November 1994

Getting Integrity in SYBASE SQL Server 10

by Craig S. Mullins

User-defined integrity is a component of the relational model that has been neglected for too long. Fortunately, several DBMS vendors have begun to implement more robust user-defined integrity into their products. Foremost among these vendors is Sybase, Inc. (Emeryville, CA) with their SQL Server 10 product offering.

Defining User-Defined Integrity

Most data professionals are aware that the relational model provides basic integrity features to support both referential integrity and entity integrity. However, the concept of user-defined integrity is also inherent to the relational model. When user-defined integrity is supported, the RDBMS can automatically manage the particular values that are stored within the database.

User-defined integrity constraints go far beyond simple data type checking and referential value checking. Values can be excluded from a specific column or columns based upon business requirements. In the absence of user-defined integrity support, this type of functionality is typically performed by an application program. A systematic and non-bypassable method of integrity checking without the need to write code provides an obvious benefit in terms of reduced development time.

Types of User-Defined Integrity

SYBASE SQL Server 10 provides user-defined integrity in several different ways, each of which will be introduced in this article. The following features support user-defined integrity:

- Check Constraints
- Rules
- Unique Constraints
- User-Defined Data Types
- Defaults
- Triggers

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 supplied values. Each check constraints is performed whenever data is inserted or updated. Check constraints can be coded at the column or table level.

Let’s examine column-level constraints first. Column-level constraints consist of a name and the actual predicate. Refer to Figure 1.

Figure 1. Column-Level Check Constraint

```
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)
```

Every constraint must have a name. Failure to explicitly specify a name causes SQL Server to automatically generate a unique name for the constraint. It is wise to always explicitly assign names to each check constraint because the constraint name that SQL Server generates can be difficult to administer later. The name of the constraint in the depicted example in **salary_cons**. The predicate portion defines the actual conditions of the edit check and is coded as a typical SQL where clause (without the actual "where" keyword, of course).

Unfortunately, check constraints can not be defined as a select from another table. This limits their overall benefit. However, check constraints, even in this limited form, are superior to coding the condition into each and every application program that updates the column or columns in question.

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 were table-level check constraints are required. Any time two columns of the same table need to be specified in the constraint, a table-level check constraint is required.

The sample shown in Figure 2 depicts a table-level check constraint to ensure that an employee’s bonus is less than or equal to his commission. It would have been impossible to code this particular constraint at the column level because it accesses two columns instead of one column and one constant as the previous example showed.

Figure 2. Table-Level Check Constraint

```
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,
    comm            numeric(12,2)
null,
    bonus           numeric(9,2)
null)
          constraint
do_not_payem
          check (bonus <= comm)
)
```

User-defined messages can be attached to both column-level and table-level constraints. Consider the check constraint depicted in Figure 2 for the salary column. The message "salary too high" can be assigned to the constraint by adding a message and binding it to the constraint as follows:

```
sp_addmessage 20005, "Salary
Too High"

sp_bindmsg do_not_payem,
20005
```

Messages are always assigned a number greater than 20,000 because the first 19,999 message numbers are reserved for SQL Server use. The message text can be up to 255 characters long.

Finally, information on check constraints can be retrieved from the system using the sp_helpconstraint system procedure. By passing a table name as a parameter to sp_helpconstraint, a list of all constraints defined for the given table is displayed. This procedure is new as of System 10.

Rules

Rules are similar to check constraints, but rules are "free-standing" database objects. They are created using the **create rule** DDL statement and exist independently of any table or column. Like check constraints, rules can be used to define data validation. Once created, a rule can be bound to a table column. Thereafter, whenever data is inserted or updated, the rule is checked to ensure that the data modification complies with the rule.

The advantage of creating "free-standing" rules instead of using check constraints is to enhance reusability. Rules are reusable, check constraints are not! A free-standing rule can be created and applied as follows:

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

exec sp_bindrule "state_rule",
"authors.state"
```

The first statement creates the rule; the second binds it to a specific column in a specific table. The same rule can be bound to as many different columns in as many different tables as is desired. 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**, however, 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 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.
- A rule can be removed from a column when it is no longer required. This can be accomplished using the sp_unbindrule system procedure.

Check Constraints or Rules?

The second bullet in the previous list introduces an interesting problem. When should check constraints be used? When should rules be used? And what if both are used on the same column?

Check constraints were added to SQL Server 10 to support the ANSI SQL standard. There is no concept of a rule currently in the ANSI standard. SQL Server has featured rules for many releases. Both integrity features implement the same basic function: they place restrictions on the data values that can be stored in a column.

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.

In general, the following rules of thumb should be followed:

- Favor the use of rules over check constraints when either would suffice and portability is not essential. Rules are reusable and more flexible.
- Favor the use of check constraints if conformance to the ANSI SQL standard is required or if applications are to be ported from environment to environment or from DBMS to DBMS. Check constraints are supported by many different DBMS products; rules are not.
- Favor the use of check constraints 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)
```

This can not be achieved using a rule. Rules must always reference one variable (column) and one constant.

