views are implemented at some shops without much thought as to how they will be used. This can cause the number of views that must be supported and maintained to continually expand until so many views exist that it is impossible to categorize their uses. And the time needed to maintain and administer the system increases as the number of views increase.

There are five basic uses for which views excel. These are:

- to provide row and column level security,
- to ensure efficient access paths,
- to mask complexity from the user,
- to ensure proper data derivation, and
- to rename tables and/or columns.

Let's examine each of these uses.

## SECURITY

One of the best reasons to create a view is to support data security. Views can be created that provide a subset of rows, a subset of columns, or a subset of both rows and columns from the base table.

How do views help provide row and column level security? Consider an EMPLOYEE table that contains all of the pertinent information regarding an enterprise's employees. Typically, name, address, position, age, and salary information would be contained in such a table. However, not every user will require access to all of this information. Specifically, it may become necessary to shield the salary information from most users. You can accomplish this by creating a view that does not contain the salary column and then granting most users the ability to access the view instead of the base table. The salary column will not be visible to users of the view.

Or perhaps you need to implement security at the row level. Consider a table that contains project information. Typically this would include project name, purpose, start date, and who is responsible for the project. Perhaps the security requirements of the projects within your organization deem that only the employee who is responsible for the project can access their project data. By storing the authorization ID of the responsible employee in the PROJECT table, a view can be created using the USER special register such as the one shown below:

```
CREATE VIEW       MY_PROJECTS
      (PROJ_NO, PROJ_NAME, DEPT_NO,
       PROJ_STAFF, PROJ_START_DATE,
       PROJ_END_DATE)
AS
      SELECT      PROJNO, PROJNAME, DEPTNO,
                  PRSTAFF, PRSTDATE, PRENDATE
      FROM        DSN8810.PROJ
      WHERE RESPEMP = USER;
```

The USER special register will contain the primary authorization ID of the process initiating the request. So, if user DBAPCSM issues a SELECT statement against the MY_PROJECTS view, only rows where RESPEMP is equal to DBAPCSM will be returned. This is a fast and effective way of instituting row level security.

By eliminating restricted columns from the SELECT list and providing the proper predicates in the WHERE clause, views can be created to allow access to only those portions of a table that each user is permitted to access.

## EFFICIENT ACCESS

Views can also be used to ensure optimal access paths. By coding efficient predicates in the view definition SQL, efficient access to the underlying base tables can be guaranteed. The use of stage 1 predicates, proper join criteria, and predicates on indexed columns can be coded into the view.

For example, consider the following view:

```
CREATE VIEW       EMP_DEPTS
      (EMP_NO, EMP_FIRST_NAME, EMP_MID_INIT,
       EMP_LAST_NAME, DEPT_NO, DEPT_NAME)
AS
      SELECT      E.EMPNO, E.FIRSTNME, E.MIDINIT,
                  E.LASTNAME, D.DEPTNO, D.DEPTNAME
      FROM        DSN8810.EMP E,
                  DSN8810.DEPT      D
      WHERE D.DEPTNO = E.WORKDEPT;
```

By coding the appropriate join criteria into the view definition SQL you can ensure that the correct join predicate will always be utilized. Of course, this technique becomes more useful as the SQL becomes more complex.

## COMPLEXITY

Somewhat akin to coding appropriate access into views, complex SQL can be coded into views to mask the complexity from the user. This can be extremely useful when your shop employs novice DB2 users (whether those users are programmers, analysts, managers or typical end users).

Consider the following rather complex SQL that implements relational division:

```
SELECT      DISTINCT PROJNO
FROM        DSN8810.PROJACT         P1
WHERE NOT EXISTS
            (SELECT     ACTNO
             FROM DSN8810.ACT A
             WHERE      NOT EXISTS
                  (SELECT     PROJNO
                   FROM DSN8810.PROJACT          P2
                   WHERE      P1.PROJNO = P2.PROJNO
                   AND        A.ACTNO = P2.ACTNO);
```

This query uses correlated subselects to return a list of all projects in the PROJACT table that require every activity listed in the ACT table. By coding this SQL into a view called, say ALL_ACTIVITY_PROJ, the end user will need only to issue the following simple SELECT statement instead of the more complicated query:

```
SELECT        PROJNO
FROM   ALL_ACTIVTY_PROJ;
```

Now isn't that a lot simpler?

