Craig S. Mullins

Return to Home Page

January 2006

As published in: dbazine.com

Assuring Data Integrity in DB2 - Part 2

by Craig S. Mullins

In part 2 of this two-part article we continue our coverage of DB2 integrity constraints by offering some implementation guidelines.

Referential Integrity Guidelines

The general rule for implementing referential integrity is to use DB2's inherent features instead of coding RI with application code. DB2 usually has a more efficient means of implementing RI than the application. Also, why should a programmer code what already is available in the DBMS?

Exceptions to this rule are the subject of the subsequent guidelines in this section.

Consider Programmatic RI for Efficiency

DB2 does a referential integrity check for every row insertion. You can increase efficiency if your application does a single check of a row from the parent table and then makes multiple inserts to the child table. Of course, you should not allow any data modifications to be made outside the control of your programs if DB2 RI is not used.

Sometimes the flow of an application can dictate whether RI is more or less efficient than programmatic RI. If the application processing needs are such that the parent table is always (or usually) read before even one child is inserted, consider implementing programmatic RI instead of DB2 RI. DB2 RI would repeat the read process that the application must do anyway to satisfy its processing needs.

Of course, DB2 RI may still be preferable in both of these situations because it enforces data integrity for both planned and ad hoc updates, something that programmatic RI cannot do.

Consider Avoiding RI for Intact, Stable Data

When tables are built from an existing source system and are populated using existing data, and that source system is referentially intact, you may want to avoid using DB2 RI on those tables. This is especially so if data is propagated from the existing system and the new tables will not be modified in any other manner.

However, if the new tables will be modified DB2 RI is the best way to ensure the on-going consistency and integrity of the data.

Avoid RI for Read Only Systems

Do not use DB2 RI if tables are read only. Tables containing static data that is loaded and then never (or even rarely) modified are not good candidates for RI. The data should be analyzed and scrubbed prior to loading so that it is referentially intact. Because of the stability of the data there is no need for on-going referential constraints to be applied to the data. For data that is updated, but rarely, using application programs to enforce integrity is usually preferable to DB2 RI.

Sometimes, to scrub the data when loading, you may want to use DB2 RI. Specifying ENFORCE CONSTRAINTS for the LOAD utility can save a lot of application coding to enforce RI.

If application code is used to load the tables, base your decision for implementing RI with DB2 DDL according to the other guidelines in this chapter.

Beware of Self-Referencing Constraints

A self-referencing constraint is one in which the parent table is also the dependent table. The sample table, DSN8810.PROJ contain a self-referencing constraint specifying that the MAJPROJ column must be a valid PROJNO.

Self-referencing constraints must be defined using the DELETE CASCADE rule. Exercise caution when deleting rows from these types of tables because a single delete could cause all of the table data to be completely wiped out!

Check Constraints

Check constraints can be used to 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 (e.g. during INSERT or UPDATE processing) will cause the expression to be evaluated. If the modification conforms to the Boolean expression, the modification is permitted to continue. If not, the statement will fail with a constraint violation.

Check constraints consist of two components: a constraint name and a check condition. The same constraint name cannot be specified more than once for the same table. If a constraint name is not explicitly coded, DB2 will automatically create a unique name for the constraint derived from the name of the first column in the check condition.

The check condition defines the actual constraint logic. The check condition can be defined using any of the basic predicates (>, <, =, <>, <=, >=), as well as BETWEEN, IN, LIKE, and NULL. Furthermore, AND and OR can be used to string conditions together. However, please note the following restrictions:

- The constraint can only refer to columns in the table in which it is created. Other tables cannot be referenced in the constraint.
- Subselects, column functions, host variables, parameter markers, special registers and columns defined with field procedures cannot be specified in a check constraint.
- The NOT logical operator cannot be used.
- The first operand *must be* the name of a column contained in the table. The second operand must be either another column name or a

constant.

- If the second operand is a constant, it must be compatible with the data type of the first operand. If the second operand is a column, it must be the same data type as the first column specified.

The EMP table contains the following check constraint:

```
PHONENO  CHAR(4) CONSTRAINT NUMBER CHECK
                (PHONENO >= '0000' AND
                 PHONENO <= '9999'),
```

This constraint defines the valid range of values for the PHONENO column. The following are examples of check constraints which could be added to the EMP table:

```
CONSTRAINT CHECK_SALARY
CHECK (SALARY < 50000.00)

CONSTRAINT COMM_VS_SALARY
CHECK (SALARY > COMM)

CONSTRAINT COMM_BONUS
CHECK (COMM > 0 OR BONUS > 0)
```

The first check constraint ensures that no employee can earn a salary greater than $50,000; the second constraint ensures that an employee's salary will always be greater than his or her commission; and the third constraint ensure that each employee will have either a commission or a bonus set up.

