Creating Tables and Defining Data Integrity

In this section I'll describe the fundamentals of creating tables and defining data integrity using T-SQL. Feel free to run the included code samples in your environment. If you don't know yet how to run code against SQL Server, Appendix A will help you get started.

As I mentioned earlier, DML rather than DDL is the focus of this book. Still, it is important that you understand how to create tables and define data integrity. I will not go into the gory details here, but I will provide a brief description of the essentials.

Before you look at the code for creating a table, remember that tables reside within schemas, and schemas reside within databases. My examples use a database called testdb and a schema called dbo. You can create a database called testdb in your environment by running the following code:

```
IF DB_ID('testdb') IS NULL
  CREATE DATABASE testdb;
```

If a database called testdb does not exist, this code creates a new one. The *DB_ID* function accepts a database name as input, and returns its internal database ID. If a database with the input name does not exist, the function returns a NULL. This is a simple way to check whether a database exists. Note that in this simple *CREATE DATABASE* statement I relied on defaults in terms of file settings such as location and initial size. In production environments you will usually explicitly specify all desired database and file settings, but for our purposes, the default settings are perfectly fine.

I will use a schema called dbo that is created automatically in every database, and is also used as the default schema for users that were not associated explicitly with another schema.

Creating Tables

The following code creates a table called Employees in the testdb database:

```
USE testdb;

IF OBJECT_ID('dbo.Employees', 'U') IS NOT NULL
  DROP TABLE dbo.Employees;

CREATE TABLE dbo.Employees
(
  empid     INT         NOT NULL,
  firstname VARCHAR(30) NOT NULL,
  lastname  VARCHAR(30) NOT NULL,
  hiredate  DATE        NOT NULL,
  mgrid     INT         NULL,
  ssn       VARCHAR(20) NOT NULL,
  salary    MONEY       NOT NULL
);
```

The *USE* statement changes the current database context to that of testdb. It is important to incorporate the *USE* statement in scripts that create objects to ensure that the objects will be created in the desired database.

The *IF* statement invokes the *OBJECT_ID* function to check whether the Employees table already exists in the current database. The *OBJECT_ID* function accepts an object name and type as inputs. The type 'U' represents a user table. This function returns the internal object ID if an object with the given input name and type exists, and NULL otherwise. If the function returns a NULL, you know that the object doesn't exist. In our case, the code drops the table if it already exists, and then creates a new one. Of course you could have chosen a different treatment, such as simply not creating the object if it already exists.

The *CREATE TABLE* statement is in charge of defining what I referred to earlier as the body of the relation. Here you specify the name of the table and, in parentheses, the definition of its attributes (columns).

Notice the use of the two-part name dbo.Employees for the table name, as recommended earlier. If you omit the schema name, SQL Server will assume the default schema associated with the database user running the code.

For each attribute you specify the attribute name, datatype, and NULLability.

In our Employees table the attributes empid (employee ID) and mgrid (manager ID) are defined as *INT* (four byte integer); firstname, lastname, and ssn (social security number) are defined as *VARCHAR* (variable length character string with the specified max supported number of characters); hiredate is defined as *DATE* and salary is defined as *MONEY*. Note that the datatype *DATE* was added in SQL Server 2008. If you are working with an earlier version of the product, use the *DATETIME* or *SMALLDATETIME* data type instead.

If you don't explicitly specify whether a column allows or disallows NULLs, SQL Server will have to rely on defaults. ANSI dictates that when a column NULLability is not specified, the assumption should be NULL (allowing NULLs), but SQL Server has settings that can change this behavior. I strongly recommend being explicit in this sense and not relying on defaults. Also, I strongly recommend defining a column as NOT NULL unless you have a compelling reason to support NULLs. If a column is not supposed to allow NULLs and you don't enforce this with a NOT NULL constraint, you can rest assured that NULLs will get there. In our Employees table all columns are defined as NOT NULL except for the mgrid column. A NULL in the mgrid attribute would represent the fact that the employee has no manager, as in the case of the CEO of the organization.

