Inserting Data

T-SQL provides several statements to insert data into tables: *INSERT VALUES*, *INSERT SELECT*, *INSERT EXEC*, *SELECT INTO*, and *BULK INSERT*. I'll first describe those statements and then I'll talk about a column property called *IDENTITY* that automatically generates numeric values in the target column upon insert.

The INSERT VALUES Statement

You use the *INSERT VALUES* statement to insert rows into a table based on specified values. To demonstrate this statement and others, you will work with a table called Orders in the dbo schema in the tempdb database. Run the following code to create the Orders table:

```
USE tempdb;

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

CREATE TABLE dbo.Orders
(
  orderid   INT          NOT NULL
    CONSTRAINT PK_Orders  PRIMARY KEY,
  orderdate DATE         NOT NULL
    CONSTRAINT DFT_orderdate DEFAULT(CURRENT_TIMESTAMP),
  empid     INT          NOT NULL,
  custid    VARCHAR(10) NOT NULL
)
```

The following example demonstrates how to use the *INSERT VALUES* statement to insert a single row into the Orders table:

```
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid)
  VALUES(10001, '20090212', 3, 'A');
```

Specifying the target column names right after the table name is optional, but by doing so you control the value-column associations instead of relying on the order in which the columns appeared when the table was defined (or the table structure was last altered).

If you specify a value for a column, Microsoft SQL Server will use that value. If you don't, SQL Server will check whether a default value is defined for the column, and if so, the default will be used. If a default value isn't defined and the column allows NULLs, a NULL will be used. If you do not specify a value for a column that does not somehow get its value automatically, your *INSERT* statement will fail. As an example of relying on a default value or expression, the following statement inserts a row into the Orders table without specifying a value for the orderdate column, but because this column has a default expression defined for it (CURRENT_TIMESTAMP), that default will be used:

```
INSERT INTO dbo.Orders(orderid, empid, custid)
  VALUES(10002, 5, 'B');
```

SQL Server 2008 enhances the VALUES clause by allowing you to specify multiple rows separated by commas. For example, the following statement inserts four rows into the Orders table:

```
INSERT INTO dbo.Orders
  (orderid, orderdate, empid, custid)
VALUES
  (10003, '20090213', 4, 'B'),
  (10004, '20090214', 1, 'A'),
  (10005, '20090213', 1, 'C'),
  (10006, '20090215', 3, 'C');
```

This statement is processed as an atomic operation, meaning that if any row fails to enter the table, none of the rows in the statement enters the table.

There's more to this enhancement than meets the eye. Not only was the *INSERT VALUES* statement enhanced, but the VALUES clause itself was also enhanced so that you can use it to construct a virtual table. This feature, called Row Value Constructor and also Table Value Constructor, is standard. This means that you can define a table expression based on the VALUES clause. Here's an example of a query against a

derived table that is defined based on the VALUES clause:

```
SELECT *
FROM ( VALUES
         (10003, '20090213', 4, 'B'),
         (10004, '20090214', 1, 'A'),
         (10005, '20090213', 1, 'C'),
         (10006, '20090215', 3, 'C') )
     AS O(orderid, orderdate, empid, custid);
```

Following the parentheses that contain the table value constructor, you assign an alias to the table (O in our case), and following the table alias you assign aliases to the target columns in parentheses. This query generates the following output:

```
orderid     orderdate   empid       custid
----------- ----------- ----------- ------
10003       20090213    4           B
10004       20090214    1           A
10005       20090213    1           C
10006       20090215    3           C
```

The INSERT SELECT Statement

The *INSERT SELECT* statement inserts a set of rows returned by a SELECT query into a target table. The syntax is very similar to that of an *INSERT VALUES* statement, but instead of the VALUES clause you specify a SELECT query. For example, the following code inserts into the dbo.Orders table in tempdb the result of a query against the Sales.Orders table in TSQLFundamentals2008 returning orders that were shipped to the UK:

```
USE tempdb;

INSERT INTO dbo.Orders(orderid, orderdate, empid, custid)
  SELECT orderid, orderdate, empid, custid
  FROM TSQLFundamentals2008.Sales.Orders
  WHERE shipcountry = 'UK';
```

The *INSERT SELECT* statement also optionally allows you to specify the target column names, and the recommendations I gave earlier regarding specifying those names remain the same.

The requirement to provide values for all columns that do not somehow get their values automatically and the implicit use of default values/NULLs when a value is not provided also behave the same way as with the *INSERT VALUES* statement. The *INSERT SELECT* statement is performed as an atomic operation, so if any row fails to enter the target table, none of the rows enters the table.

