Using Check Constraints to Simulate Domains
by Craig S. Mullins

DB2 has provided table check constraints for a number of releases now, but many organizations have yet to capitalize on their functionality. Check constraints enable enhanced data integrity without requiring procedural logic (such as in stored procedures and triggers). Let's examine the basics of table check constraints.

A constraint is basically a restriction placed upon the data values that can be stored in a column or columns of a table. Of course, most RDBMS products provide several different types of constraints, such as referential constraints (to define primary and foreign keys) and unique constraints (to prohibit duplicates).

Check constraints place specific data value restrictions on the contents of a column through the specification of a Boolean 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 Boolean expression, the modification is permitted to continue. If not, the statement will fail with a constraint violation.

This functionality is great for simulating the relational concept of a domain. A domain is basically the set of valid values that a column or data type can take on. Check constraints only simulate domains, though, because there are other features provided by domains that are not provided by check constraints. One such feature is that columns pooled from separate domains must not be compared or operated on by expressions that require the same type of data for both operands. For domains to truly be supported the DBMS should support both check constraints and user-defined data types with strong type checking. This prohibits allowing ridiculous operations, such as comparing IQ to shoe size or adding Australian dollars to Euros.

Forming Check Constraints

Check constraints are written using recognizable SQL syntax. This makes them easy to implement for anyone who has even a passing familiarity with SQL. The check constraint consists of two components: a constraint name and a check condition.

The constraint name is an SQL identifier and is used to reference or identify the constraint. The same constraint name cannot be specified more than once for the same table. If a constraint name is not explicitly coded, DB2 will create a unique name automatically for the constraint.

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.

There are, however, restrictions on how check constraints are formulated. Some of these restrictions include:

- Limitations on the entire length of the check condition.
- Other tables may not be accessed in the check condition.
- Only a limited subset of SQL operations are permitted (for example subselects and column functions are prohibited in a check constraint).
- One of the operands (usually the first) of the check constraint must be the name of a column contained in the table for which the constraint is defined.
- The other operand (usually the second) must be either another column name in the same table or a constant value.
- 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.

Check Constraint Examples

Check constraints enable the DBA or database designer to specify more robust data integrity rules directly into the database. Consider the following example:

```
CREATE TABLE EMP
    (EMPNO              INTEGER
         CONSTRAINT CHECK_EMPNO
         CHECK (EMPNO BETWEEN 100 and 25000),
     EMP_ADDRESS        VARCHAR(70),
     EMP_TYPE           CHAR(8)
         CHECK (EMP_TYPE IN ('TEMP',  'FULLTIME', 'CONTRACT')),
     EMP_DEPT           CHAR(3)             NOT NULL WITH DEFAULT,
     SALARY             DECIMAL(7,2)        NOT NULL
         CONSTRAINT CHECK_SALARY
         CHECK (SALARY < 50000.00),
     COMMISSION         DECIMAL(7,2),
     BONUS              DECIMAL(7,2)
     );
```

The CREATE statement for the EMP table contains three different check constraints:

1. The name of the first check constraint for the EMP table is CHECK_EMPNO. It is defined on the EMPNO column. The constraint ensures that the EMPNO column can contain values that range from 100 to 25000 (instead of the domain of all valid integers).
2. The second check constraint for this table is on the EMP_TYPE column. This is an example of an unnamed constraint. Though this is possible, it is not recommended. It is best to always provide an explicit constraint name in order to ease identification and administration. This specific constraint restricts the values that can be placed into EMP_TYPE as: 'TEMP', 'FULLTIME', and 'CONTRACT'; no other values would be accepted.

3. The last check constraint on this table is named CHECK_SALARY. It effectively ensures that no employee can be entered with a salary of more than $50,000. (Now who would want to work there?)

Column vs. Table Level Constraints

The first check constraint example we reviewed showed a column-level check constraint. However, check constraints also may be coded at the table-level. A column-level check constraint is defined in the DDL immediately after the column. Appropriately enough, a table-level check constraint is defined after all of the columns of the table have already been defined.

It is quite common for business rules to require access to multiple columns within a single table. When this situation occurs, it is wise to code the business rule into a check constraint at the table-level, instead of at the column level. Of course, any column-level check constraint can also be defined at the table-level, as well. 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.