## DERIVED DATA

Another valid usage of views is to ensure consistent derived data by creating new columns for views that are based upon arithmetic formulae. For example, creating a view that contains a column named TOTAL_COMPENSATION which is defined by selecting SALARY + COMMISSION + BONUS is a good example of using derived data in a view.

## COLUMN RENAMING

As you can tell from looking at the sample views shown in the other sections, you can rename columns in views. This is particularly useful if a table contains arcane or complicated column names. There are some prime examples of such tables in the DB2 Catalog. Consider the following view:

```
CREATE VIEW PLAN_DEPENDENCY
      (OBJECT_NAME, OBJECT_CREATOR, OBJECT_TYPE,
       PLAN_NAME, IBM_REQD)
AS
      SELECT      BNAME, BCREATOR, BTYPE,
                  DNAME, IBMREQD
      FROM        SYSIBM.SYSPLANDEP;
```

Not only have we renamed the entity from SYSPLANDEP to the more easily understood name, PLAN_DEPENDENCY, but we have also renamed each of the columns. Isn't it much easier to understand PLAN_NAME than DNAME, or OBJECT_CREATOR than BCREATOR? Views can be created on each of the DB2 Catalog tables in this manner so that your programmers will be better able to determine which columns contain the information that they require. Additionally, if other tables exist with clumsy table and/or column names, views can provide an elegant solution to renaming without having to drop and recreate anything.

Sometimes older applications were developed without sound DB2 naming conventions. I have actually seen tables where the column names are A1, A2, A3, etc. Using a view to rename those columns into something more useful would be a very good idea. The view option is worth considering because actually renaming the columns in the table would require dropping and recreating the table – with all of the change management headache that is entailed with such a change.

## THE PROLIFERATION AVOIDANCE RULE

The second rule is the proliferation avoidance rule. This rule is simple and to the point: Do not needlessly create DB2 objects that are not absolutely required.

Whenever a DB2 object is created additional entries are placed in the DB2 Catalog. Creating needless views causes what I call "catalog clutter" – that is, entries in the catalog for objects which are not needed and/or are not used.

In terms of views, for every unnecessary view that is created DB2 will potentially insert rows into 4 view-specific catalog tables (SYSVTREE, SYSVLTREE, SYSVIEWS, and SYSVIEWDEP) and 3 table-specific catalog tables (SYSTABLES, SYSTABAUTH, and SYSCOLUMNS). If uncontrolled view creation is permitted then disk growth, I/O problems, and inefficient catalog organization can result.

The proliferation avoidance rule is based on common sense. Why create something that is not needed? It just takes up space that could be used for something that is needed.

## THE VIEW SYNCHRONIZATION RULE

The third, and final view implementation rule is the view synchronization rule. The basic intention of this rule is to ensure that views are kept in sync with the base tables upon which they are based.

Whenever a change is made to a base table, all views that are dependent upon that base table should be analyzed to determine if the change impacts them. All views should remain logically pure. The view was created for a specific reason (see the View Usage Rule above). The view should therefore remain useful for that specific reason. This can only be accomplished 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 a view was created to satisfy an access usage, such as the EMP_DEPTS view previously depicted. The view was created to provide information about employees and their departments. If a column is added to the EMP table specifying the employee's social security number, it should also be added to the EMP_DEPT view if it is pertinent to that view's specific use. Of course, the column can be added to the table immediately and to the view at the earliest convenience of the development team.

The synchronization rule requires that strict change impact analysis procedures be in place. Every change to a base table should trigger the usage of these procedures. Simple SQL queries can be created to assist in the change impact analysis. These queries should pinpoint QMF queries, application plans, and dynamic SQL users that could be using views affected by the specific changes to be implemented.