Coding Style

You should be aware of a few general notes regarding coding style, the use of white spaces (space, tab, new line, and so on), and semicolons. I'm not aware of any formal coding styles. My advice is that you use a style that you and your fellow developers feel comfortable with. What ultimately matters most is consistency and the readability and maintainability of your code. I have tried to reflect these aspects in my code throughout the book.

T-SQL allows you to use white spaces quite freely in your code. You can take advantage of this fact to facilitate readability. For example, I could have written the code in the previous section in one line. However, it wouldn't have been as readable as when I break it into multiple lines and use indentation.

The practice of using a semicolon to terminate statements is standard and is also a requirement in several other database platforms. In SQL Server you are required to use a semicolon only in particular cases, but in cases where a semicolon is not required, it doesn't interfere. I strongly recommend that you adopt the practice of terminating all statements with a semicolon. This will improve the readability of your code and in some cases will even save you some grief. (When a semicolon is required and is not specified, the error message SQL Server produces is not always very clear.)

Defining Data Integrity

As mentioned earlier, one of the great benefits in the relational model is that data integrity is an integral part of it. Data integrity that is enforced as part of the model—namely, as part of the table definitions—is considered *declarative data integrity*. Data integrity that is enforced with code—such as with stored procedures or triggers—is considered *procedural data integrity*.

Data type and NULLability choices for attributes and even the data model itself are examples of declarative data integrity constraints. In this section I will describe other examples for declarative constraints, including primary key, unique, foreign key, check, and default constraints. You can define such constraints when creating a table as part of the *CREATE TABLE* statement, or after the table was already created using an *ALTER TABLE* statement. All types of constraints except for default can be defined as composite ones—that is, based on more than one attribute.

Primary Key Constraints

A primary key constraint enforces uniqueness of rows and also disallows NULLs in the constraint attributes. Each unique set of values in the constraint attributes can appear only once in the table—in other words, only in one row. An attempt to define a primary key constraint on a column that allows NULLs will be rejected by the RDBMS. Each table can have only one primary key.

Here's an example of defining a primary key constraint on the empid column in the Employees table that you created earlier:

```
ALTER TABLE dbo.Employees
  ADD CONSTRAINT PK_Employees
  PRIMARY KEY(empid);
```

With this primary key in place, you can be assured that all empid values will be unique and known. An

attempt to insert or update a row such that the constraint would be violated will be rejected by the RDBMS and result in an error.

To enforce the uniqueness of the logical primary key constraint, SQL Server will create a unique index behind the scenes. A unique index is a physical mechanism used by SQL Server to enforce uniqueness. Indexes (not necessarily unique ones) are also used to speed up queries by avoiding unnecessary full table scans (similar to indexes in books).

Unique Constraints

A unique constraint enforces uniqueness of rows, allowing you to implement the concept of alternate keys from the relational model in your database. Unlike primary keys, multiple unique constraints can be defined in the same table. Also, a unique constraint is not restricted to columns defined as NOT NULL. ANSI SQL supports two types of unique constraints—one that allows a single NULL in a column with a unique constraint and another that allows multiple NULLs. SQL Server implemented only the former.

The following code defines a unique constraint on the ssn column in the Employees table:

```
ALTER TABLE dbo.Employees
  ADD CONSTRAINT UNQ_Employees_ssn
  UNIQUE(ssn);
```

As with a primary key constraint, SQL Server will create a unique index behind the scenes as the physical mechanism to enforce the logical unique constraint.

Foreign Key Constraints

A foreign key enforces referential integrity. This constraint is defined on a set of attributes in what's called the *referencing* table, and points to a set of candidate key (primary key or unique constraint) attributes in what's called the *referenced* table. Note that the referencing and referenced tables can be one and the same. The foreign key's purpose is to restrict the domain of values allowed in the foreign key columns to those that exist in the referenced columns.

