Deleting Data

T-SQL provides two statements for deleting rows from a table—*DELETE* and *TRUNCATE*. In this section, I'll describe those statements. The examples I provide in this section are against copies of the Customers and Orders tables from the TSQLFundamentals2008 database created in the tempdb database. Run the following code to create and populate those tables:

```
USE tempdb;

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

SELECT * INTO dbo.Customers FROM TSQLFundamentals2008.Sales.Customers;
SELECT * INTO dbo.Orders FROM TSQLFundamentals2008.Sales.Orders;

ALTER TABLE dbo.Customers ADD
  CONSTRAINT PK_Customers PRIMARY KEY(custid);
ALTER TABLE dbo.Orders ADD
  CONSTRAINT PK_Orders PRIMARY KEY(orderid),
  CONSTRAINT FK_Orders_Customers FOREIGN KEY(custid)
    REFERENCES dbo.Customers(custid);
```

The DELETE Statement

The *DELETE* statement is a standard statement used to delete data from a table based on a predicate. The standard statement has only two clauses—the FROM clause, in which you specify the target table name, and a WHERE clause, in which you specify a predicate. Only the subset of rows for which the predicate evaluates to TRUE will be deleted.

For example, the following statement deletes, from the dbo.Orders table in tempdb, all orders that were placed prior to 2007:

```
USE tempdb;

DELETE FROM dbo.Orders
WHERE orderdate < '20070101';
```

Run this statement and SQL Server will report that it deleted 152 rows:

```
(152 row(s) affected)
```

Note that the message indicating how many rows were affected only appears if the session option NOCOUNT is OFF, which it is by default. If it is ON, SQL Server Management Studio will only state that the command completed successfully.

The *DELETE* statement is fully logged. Therefore, you should expect it to run for a while when you delete a large number of rows.

The TRUNCATE Statement

The *TRUNCATE* statement is a nonstandard statement that deletes all rows from a table. Unlike the *DELETE* statement, *TRUNCATE* has no filter. For example, to delete all rows from a table called dbo.T1, you run the following code:

```
TRUNCATE TABLE dbo.T1;
```

The advantage that *TRUNCATE* has over *DELETE* is that the former is minimally logged while the latter is fully logged, resulting in significant performance differences. For example, if you use the *TRUNCATE* statement to delete all rows from a table with millions of rows, the operation will finish in a matter of seconds. If you use the *DELETE* statement, the operation can take minutes or even hours.

TRUNCATE and *DELETE* also have a functional difference when the table has an identity column. *TRUNCATE* resets the identity value back to the original seed, while *DELETE* doesn't.

The *TRUNCATE* statement is not allowed when the target table is referenced by a foreign key constraint, even if the referencing table is empty and even if the foreign key is disabled. The only way to allow a *TRUNCATE* statement is to drop all foreign keys referencing the table.

Because the *TRUNCATE* statement is so fast, it can also be dangerous. Accidents such as truncating or dropping the incorrect table can happen. For example, let's say you have connections open against both the production and the development environments, and you submit your code in the wrong connection. Both the *TRUNCATE* and *DROP* statements are so fast that before you realize your mistake, the transaction is committed. To prevent such accidents, you can protect a production table by simply creating a dummy table with a foreign key pointing to the production table. You can even disable the foreign key so that it won't have any impact on performance. As I mentioned earlier, even when disabled, this foreign key prevents truncating or dropping the referenced table.

DELETE Based on a Join

T-SQL supports a nonstandard DELETE syntax based on joins. The join itself serves a filtering purpose because it has a filter based on a predicate (the ON clause). The join also gives you access to attributes of related rows from another table that you can refer to in the WHERE clause. This means that you can delete rows from one table based on a filter against attributes in related rows from another table.

For example, the following statement deletes orders placed by customers from the USA:

```
USE tempdb;

DELETE FROM O
FROM dbo.Orders AS O
  JOIN dbo.Customers AS C
    ON O.custid = C.custid
WHERE C.country = N'USA';
```

Very much like in a *SELECT* statement, the first clause that is logically processed in a *DELETE* statement is the FROM clause (the second one that appears in this statement). Then the WHERE clause is processed, and finally the DELETE clause. The way to "read" or interpret this query is: "The query joins the Orders table (aliased as O) with the Customers table (aliased as C) based on a match between the order's customer ID and the customer's customer ID. The query then filters only orders placed by customers from the USA. Finally, the query deletes all qualifying rows from O (the alias representing the Orders table)."

The two FROM clauses in a *DELETE* statement based on a join might be confusing. But when you develop the code, develop it as if it were a *SELECT* statement with a join. That is, start with the FROM clause with the joins, move on to the WHERE clause, and finally, instead of specifying a SELECT clause, specify a DELETE clause with the alias of the side of the join that is supposed to be the target for the delete.

As I mentioned earlier, a *DELETE* statement based on a join is nonstandard. If you want to stick to standard code, you can use subqueries instead of joins. For example, the following *DELETE* statement uses a subquery to achieve the same task:

```
DELETE FROM dbo.Orders
WHERE EXISTS
  (SELECT *
   FROM dbo.Customers AS C
   WHERE Orders.Custid = C.Custid
     AND C.Country = 'USA');
```

This code deletes all rows from the Orders table where a related customer in the customers table from the USA exists.

SQL Server will most likely process the two queries the same way; therefore you shouldn't expect any performance difference between the two. So why do people even consider using the nonstandard syntax? Some people feel more comfortable with joins while others feel more comfortable with subqueries. I usually recommend sticking to the standard as much as possible unless you have a very compelling reason to do otherwise—for example, in the case of a big performance difference.