View synchronization is needed to support the view usage rule. By

keeping views in sync with table changes the original purpose of the view is maintained.

## SOMEWHAT OUTDATED USES FOR VIEWS

Over the years views have been used for other purposes that made sense at the time, but have been rendered obsolete with the advent of new DB2 functionality. Two of these view usages are to simulate domain support and to implement queries that access both summary and detail information in a single row. Let me elaborate on both and tell you why these usages are outdated.

## DOMAIN SUPPORT

Most database systems do not support domains, and DB2 is no exception. Domains are an instrumental component of the relational model that were in the original relational paper published by Ted Codd in 1970 — 35 years ago! Although the purpose of this article is not to explain the concept of domains, a quick explanation is in order. A domain basically identifies the valid range of values that a column can contain. Of course, domains are more complex than this. For example, the relational model states that only columns pooled from the same domain should be able to be compared within a predicate (unless explicitly overridden).

Some of the functionality of domains can be implemented using views and the WITH CHECK OPTION clause. The WITH CHECK OPTION clause ensures the update integrity of DB2 views. This will guarantee that all data inserted or updated using the view will adhere to the view specification. For example, consider the following view:

```
CREATE VIEW       EMPLOYEE
      (EMP_NO, EMP_FIRST_NAME, EMP_MID_INIT,
       EMP_LAST_NAME, DEPT, JOB, SEX, SALARY)
AS
      SELECT      EMPNO, FIRSTNME, MIDINIT, LASTNAME,
            WORKDEPT, JOB, SEX, SALARY
      FROM  DSN8810.EMP
      WHERE       SEX IN ('M', 'F')
WITH CHECK OPTION;
```

The WITH CHECK OPTION clause, in this case, ensure that all updates made to this view can specify only the values ‘M’ or ‘F’ in the SEX column. Although this is a simplistic example, it is easy to extrapolate from this example where your organization can create views with predicates that specify code ranges using BETWEEN, patterns using LIKE, or a subselect against another table to identify the domain of a column.

When inserts or updates are done using these types of views, DB2 will evaluate the predicates to ensure that the data modification conforms to the predicates in the view. Be sure to perform adequate testing prior to implementing domains in this manner to safeguard against possible performance degradation.

Now this method of using views to simulate domains is still viable, but a better technique to provide the same functionality is available, namely check constraints. Check constraints place specific data value restrictions on the contents of a column through the specification of an expression. The expression is explicitly defined in the table DDL and is formulated in much the same way that SQL WHERE clauses are formulated. Any attempt to modify the column data (i.e. during INSERT and UPDATE processing) will cause the expression to be evaluated. If the modification conforms to the expression, the modification is permitted to continue. If not, the statement will fail with a constraint violation. This approach is simpler than creating views using the WITH CHECK option.

## SINGLE SOLUTION VIEWS

Another past usage for views was to enable solutions where views were required to solve a data access problem. Without a view, complex data access requests could be encountered that were not capable of being coded using SQL alone.

Consider the scenario where you want to report on detail information and summary information from a single table. Consider, for example, a report on column details from the DB2 Catalog. For each table, we need to provide all column details, and on each row, also report the maximum, minimum, and average column lengths for that table. Additionally, report the difference between the average column length and each individual column length. To solve this problem you could create a view. Consider the COL_LENGTH view based on SYSIBM.SYSCOLUMNS shown below:

After the view is created, the following SELECT statement can be issued joining the view to the base table, thereby providing both detail and aggregate information on each report row:

```
SELECT      TBNAME, NAME, COLNO, LENGTH,
            MAX_LENGTH, MIN_LENGTH, AVG_LENGTH,
            LENGTH - AVG_COL_LENGTH
FROM        SYSIBM.SYSCOLUMNS C,
            authid.COL_LENGTH V
WHERE       C.TBNAME = V.TABLE_NAME
ORDER BY 1, 3;
```

