December 1998

SQL Server update

Using Constraints in SQL Server

By Craig S. Mullins

SQL Server provides the ability to use DDL to code constraints within the database. This enables business rules to be enforced by the database instead of via application code. Through the judicious use of constraints, application and SQL coding can be minimized and data integrity can be maximized.

Constraints, in SQL Server, can be used to:

- enforce the range of data values that can be stored in a column (check constraints)
- enforce the uniqueness of a column or group of columns within a table (unique / primary key constraints)
- enforce referential integrity (primary key and foreign key constraints)

SQL Server also provides rules, a type of integrity check that is similar to, yet distinct from check constraints. This article will focus on check constraints and rules, when to use them, and options for their use. Coverage of referential integrity and the other constraints is beyond the scope of this article.

Check Constraints
A check constraint is a mechanism for allowing predicates to be defined on a column. The predicate is attached to the column as DDL and performs automatic edit checking of values as they are presented for insert or update to the table.

Check constraints consist of two parts:

- **name** — every constraint must have a name. Failure to explicitly specify a name causes SQL Server to automatically generate a unique name for the constraint.
- **predicate** — the actual conditions of the edit check are coded as a typical SQL predicate (without the "where" keyword)

Check constraints can be coded at the column or table level. Let's examine column-level constraints first. An example of a check constraint is depicted in the following SQL:

```
create table employee
(emp_id      int                       not null,
ssno         char(9)                   not null,
emp_name     varchar(50)               not null,
salary       numeric(12,2)             not null
             constraint salary_cons
             check (salary < 50000.00),
comm         numeric(12,2)             null,
bonus           numeric(9,2)                    null
)
```

The check constraint, named salary_cons, checks to make sure that the value of the salary column is always less than $50,000.00. If the value to be inserted or updated is 50,000 or greater the modification will fail. Obviously, a company with this type of constraint in its employee table would be a poor company for a SQL Server expert to work for!

It is wise to always explicitly assign names to each and every constraint because the constraint name that SQL Server generates can be difficult to administer later. Of course, the name of an automatically generated constraint can be found by issuing the system procedure: sp_helpconstraint.

Check constraints must provide values or a column name within the same table for the values to be checked against. Unfortunately, check constraints can not be defined as a select from another table.

In addition to column-level check constraints, it is possible to specify check constraints at the table level. Instead of being attached to a single column, the constraint is attached to the entire table.

It is usually sufficient to code a check constraint at the column level. However, there are situations where table-level check constraints are required. Any time two columns of the table need to be specified in the constraint, a table-level check constraint is required.

The following example depicts the same table as above adding a table-level check constraint to ensure that an employee's bonus is less than or equal to his commission:

```
create table employee
(emp_id      int                    not null,
ssno         char(9)                not null,
emp_name     varchar(50)            not null,
salary       numeric(12,2)          not null
             constraint salary_cons
             check (salary < 50000.00),
comm         numeric(12,2)          null,
bonus        numeric(9,2)           null,
             constraint do_not_pay_em
             check (bonus <= comm)
)
```

Creating Rules and Defaults
In addition to check constraints Microsoft provides rules. Rules are free-standing database objects that can be used to enforce data integrity. This is desirable because it promotes **reusability**.

Although rules are similar to check constraints, they are different because they are "free-standing" database objects; meaning they stand by themselves outside the scope of any other object. Check constraints always must be defined within the scope of a table.

Rules define the parameters for data validation. Before a rule is applied to any column, though, it must first be bound to that column using the sp_bindrule system procedure. Once bound, similar to a check constraint, whenever data is inserted or updated, the rule is checked to ensure that the data modification complies with the rule.