If you wanted to construct a virtual table based on values prior to SQL Server 2008, you had to use multiple *SELECT* statements, each returning a single row based on values, and unify the rows with UNION ALL set operations. In the context of an *INSERT SELECT* statement, you could use this technique to insert multiple rows based on values in a single statement that is considered an atomic operation. For example, the following statement inserts four rows based on values into the Orders table:

```
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid)
  SELECT 10007, '20090215', 2, 'B' UNION ALL
  SELECT 10008, '20090215', 1, 'C' UNION ALL
  SELECT 10009, '20090216', 2, 'C' UNION ALL
  SELECT 10010, '20090216', 3, 'A';
```

As I mentioned earlier, SQL Server 2008 supports table value constructors, so you don't really need this technique anymore.

Almost all *INSERT SELECT* operations were fully logged (that is, fully written to the database's transaction log) prior to SQL Server 2008, and compared to minimally logged operations, fully logged operations can be substantially slower. SQL Server 2008 supports minimal logging in more scenarios than in previous versions, including with the *INSERT SELECT* statement. Performance discussions are outside the scope of this book, but if you're interested in learning more, you can find details in the SQL Server Books Online article "Operations That Can Be Minimally Logged."

The INSERT EXEC Statement

You use the *INSERT EXEC* statement to insert a result set returned from a stored procedure or a dynamic SQL batch into a target table. You'll find information about stored procedures, batches, and dynamic SQL in Chapter 10, "Programmable Objects." The *INSERT EXEC* statement is very similar in syntax and concept to the *INSERT SELECT* statement, but instead of a *SELECT* statement, you specify an *EXEC* statement.

For example, the following code creates a stored procedure called Sales.usp_getorders in the TSQLFundamentals2008 database, returning orders that were shipped to a given input country (*@country* parameter):

```
USE TSQLFundamentals2008;

IF OBJECT_ID('Sales.usp_getorders', 'P') IS NOT NULL
  DROP PROC Sales.usp_getorders;
GO
CREATE PROC Sales.usp_getorders
  @country AS NVARCHAR(40)
AS

SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE shipcountry = @country;
GO
```

To test the stored procedure, execute it with the input country France:

```
EXEC Sales.usp_getorders @country = 'France';
```

You get the following output:

```
orderid     orderdate                empid       custid
----------- ------------------------ ----------- -----------
10248       2006-07-04 00:00:00.000   5           85
10251       2006-07-08 00:00:00.000   3           84
10265       2006-07-25 00:00:00.000   2           7
10274       2006-08-06 00:00:00.000   6           85
10295       2006-09-02 00:00:00.000   2           85
10297       2006-09-04 00:00:00.000   5           7
10311       2006-09-20 00:00:00.000   1           18
10331       2006-10-16 00:00:00.000   9           9
10334       2006-10-21 00:00:00.000   8           84
10340       2006-10-29 00:00:00.000   1           9
...

(77 row(s) affected)
```

Using an *INSERT EXEC* statement, you can direct the result set returned from the procedure to the dbo.Orders table in the tempdb database:

```
USE tempdb;

INSERT INTO dbo.Orders(orderid, orderdate, empid, custid)
  EXEC TSQLFundamentals2008.Sales.usp_getorders @country = 'France';
```

The SELECT INTO Statement

The *SELECT INTO* statement is a nonstandard T-SQL statement that creates a target table and populates it with the result set of a query. By "nonstandard," I mean not part of the ANSI SQL standard. You cannot use this statement to insert data into an existing table. In terms of syntax, simply add INTO *<target_table_name>* right before the FROM clause of the SELECT query that you want to use to produce the result set. For example, the following code creates a table called dbo.Orders in tempdb and populates it with all rows from the Sales.Orders table from TSQLFundamentals2008:

```
USE tempdb;

IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;
SELECT orderid, orderdate, empid, custid
INTO dbo.Orders
FROM TSQLFundamentals2008.Sales.Orders;
```

The target table's structure and data are based on the source table. The *SELECT INTO* statement copies from the source the base structure (column names, types, NULLability, IDENTITY property) and the data. There are three things that the statement does not copy from the source: constraints, indexes, and triggers. If you need those in the target, you will need to create them yourself.

One of the advantages of the *SELECT INTO* statement is that as long as a database property called *Recovery Model* is not set to FULL, the SELECT INTO operation is performed in a minimally logged mode. This translates to a very fast operation compared to a fully logged one.