The following code creates a table called Orders with a primary key defined on the orderid column:

```
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL
  DROP TABLE dbo.Orders;

CREATE TABLE dbo.Orders
(
  orderid   INT         NOT NULL,
  empid     INT         NOT NULL,
  custid    VARCHAR(10) NOT NULL,
  orderts   DATETIME    NOT NULL,
  qty       INT         NOT NULL,
  CONSTRAINT PK_Orders
    PRIMARY KEY(OrderID)
);
```

Say you want to enforce an integrity rule that restricts the domain of values supported by the empid column in the Orders table to the values that exist in the empid column in the Employees table. You can achieve this by defining a foreign key constraint on the empid column in the Orders table pointing to the empid column in the Employees table like so:

```
ALTER TABLE dbo.Orders
  ADD CONSTRAINT FK_Orders_Employees
  FOREIGN KEY(empid)
  REFERENCES dbo.Employees(empid);
```

Similarly, if you want to restrict the domain of values supported by the mgrid column in the Employees table to the values that exist in the empid column of the same table, you can do so by adding the following foreign key:

```
ALTER TABLE dbo.Employees
  ADD CONSTRAINT FK_Employees_Employees
  FOREIGN KEY(mgrid)
  REFERENCES Employees(empid);
```

Note that NULLs are allowed in the foreign key columns (mgrid in the last example) even if there are no NULLs in the referenced candidate key columns.

The preceding two examples are basic definitions of foreign keys that enforce a referential action called *no action*. No action means that attempts to delete rows from the referenced table or update the referenced candidate key attributes will be rejected if related rows exist in the referencing table. For example, if you try to delete an employee row from the Employees table when there are related orders in the Orders table, the RDBMS will reject such an attempt and produce an error.

You can define the foreign key with actions that will compensate for such attempts (to delete rows from the referenced table or update the referenced candidate key attributes when related rows exist in the referencing table). You can define the options *ON DELETE* and *ON UPDATE* with actions like *CASCADE*, *SET DEFAULT*, and *SET NULL* as part of the foreign key definition. *CASCADE* means that the operation (delete or update) will be cascaded to related rows. For example, *ON DELETE CASCADE* means that when you delete a row from the referenced table, the RDBMS will delete the related rows from the referencing table. *SET DEFAULT* and *SET NULL* mean that the compensating action will set the foreign key attributes of the related rows to the column's default value or NULL respectively. Note that regardless of which action you chose, the referencing table will only have orphaned rows in the case of the exception with NULLs that I mentioned earlier.

Check Constraints

A check constraint allows you to define a predicate that a row must meet to enter the table or to be modified. For example, the following check constraint ensures that the salary column in the Employees table will support only positive values:

```
ALTER TABLE dbo.Employees
  ADD CONSTRAINT CHK_Employees_salary
  CHECK(salary > 0);
```

An attempt to insert or update a row with a nonpositive salary value will be rejected by the RDBMS. Note that a check constraint rejects an attempt to insert or update a row when the predicate evaluates to FALSE. The modification will be accepted when the predicate evaluates to either TRUE or UNKNOWN. For example, salary -1000 will be rejected, while salaries 50000 and NULL will both be accepted.

When adding CHECK and FOREIGN KEY constraints, you can specify an option called WITH NOCHECK telling the RDBMS that you want it to bypass constraint checking for existing data. This is considered a bad practice because you cannot be sure that your data is consistent.

You can also disable or enable existing CHECK and FOREIGN KEY constraints.

Default Constraints

A default constraint is associated with a particular attribute. It is an expression that is used as the default value when an explicit value is not specified for the attribute when you insert a row. For example, the following code defines a default constraint for the orderts attribute (representing the order's timestamp):

```
ALTER TABLE dbo.Orders
  ADD CONSTRAINT DFT_Orders_orderts
  DEFAULT(CURRENT_TIMESTAMP) FOR orderts;
```

The default expression invokes the *CURRENT_TIMESTAMP* function, which returns the current date and time value. Once this default expression is defined, whenever you insert a row in the Orders table and do not explicitly specify a value in the orderts attribute, SQL Server will set the attribute value to *CURRENT_TIMESTAMP*.