Let's augment our sample table DDL to add two table-level check constraints:

```
CREATE TABLE EMP
    (EMPNO              INTEGER
         CONSTRAINT CHECK_EMPNO
         CHECK (EMPNO BETWEEN 100 and 25000),
     EMP_ADDRESS    VARCHAR(70),
     EMP_TYPE       CHAR(8)
         CHECK (EMP_TYPE IN ('TEMP',   'FULLTIME', 'CONTRACT')),
     EMP_DEPT       CHAR(3)             NOT NULL WITH DEFAULT,
     SALARY         DECIMAL(7,2)        NOT NULL
         CONSTRAINT CHECK_SALARY
         CHECK (SALARY < 50000.00),
     COMMISSION     DECIMAL(7,2),
     BONUS          DECIMAL(7,2),
         CONSTRAINT COMM_VS_SALARY
         CHECK (SALARY > COMMISSION),
         CONSTRAINT COMM_BONUS
         CHECK (COMMISSION>0 OR BONUS > 0),
     );
```

The CREATE statement for the EMP table has been modified to contain two table-level check constraints having the following ramifications:

1. The name of the first table-level check constraint for the EMP table is COMM_VS_SALARY. This constraint will ensure that no employee can earn more commission than salary.
2. The second table-level check constraint is named COMM_BONUS. This constraint will ensure that every employee either earns a commission or a bonus (or possibly, both).

Check Constraint Benefits

So what are the benefits of check constraints? The primary benefit 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. This effectively minimizes the amount of code that must be written by the programming staff. With the significant application backlog within most organizations, this can be the most crucial reason to utilize check constraints.
- Check constraints provide better data integrity. As check constraints are always executed whenever the data in the column upon which they are defined is to be modified, the business rule is not bypassed during ad hoc processing and dynamic SQL. When business rules are enforced using application programming logic instead, the rules can not be checked during ad hoc processes.
- 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 within each program that modifies any 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.

The overall impact of check constraints will be to increase application development productivity while at the same time promoting higher data integrity.

Check Constraints, NULLs, and Defaults

An additional consideration for check constraints is the relational NULL. Any nullable column also defined with a check constraint can be set to null. When the column is set to null, the check constraint evaluates to unknown. Because null indicates the lack of a value, the presence of a null will not violate the check constraint.

Additionally, DB2 provides the ability to specify defaults for table columns – both system-defined defaults (pre-defined and automatically set by the DBMS) and user-

defined defaults. When a row is inserted or loaded into the table and no value is specified for the column, the column will be set to the value that has been identified in the column default specification. For example, we could define a default for the EMP_TYPE column of our sample EMP table as follows:

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

If a row is inserted without specifying an EMP_TYPE, the column will default to the value, 'FULLTIME'.

A problem can arise when using defaults with check constraints. Most DBMS products do not perform semantic checking on constraints and defaults. The DBMS, therefore, will allow the DBA to define defaults that contradict check constraints. Furthermore, it is possible to define check constraints that contradict one another. Care must be taken to avoid creating this type of problem.

Examples of contradictory constraints are depicted below:

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.

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 if the constraints contradicts a referential integrity DELETE or UPDATE rule, if two constraints are defined on the same column with contradictory conditions, or if the constraint requires that the column be NULL, but the column is defined as NOT NULL.

Other Potential Hazards

Take care when using the LOAD utility on a table with check constraints defined to it. By specifying the ENFORCE NO parameter you can permit DB2 to load data that does not conform to the check constraints (as well as the referential constraints). Although this eases the load process by enabling DB2 to bypass constraint checking, it will place the table space into a check pending state. You can run CHECK DATA to clear this state (or

force the check pending off by using START with the FORCE option or the REPAIR utility). If you do not run CHECK DATA, constraint violations may occur causing dirty data.

Summary

Check constraints provide a very powerful vehicle for supporting business rules in the database. They can be used to simulate relational domains. Because check constraints are non-bypassable, they provide better data integrity than corresponding logic programmed into the application. It is a wise course of action to use check constraints in your database designs to support data integrity, domains, and business rules in all of your relational database applications.