- 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. For example, avoid the following scenario:

  ```
  check (state in ('IL', 'PA',
  'FL'))

  create rule state_rule as
     @state in ('GA', 'CA',
  'IL')

  sp_bindrule state_rule,
  "table.state"
  ```

  In this case, only rows specifying Illinois (IL) as the state could ever be inserted to the specified table!

Unique Constraints

SQL Server 10 also supports unique constraints. This type of constraint is applied to a column or columns to ensure that duplicate value can not be stored.

SQL Server enforces uniqueness by automatically creating a unique index on the specified column(s). Therefore, unique constraints can also specify the type of index to be generated: clustered or nonclustered. Nonclustered is the default. A unique constraint will allow one null to be stored in the column(s).

Although the relational model forbids duplicate rows, most relational DBMS products allow duplicates to be stored by default. Unique constraints enable the database designer to force the DBMS to follow this relational tenet.

User-Defined Data Types

All RDBMS products provide basic system data types such as integer, character, and decimal. SQL Server, however, enables users to define additional data types called user-defined data types. User-defined data types are based upon system-defined data types but can provide additional constraints on the data content. For example, the user-defined data type can provide a precision, scale, and/or length attribute as well as a column property (i.e., **null** or **not null**). User-defined data types, once created in the database, become fundamental data types; usable by any table in that database just like a system-defined data type. Rules and defaults can be bound to user-defined data types.

An example of a user-defined data type definition follows:

```
sp_addtype proper_name,
"char(40)", "null"
```

User-defined data types are quite useful for ensuring consistency throughout a database design. Consider, for example, a system in which it is necessary to store a social security number in multiple tables. Confusion may arise as to whether the number should be stored in character or numeric format. Furthermore, if it stored in character format, should it contain embedded hyphens? Consult Figure 3 for a list of valid options for storing social security numbers.

However, a user-defined data type can be defined, say SSN. The SSN data type will be standard and can be used for all columns that store social security number data. This ensures consistency from table to table and column to column. Whether the user-defined data type is character or numeric is not important. The point is that the data definition is consistent.

Figure 3. Social Security Number Storage Options

Data Type	Example

char(11)	"123-45-6789"
char(9)	"123456789"
integer	123456789
decimal	123,456,789

An additional benefit that is accrued by establishing user-defined data types is a higher level of abstraction in a database design. It is much easier to discuss the salary data type (with all its implied definitions, properties, and constraints) than it is to talk about a decimal(12,2) or smallmoney data type (with no implied characteristics other than its inherent type).

User-defined data types can also decrease maintenance. When a default or rule is bound to a user-defined data type, every column that is assigned that user-defined data type "inherits" the attached default and/or rule. Likewise, when a rule or default bound to a user-defined data type is changed, the change is automatically reflected in all columns that are assigned that user-defined data type.

Supporting Domains Using SQL Server

Domains have been a part of the relational model since its inception in 1969. However, no current RDBMS explicitly supports domains. SQL Server 10 supports domains only implicitly and incompletely.

What is a domain? According to Chris Date: "A domain is the set of all possible data values of some particular type." 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

Domains can be partially implemented in SQL Server using a combination of user-defined data types, rules, and defaults. 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:

1. Create a user defined data type, say prodcode, as follows:

```
sp_addtype prodcode,
"char(6)", "null"
```

2. Create a rule, say prodcode_rule, as follows:

```
create rule prodcode_rule
as

@prodcode like "[A-Z][1-
9]___[#,@]"
```

3. Create a default, say prodcode_deflt, as follows:

```
create default
prodcode_deflt as NULL
```

4. 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 over-ride the property in the user-defined data type. Bind the prodcode_rule and the prodcode_deflt to all columns containing product codes.

Remember, however, that this provides rudimentary domain support only. It does not implement full domain support as described in Codd and Date's relational writings.

Triggers

It is also possible to support user-defined integrity in SQL Server through the use of triggers. Triggers are event-driven specialized procedures that are stored in the RDBMS. Each trigger is attached to a single, specified table. Triggers can be thought of as an advanced form of "rule" or "constraint" written using procedural logic. A trigger can not be directly called or executed; it is automatically executed (or "fired") by the RDBMS as the result of an action—usually a data modification to the associated table.

Although triggers are often used for implementing referential integrity, there are many practical reasons for using triggers to implement user-defined integrity. Quite often it is impossible to code business rules into the database using only DDL. For example, the business rule may be too complex to support using a rule or check constraint. Triggers offer a flexible vehicle for the specification of user-defined integrity. Complex strings of instructions can be coded and stored within the DBMS as a trigger. Whenever data is added, removed, or modified, the logic in the trigger will be executed to ensure the required integrity constraints are maintained.

Synopsis

SQL Server 10 provides a wealth of mechanisms for supporting user-defined relational integrity. Developers would be wise to utilize these features to develop robust applications that provide system-manage data integrity, promote reusability, and provide consistent data structures.

From *DBMS Magazine*, November 1994.

Home.