Craig S. Mullins

Database Performance Management

Return to Home Page

October 1994

DB2 update

Views on Views

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 seems to be a very cloudy issue. Some analysts promote the liberal creation and usage of views, whereas others preach a more conservative approach.

Views are wonderful tools that ease data access and system development when used prudently. Views are basically simple to create and implement, but unfortunately a systematic and logical approach to view creation is not usually taken, thereby causing the advantages of views to become muddled and

misunderstood. As you read this article you will find that views are very useful when implemented wisely, but can be an administrative burden if implemented without planning.

View Overview
Before proceeding with a discussion of view implementation strategy, let's review the basics of views. All operations on a DB2 table result in another table. This is a requirement of the relational model. A view can be considered to be a logical table. No physical structure is required of a view; it is a representation of data that is stored in other tables that do physically exist. Views can also be based on other views.

Views are defined using SQL and are represented internally to DB2 by SQL SELECT statements, not by stored data. The SQL comprising the view is executed only when the view is accessed and views can be accessed by SQL in the same way that tables are - accessed by 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
- 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 exceptions, however. A VIEW can not be defined that contains any of the following clauses: FOR UPDATE OF, ORDER BY, UNION or UNION ALL, and OPTIMIZE FOR n ROWS.

View Implementation Rules

Understanding the basic features of views will provide a framework around which your shop can develop rules governing view usage. It is imperative that you institute 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 presenting information on views and conversing with many different users in many different environments. There may be more uses for views than are presented here, so do not needlessly fret if you do not see your favorite use for views covered in this article-unless you blindly use base table views. There is no adequate rationale for enforcing a strict rule of one view per base table for DB2 application systems. In fact, the evidence supports not using views in this manner.

There are three basic view implementation rules:

- The View Usage Rule
- The Proliferation Avoidance Rule
- 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.

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 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.

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

1. to provide row and column level security
2. to ensure efficient access paths
3. to mask complexity from the user
4. to ensure proper data derivation
5. to provide domain support
6. to rename columns, and
7. to provide solutions which can not be accomplished without views

Let's examine each of these uses.

Security
One of the most beneficial purposes served by views is to extend the data security features of DB2. 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. This

can be done by creating a view that does not contain the salary column and granting most users the ability to access the view, instead of the base table.

Similarly, row level security may be necessary. 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 authid 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    DSN8230.PROJ
       WHERE   RESPEMP = USER
```

The USER special register will contain the primary authorization ID of the process initiating the request. So, if user TED001 issues a SELECT statement against the MY_PROJECTS view, only rows where RESPEMP is equal to TED001 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     DSN8230.EMP       E,
               DSN8230.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.

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    DSN8230.PROJACT          P1
WHERE   NOT EXISTS
        (SELECT ACTNO
         FROM    DSN8230.ACT     A
         WHERE  NOT EXISTS
              (SELECT PROJNO
               FROM    DSN8230.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, then the end user need only 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.

Domain Support

It is a sad fact of life that most relational database management systems do not support domains, and DB2 is no exception. Domains are an instrumental component of the relational model and, in fact, were in the original relational model published by Ted Codd in 1970-almost 25 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 simple definition can support. 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     DSN8230.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.

A word of caution however: 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 be safeguard against possible performance degradation.

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. A wonderful example of such tables are the DB2 Catalog tables.

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 more easy to understand that 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!

Single Solution Views

The final view usage situation that may be encountered is probably the most practical usage for views-when views are the only solution! Sometimes, a complex data access request may be encountered that can not be coded using SQL alone. However, sometimes a view can be created to implement a portion of the access. Then, the view can be queried to satisfy the remainder.

Consider the scenario where you want to report on detail information and summary information from a single table. For instance, what if you would like to report on column length information from the DB2 Catalog. For each table, 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. Try doing that in one SQL statement!

Or, you could create a view to solve the dilemma. Consider the COL_LENGTH view based on SYSIBM.SYSCOLUMNS shown below:

```
CREATE VIEW        COL_LENGTH
       (TABLE_NAME, MAX_LENGTH,
        MIN_LENGTH, AVG_LENGTH)
AS
       SELECT   TBNAME, MAX(LENGTH),
                MIN(LENGTH), AVG(LENGTH)
       FROM      SYSIBM.SYSCOLUMNS
       GROUP BY TBNAME
```

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
```

Situations such as these are a great opportunity for using views to make data access a much simpler proposition.

The Proliferation Avoidance Rule

The second rule is the proliferation avoidance rule. It is simple to state and directly to the point: do not needlessly create DB2 objects that are not

necessary.

Whenever a DB2 object is created additional entries are placed in the DB2 Catalog. Creating needless views (and, indeed any object), causes what I call catalog clutter-entries in the catalog for objects which are not needed or 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, DASD growth, I/O problems, and inefficient catalog organization may 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 will impact 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.

View Naming Conventions

Views also instigate another area of conflict within the world of DB2-that being how to name views. 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 my 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 need not know whether they are accessing a view or a table. That is the whole purpose of views. Why then, enforce an arbitrary naming standard, such as putting a V in the first or last position of a view name, on views? 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 can not 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 on-line transactions. Most end users need to query tables dynamically. 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

DB2 provides the very useful capability to create a virtual table known as a view. Often times the dubious recommendation is made to create one view for each base table in a DB2 application system. This is what I call The Big View Myth. The reasoning behind The Big View Myth revolves around the desire to insulate application programs from database changes. This insulation is purported to be achieved by mandating that all programs are 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.

Although this sounds 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. 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.

All of the examples in this article follow these guidelines.

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!

From DB2 Update (Xephon), October 1994.

Home.