Both columns and user-defined data types can have rules defined for them. Rules are implemented as:

- a list of valid or invalid values is implemented using "in"
- a range of valid or invalid values is implemented using "between"
- an edit picture is implemented using "like"
- predicate conformance is implemented using "<, >, <=, >=, =, or <>"

Similar to check constraints, rules can not be defined as selects from other tables. Rules are created using the **create rule** DDL statement. It accepts the following parameters:

- **name** — a unique name for the rule object being created
- **expression** — an SQL predicate defining the rule to be implemented. The expression consists of two components:
    - ***place_holder*** — a variable that is referenced only within the rule. The name of the place_holder variable must be preceded with the '@' character. Example: @state
    - ***remainder of the expression*** — the definitional component of the rule that can contain:
        - a list of valid or invalid values
        - a range of valid or invalid values
        - an edit picture
        - predicate conformance

Two examples of rules follow:

```
create rule state_rule as
@state in ("IL", "WI", "IN", "IA")

create rule grade_rule as
@grade like "[A-F] [ +-]"
```

The first example of a rule is defined using a list of valid values; the second uses an edit picture to define the valid values. The edit picture shows that the valid values for a grade can be the letters A through F with a space, a plus sign, or a minus sign appended to the letter.

After the rule has been created, it can be bound to columns and user-defined data types. This is accomplished using the sp_bindrule system procedure. It accepts the following parameters:

- **rule_name** — the name of the rule to be bound to the column or user-defined data type
- **object_name** — the name of the column or user-defined data type to which the rule is being bound

Keep these rules of thumb in mind before binding a rule to a column:

- A column can have one and only one rule assigned to it. SQL Server **will** allow a rule to be bound to a column that already has a rule defined. In this case, the last rule bound to the column takes precedence.
- A rule can be bound to many different columns (and user-defined data types).
- A column can have both a rule and a column-level check constraint assigned to it. If the rule and the check constraint conflict, then you may have trouble!
- Rules are applied whenever data values are inserted or updated.

Rules can be removed from columns or user-defined data types when the rule is no longer required. This can be accomplished using the sp_unbindrule system procedure. It accepts the following parameters:

- **object_name** — the name of the column or user-defined data type from which the rule is being removed (unbound)
- **futureonly** — futureonly is applicable to rules bound to user-defined data types only. When the futureonly option is specified, it causes the rule to be in effect on new columns only; current data that does not conform to the rule is maintained as is. The rule will not be inherited by columns that are currently defined using the user-defined data type.

Note that if the name of the rule is not provided, issuing the sp_unbindrule will remove any rule from the named object.

Which Is Checked First?
Since columns and user-defined data types can have both rules and defaults, the question is raised "which does SQL Server apply first, the rule or the default?"

- When a row is inserted, SQL Server will first check for defaults on columns having no value supplied, and then check the data against any rules
- When a row is updated, SQL Server will simply check the data against the rule.

Rules and defaults defined directly on columns in DDL over-ride any rule and/or default bound to a column's user-defined data type.

Rules vs. Constraints
Both rules and constraints implement data integrity for column values in SQL Server. So, when should one be used over the other? Well, there are pros

and cons for each.

Of the two methods, rules are more flexible. Rules are created as free-standing database objects and can be bound to columns and user-defined data types. Check constraints, on the other hand, are specified in the table DDL. They are useful when a constraint exists between two columns of the same table.

General rules of thumb:

- Favor the use of rules over check constraints if reuse is a concern. Because a rule is created once and then bound to each column to which it applies you can be sure that the same checking is done for each column to which the rule is bound.
- Favor the use of check constraints if conformance to the ANSI SQL standard is important to you.
- Favor the use of check constraints if you operate in a heterogeneous environment using multiple DBMS products in addition to SQL Server. Oracle, DB2, and Informix all support check constraints, but not rules. (Note: Sybase Adaptive Server also supports rules.)
- You must use check constraints instead of rules when a comparison is required between two columns of the same table. For example, if an employee's bonus must always be less than a percentage of his salary, the following check constraint would be appropriate:

  ```
  check (bonus < salary * .10)
  ```

Finally, it is possible for both a rule and a check constraint to be defined for a single column. If this occurs, be sure that the two are compatible. This is usually an undesirable situation (why have two different restrictions on one column?). Case in point, consider a rule and a check constraint applied to a state code:

- The check constraint specifies an in list of "CA", "IL", and "PA"
- The rule specifies an in list of "CA", "AZ", and "TX"

Well, in this case, only "CA" can be placed in the state code column. What would be worse is if the two lists contain mutually exclusive values. Then nothing could ever be inserted into the table. All application inserts and updates would fail!