The primary benefit of check constraints is the ability to enforce business rules directly in each database without requiring additional application logic. Once defined, the business rule is physically implemented and cannot be bypassed. Check constraints also provide the following benefits:

- Because there is no additional programming required, DBAs can implement check constraints without involving the application programming staff. However, the application programming staff should be consulted on check constraints because they may have more knowledge of the data. Additionally, the application programming staff must be informed when check constraints are implemented to avoid duplication of effort in the programs being developed.
- Check constraints provide better data integrity because a check constraint is always executed whenever the data is modified. Without a check constraint critical business rules could be bypassed during ad hoc data modification.
- Check constraints promote consistency. Because they are implemented once, in the table DDL, each constraint is always enforced. Constraints written in application logic, on the other hand, must be executed by each program that modifies the data to which the constraint applies. This can cause code duplication and inconsistent maintenance resulting in inaccurate business rule support.
- Typically check constraints coded in DDL will outperform the corresponding application code.

Check Constraint Guidelines

When using check constraints the following tips and techniques can be helpful to assure effective constraint implementation.

Beware of Semantics with Check Constraints

DB2 performs no semantic checking on constraints and defaults. It will allow the DBA to define defaults that contradict check constraints. Furthermore, DB2 will allow the DBA to define check constraints that contradict one another. Care must be taken to avoid creating this type of problem. The following are examples of contradictory constraints:

```
    CHECK (EMPNO > 10 AND EMPNO <9)
```

In this case, no value is both greater than 10 and less than 9, so nothing could ever be inserted. However, DB2 will allow this constraint to be defined.

```
    EMP_TYPE    CHAR(8) DEFAULT 'NEW'
    CHECK (EMP_TYPE IN ('TEMP', 'FULLTIME', 'CONTRACT'))
```

In this case, the default value is not one of the permitted EMP_TYPE values according to the defined constraint. No defaults would ever be inserted.

```
    CHECK (EMPNO > 10)

    CHECK (EMPNO >= 11)
```

In this case, the constraints are redundant. No logical harm is done, but both constraints will be checked, thereby impacting the performance of applications that modify the table in which the constraints exist.

Other potential semantic problems could occur:

- The parent table indicates ON DELETE SET NULL but a rule is defined on the child table stating CHECK (COL1 IS NOT NULL),
- When two constraints are defined on the same column with contradictory conditions
- When the constraint requires that the column be NULL, but the column is defined as NOT NULL

Code Constraints at the Table-Level

Although single constraints (primary keys, unique keys, foreign keys, and check constraints) can be specified at the column-level, avoid doing so. In terms of functionality, there is no difference between an integrity constraint defined at the table-level and the same constraint defined at the column-level. All constraints can be coded at the table-level; only single column constraints can be coded at the column-level. By coding all constraints at the table-level maintenance will be easier and clarity will be improved.

Code this (table-level):

```
CREATE TABLE ORDER_ITEM
 (ORDERNO          CHAR(3)        NOT NULL,
  ITEMNO           CHAR(3)        NOT NULL,
  AMOUNT_ORD       DECIMAL(7,2)   NOT NULL,
  PRIMARY KEY (ORDERNO, ITEMNO)
  FOREIGN KEY ORD_ITM (ORDERNO)
    REFERENCES ORDER ON DELETE CASCADE
)
```

Instead of this (column-level):

```
CREATE TABLE ORDER_ITEM
 (ORDERNO            CHAR(3)        NOT NULL
    REFERENCES ORDER ON DELETE CASCADE,
  ITEMNO             CHAR(3)        NOT NULL,
  AMOUNT_ORD         DECIMAL(7,2)   NOT NULL,
  PRIMARY KEY (ORDERNO, ITEMNO)
)
```

Favor Check Constraints Over Triggers

If the same data integrity rule can be achieved using a check constraint or a trigger, favor using the check constraint. Check constraints are easier to maintain and are generally more efficient than triggers.

Using DB2 Triggers for Data Integrity

DB2 triggers can be useful for enforcing complex integrity rules, maintaining redundant data across multiple tables, and ensuring proper data derivation. There are many considerations that must be addressed to properly implement triggers. For additional information on DB2 triggers, check out An Introduction to Triggers for DB2 OS/390.

Using Field, Edit, and Validation Procs for Data Integrity

Field procedures are programs that transform data on insertion and convert the data to its original format on subsequent retrieval. You can use a FIELDPROC to transform character columns, as long as the columns are 254 bytes or less in length.

No FIELDPROCs are delivered with DB2, so they must be developed by the DB2 user. They are ideal for altering the sort sequence of values.

An EDITPROC is functionally equivalent to a FIELDPROC, but it acts on an entire row instead of a column. Edit procedures are simply programs that transform data on insertion and convert the data to its original format on subsequent retrieval. Edit procedures are not supplied with DB2, so they must be developed by the user of DB2. They are ideal for implementing data compression routines.

A VALIDPROC receives a row and returns a value indicating whether LOAD, INSERT, UPDATE, or DELETE processing should proceed. A validation procedure is similar to an edit procedure but it cannot perform data transformation; it simply assesses the validity of the data.

A typical use for a VALIDPROC is to ensure valid domain values. For example, to enforce a Boolean domain, you could write a validation procedure to ensure that a certain portion of a row contains only T or F. In this way it is similar to a check constraint, but a VALIDPROC can apply to an entire row, as opposed to a single column.

Summary

DB2 offers a plethora of options for ensuring data integrity. Be sure to take care to use the appropriate options as you design your DB2 databases. Failing to build data integrity constraints into your database design almost certainly will result in invalid data in your tables.

From DBAzine, January 2006.

Home.