This works well, but with the advent of table expressions (sometimes referred to as in-line views) this usage of views is obsolete. Instead of coding the view we can take the SQL from the view and specify it directly into the SQL statement that would have called the view. Using our example above, the final SQL statement becomes:

```
SELECT      TBNAME, NAME, COLNO, LENGTH,
            MAX_LENGTH, MIN_LENGTH, AVG_LENGTH,
            LENGTH - AVG_COL_LENGTH
FROM        SYSIBM.SYSCOLUMNS C,
          (SELECT  TBNAME,
                        MAX(LENGTH) AS MAX_LENGTH,
                        MIN(LENGTH) AS MIN_LENGTH,
                        AVG(LENGTH) AS AVG_LENGTH
                 FROM  SYSIBM.SYSCOLUMNS
                 GROUP BY TBNAME
           ) AS V
WHERE       C.TBNAME = V.TBNAME
ORDER BY 1, 3;
```

So now we can use a table expression to avoid creating and maintaining a view.

## VIEW NAMING CONVENTIONS

Naming conventions for views can instigate conflict within the world of DB2 DBAs. But there is really no reason for the issue to be so contentious. Remember, a DB2 view is a logical table. It consists of rows and columns, exactly the same as a DB2 table. A DB2 view can (syntactically) be used in SQL SELECT, UPDATE, DELETE, and INSERT statements in the same way that a DB2 table can. Furthermore, a DB2 view can be used functionally the same as a DB2 table (with certain limitations on updating as outlined in this article). Therefore, it stands to reason that views should utilize the same naming conventions as are used for tables. (As an aside, the same can be said for DB2 aliases and synonyms).

End users querying views do not need to know whether they are accessing a view or a table. That is the whole purpose of views. Why then, enforce an arbitrary view-naming standard, such as putting a V in the first or last position of a view name?

DBAs and technical analysts, those individuals who have a need to differentiate between tables and views, can utilize the DB2 Catalog to determine which objects are views and which objects are tables. Most users do not care whether they are using a table, view, synonym or alias. They simply want to access the data. And, in a relational database, tables, views, synonyms, and aliases all logically appear to be identical to the end user: collections of rows and columns.

Although there are certain operations that cannot be performed on certain types of views, users who need to know this will generally be sophisticated users. For example, very few shops allow end users to update any table they want using QMF, SPUFI, or some other tool that uses dynamic SQL. Updates, deletions, and insertions (the operations which are not available to some views) are generally coded into application programs and executed in batch or via online transactions – and technicians do that work (namely, programmers). Most end users only need to query tables. Now you tell me, which name will your typical end user remember more readily when he needs to access his marketing contacts: MKT_CONTACT or VMKTCT01?

## DO NOT CREATE ONE VIEW PER BASE TABLE

Often times the ridiculous recommendation is made to create one view for each base DB2 table. This is what I call The Big View Myth. The reasoning behind The Big View Myth is the desire to insulate application programs from database changes. This insulation is purported to be achieved by mandating that all programs be written to access views instead of base tables. When a change is made to the base table, the programs do not need to be modified because they access a view — not the base table.

There is no adequate rationale for enforcing a strict rule of one view per base table for DB2 applications. In fact, the evidence supports not using views in this manner.

Although it may sound like a good idea in principle, indiscriminate view creation should be avoided. The implementation of database changes requires scrupulous analysis regardless of whether views or base tables are used by your applications. Consider the simplest type of database change – adding a column to a table. If you do not add the column to the view, no programs can access that column unless another view is created that contains that column. But if you create a new view every time you add a new column it will not take long for your environment to be swamped with views. Even more troublesome is "which view should be used by which program?" Similar arguments can be made for removing columns, renaming tables and columns, combining tables and splitting tables.

In general, if you follow good DB2/SQL programming practices, you will usually not encounter situations where the usage of views initially would have helped program/data isolation anyway. (For example, simply avoiding SELECT * in your programs is usually sufficient to insulate your programs from changes.) By dispelling The Big View Myth you will decrease the administrative burden of creating and maintaining an avalanche of base table views.