Additional Semantic Concerns

SQL Server performs no semantic checking on constraints and defaults. It will allow the DBA to define defaults that contradict check constraints. Furthermore, SQL Server will allow the DBA to define check constraints that contradict one another. Care must be taken to avoid creating this type of problem. Examples of contradictory constraints follow:

```
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.

```
create default type_dflt as "NEW"
create table name as
   .
   .
   .
emp_type       char (8)         'NEW'
  check (emp_type in ('TEMP',   'FULLTIME', 'CONTRACT')),
   .
   .
   .
sp_bindefault "type_dflt", "owner.name"
```

In this case, the default value is not one of the permitted values according to the defined check constraint. The default value would ever be inserted, it would fail with a check constraint violation.

```
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.

What is a Domain?
According to Chris Date: "A domain is the set of all possible data values of some particular type." Domains can be partially implemented in SQL Server using a combination of user-defined data types, rules, and defaults.

SQL Server's domain support is only partial because it does not support the following domain characteristics:

- user-defined comparison operators
- limiting comparison operators by domain
- checking to ensure that two columns to be compared are pooled from the same (or compatible) domains

Suppose that you wish to define a domain for product codes to be stored in a SQL Server database. All product codes conform to the following standards:

- product codes are six bytes long
- a product code must begin with an alphabetic character
- the second byte must be numeric (but can not be 0), the next three bytes can be anything, and the last byte must be either "@" or "#"

- if a product code is unknown it should default to null (unless it is the primary key or a part of a primary key)

To implement a domain for the product code take the following steps:

- Create a user defined data type, say prodcode
  ```
  sp_addtype prodcode, "char(6)", "null"
  ```
- Create a rule, say prodcode_rule
  ```
  create rule prodcode_rule as
  @prodcode like "[A-Z][1-9]___[#,@]"
  ```
- Create a default, say prodcode_deflt
  ```
  create default prodcode_deflt as NULL
  ```
- Create all columns containing product code information specifying the "prodcode" user-defined data type. If the column participates in a primary key, specify the "not null" property directly in the table to override the property in the user-defined data type. Bind the prodcode_rule and the prodcode_deflt to all columns containing product codes.
- Vóila! — a simulated domain has been created.

Check Constraint Benefits
So what are the benefits of check constraints and rules? 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 can not be bypassed. Check constraints and rules also provide the following benefits:

- Because there is no additional programming required, DBAs can implement check constraints and rules 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 and rules provide better data integrity. Since they 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 and rules promote consistency. Because they are implemented once, in the table DDL, each constraint is always enforced. Constraints written in an application program, on the other hand, must be executed by each program that modifies the data to which the constraint applies. Usually the constraint must be coded into many programs. This can cause code duplication and inconsistent maintenance resulting in inaccurate business rule support. Rules further promote consistency because one rule can be coded that is bound to multiple columns.
- Typically check constraints and rules will outperform the corresponding application code because it is performed by the DBMS.

The overall impact of check constraints will be to increase application development productivity.

Synopsis
Check constraints and rules provide a very powerful vehicle for supporting business rules in SQL Server databases. They are non-bypassable and therefore provide better data integrity than corresponding logic programmed into application programs. It is a wise course of action to use check constraints and/or rules in all new SQL Server applications and to retrofit old applications with check constraints when the tables can be modified in an ad hoc manner.

From SQL Server Update (Xephon) December 1998.

Home.