If you need to use a *SELECT INTO* statement with set operations, you specify the INTO clause right in front of the FROM clause of the first query. For example, the following *SELECT INTO* statement creates a table called Locations and populates it with the result of an EXCEPT set operation, returning locations where there are customers but not employees:

```
USE tempdb;

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

SELECT country, region, city
INTO dbo.Locations
FROM TSQLFundamentals2008.Sales.Customers

EXCEPT

SELECT country, region, city
FROM TSQLFundamentals2008.HR.Employees;
```

The BULK INSERT Statement

You use the *BULK INSERT* statement to insert into an existing table data originating from a file. In the statement, you specify the target table, the source file, and options. You can specify many options, including the data file type (for example, char or native), the field terminator, the row terminator, and others—all of which are fully documented.

For example, the following code bulk inserts the contents of the file 'c:\temp\orders.txt' into the table dbo.Orders in tempdb, specifying that the data file type is char, the field terminator is a comma, and the row terminator is newline:

```
USE tempdb;

BULK INSERT dbo.Orders FROM 'c:\temp\orders.txt'
  WITH
    (
       DATAFILETYPE    = 'char',
       FIELDTERMINATOR = ',',
       ROWTERMINATOR   = '\n'
    );
```

Note that if you want to actually run this statement you need to place the orders.txt file provided along with the source code for this book in the c:\temp folder.

You can run the *BULK INSERT* statement in a fast, minimally logged mode in certain scenarios provided that certain requirements are met. For details, please see "Prerequisites for Minimal Logging in Bulk Import" in SQL Server Books Online.

The IDENTITY Property

SQL Server allows you to define a property called *IDENTITY* for a column with any numeric type with a scale of 0 (no fraction). This property generates values automatically upon *INSERT* based on seed (first value) and increment (step value) that are provided in the column's definition. Typically you would use this property to generate s*urrogate keys*, which are keys that are produced by the system and are not derived from the application data.

For example, the following code creates a table called dbo.T1 in tempdb:

```
USE tempdb;
```

```
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;

CREATE TABLE dbo.T1
(
  keycol  INT         NOT NULL IDENTITY(1, 1)
    CONSTRAINT PK_T1 PRIMARY KEY,
  datacol VARCHAR(10) NOT NULL
    CONSTRAINT CHK_T1_datacol CHECK(datacol LIKE '[A-Za-z]%')
);
```

The table contains a column called keycol which is defined with an *IDENTITY* property using 1 as the seed and 1 as the increment. The table also contains a character string column called datacol whose data is restricted with a CHECK constraint to strings starting with an alpha character.

In your *INSERT* statements, you should completely ignore the identity column, pretending as though it isn't in the table. For example, the following code inserts three rows into the table, specifying values only for the column datacol:

```
INSERT INTO dbo.T1(datacol) VALUES('AAAAA');
INSERT INTO dbo.T1(datacol) VALUES('CCCCC');
INSERT INTO dbo.T1(datacol) VALUES('BBBBB');
```

SQL Server produced the values for keycol automatically. To see the values that SQL Server produced, query the table:

```
SELECT * FROM dbo.T1;
```

You get the following output:

```
keycol      datacol
----------- ----------
1           AAAAA
2           CCCCC
3           BBBBB
```

When you query the table, naturally you can refer to the identity column by its name (keycol in our case). SQL Server also provides a way to refer to the identity column using the more generic form $identity. This form is supported as of SQL Server 2005, replacing the deprecated form IDENTITYCOL. The deprecated form is still supported for backward compatibility, but will be removed from the product in a future version.

For example, the following query selects the identity column from T1 using the generic form:

```
SELECT $identity FROM dbo.T1;
```

This query returns the following output:

```
keycol
-----------
1
2
3
```

When you insert a new row into the table, SQL Server generates a new identity value based on the current identity value in the table and the increment. If you need to obtain the newly generated identity value—for example, to insert child rows into a referencing table—you query one of two functions called *@@identity* and *SCOPE_IDENTITY()*. The *@@identity* function is a legacy feature (predating even SQL Server 2000), and it returns the last identity value generated by the session, regardless of scope. *SCOPE_IDENTITY()* returns the last identity value generated by the session in the current scope (for example, the same procedure). Except for very special cases when you don't really care about scope, you should use the *SCOPE_IDENTITY* function.