## ALWAYS SPECIFY COLUMN NAMES

When creating views, DB2 provides the option of specifying new column names for the view or defaulting to the same column names as the underlying base table(s). It is always advisable to explicitly specify view column names instead of allowing them to default, even if using the same names as the underlying base tables. This will provide for more accurate documentation.

## CODE SQL STATEMENTS IN BLOCK STYLE

All SQL within each view definition should be coded in block style. As an aside, this standard should apply not only to views but to all SQL whether embedded in a COBOL program, coded as a QMF query, or implemented using any other tool. Follow these guidelines for coding the SELECT component of your views:

- Code keywords such as SELECT, WHERE, FROM, and ORDER BY such that they stand off and always begin at the far left of a new line.
- Use parentheses where appropriate to clarify the intent of the SQL statement.
- Use indentation to show the different levels within the WHERE clause.

## MATERIALIZED QUERY TABLES ARE PHYSICAL VIEWS

We have another type of "view" at our disposal as of DB2 V8 – the Materialized Query Table, or MQT. Of course, MQTs are not exactly views, but they are close enough to warrant a quick examination in this article. In fact, MQTs are so close to being view-like, that Oracle and SQL Server refer to them as materialized views.

An MQT can be thought of as a view that has been materialized – that is, a view whose data is physically stored instead of virtually accessed when needed. Each MQT is defined as a SQL query (similar to a view), but the MQT actually stores the query results as data. Subsequent user queries that require the data can use the MQT data instead of re-accessing it from the base tables. By materializing complex queries into MQTs and then accessing the materialized results, the cost of materialization is borne only once, when the MQT is refreshed.

But there are potential drawbacks to using MQTs, in terms of data currency, resource consumption and administration. First of all, MQTs are not magic; they need to be refreshed when the data upon which they are based changes. Therefore, for most MQTs the underlying data should be relatively static. Additionally, MQTs consume disk storage. If your shop is storage-constrained you may not be able to create many MQTs. Finally, keep in mind that MQTs need to be maintained. If data in the underlying base table(s) changes, then the MQT must periodically be refreshed with that current data.

There are two methods for creating an MQT: You can create it anew starting from scratch using CREATE TABLE or you can modify an existing table into an MQT using ALTER TABLE.

MQTs are fascinating because the DB2 optimizer understands them. Your queries can continue to reference base tables, but DB2 may access an MQT instead. During access path selection, the optimizer examines your query to determine whether replacing your table(s) with an MQT can reduce query cost. The process undertaken by the DB2 optimizer to recognize MQTs and then rewrite the query to use them is called automatic query rewrite (AQR).

EXPLAIN will show whether AQR was invoked to use an MQT. If the final query plan comes from a rewritten query, the PLAN_TABLE contains the new access path using the name of the matched MQTs in the TNAME column. Also, TABLE_TYPE will be set to 'M' indicating an MQT was used.

MQTs are useful when you have to aggregate data in your queries. With an MQT, the data can be pre-aggregated and stored. Consider using MQTs for analytical and data warehousing queries.

This overview of MQTs has been brief. Be sure to read the IBM DB2 manuals to understand all of the functionality of MQTs before implementing them as a solution. They can be quite helpful if used properly, but you need to know how to properly create and manage MQTs for them to be useful.

## SYNOPSIS

DB2 views are practical and helpful when implemented in a systematic and thoughtful manner. Hopefully this article has provided you with some food for thought pertaining to how views are implemented at your shop. And if you follow the guidelines contained in this article, in the end, all that will remain is a beautiful view!


### ABOUT THE AUTHOR

Craig S. Mullins, is a data management strategist and director of Product Strategy for Embarcadero Technologies. He is the author of the industry-leading book on DB2 for z/OS, DB2 Developer's Guide (currently available in its 5th edition), as well as the only book on heterogeneous DBA practices, Database Administration: The Complete Guide to Practices and Procedures. You can contact him via his Web site at http://www.craigsmullins.com.