For example, the following code inserts a row into the table T1, obtains the newly generated identity value into a variable by querying the *SCOPE_IDENTITY* function, and queries the variable:

```
DECLARE @new_key AS INT;
```

```
INSERT INTO dbo.T1(datacol) VALUES('AAAAA');

SET @new_key = SCOPE_IDENTITY();

SELECT @new_key AS new_key
```

If you ran all previous code samples provided in this section, this code returns the following output:

```
new_key
-----------
4
```

Remember that both *@@identity* and *SCOPE_IDENTITY* return the last identity value produced by the current session. Neither is affected by inserts issued by other sessions. However, if you want to know the current identity value in a table (last value produced) regardless of session, you should use the *IDENT_CURRENT* function and provide the table name as input. For example, run the following code from a new session (not the one where you ran the previous *INSERT* statements):

```
SELECT
  SCOPE_IDENTITY() AS [SCOPE_IDENTITY],
  @@identity AS [@@identity],
  IDENT_CURRENT('dbo.T1') AS [IDENT_CURRENT];
```

You get the following output:

```
SCOPE_IDENTITY   @@identity   IDENT_CURRENT
---------------- ------------ -------------
NULL             NULL         4
```

Both *@@identity* and *SCOPE_IDENTITY* returned NULLs because no identity values were created in the session where this query ran. *IDENT_CURRENT* returned the value 4 because it returns the current identity value in the table, regardless of the session in which it was produced.

Note the following important details regarding the identity property.

The change to the current identity value in a table is not undone if the *INSERT* that generated the change fails or the transaction in which the statement runs is rolled back. For example, run the following *INSERT* statement, which contradicts the CHECK constraint defined in the table:

```
INSERT INTO dbo.T1(datacol) VALUES('12345');
```

The insert fails, and you get the following error:

```
Msg 547, Level 16, State 0, Line 1
The INSERT statement conflicted with the CHECK constraint "CHK_T1_datacol". The conflict
occurred in database "tempdb", table "dbo.T1", column 'datacol'.
The statement has been terminated.
```

Even though the insert failed, the current identity value in the table changed from 4 to 5, and this change was not undone because of the failure. This means that the next insert will produce the value 6:

```
INSERT INTO dbo.T1(datacol) VALUES('EEEEE');
```

Query the table:

```
SELECT * FROM dbo.T1;
```

Notice a gap between the values 4 and 6 in the output:

```
keycol      datacol
----------- ----------
1           AAAAA
2           CCCCC
3           BBBBB
4           AAAAA
6           EEEEE
```

Of course, this means that you should only rely on the identity property to auto-generate values when you don't care about having gaps. Otherwise, you should consider using your own alternative mechanism.

Another important aspect of the identity property is that you cannot add it to an existing column or remove it from an existing column; you can only define the property along with a column as part of a *CREATE TABLE* statement or an *ALTER TABLE* statement that adds a new column. However, SQL Server does allow you to explicitly specify your own values for the identity column in *INSERT* statements, provided that you set a session option called *IDENTITY_INSERT* against the table involved. No option allows you to update an identity column, though.

For example, the following code demonstrates how to insert a row to T1 with the explicit value 5 in keycol:

```
SET IDENTITY_INSERT dbo.T1 ON;
INSERT INTO dbo.T1(keycol, datacol) VALUES(5, 'FFFFF');
SET IDENTITY_INSERT dbo.T1 OFF;
```

Interestingly, SQL Server changes the current identity value in the table only if the explicit value provided for the identity column is higher than the current identity value in the table. Because the current identity value in the table prior to running the preceding code was 6, and the *INSERT* statement in this code used the lower explicit value 5, the current identity value in the table did not change. So if at this point, after running the preceding code, you query the *IDENT_CURRENT* function for this table, you will get 6 and not 5. This way the next *INSERT* statement against the table will produce the value 7:

```
INSERT INTO dbo.T1(datacol) VALUES('GGGGG');
```

Query the current contents of the table T1:

```
SELECT * FROM dbo.T1;
```

You get the following output:

```
keycol      datacol
----------- ----------
1           AAAAA
2           CCCCC
3           BBBBB
4           AAAAA
5           FFFFF
6           EEEEE
7           GGGGG
```

It is important to understand that the identity property itself does not enforce uniqueness in the column. I already explained that you can provide your own explicit values after setting the *IDENTITY_INSERT* option to ON, and those values can be ones that already exist in rows in the table. Also, you can reseed the current identity value in the table by using the *DBCC CHECKIDENT* command. For details about the syntax of the *DBCC CHECKIDENT* command please refer to "*DBCC CHECKIDENT* (Transact-SQL)" at SQL Server Books Online. In short, the identity property does not enforce uniqueness. If you need to guarantee uniqueness in an identity column, make sure you also define a primary key or a unique constraint on that column.