Inside Microsoft SQL Server 2005: T-SQL Querying

by Itzik Ben-Gan, Lubor Kollar and Dejan Sarka

Chapter 8: Data Modification

This chapter covers different facets of data modification. I'll discuss logical aspects such as inserting new rows and removing rows with duplicate data and performance aspects such as dealing with large volumes of data. Note that I covered some aspects of data modification in other chapters where they fit the subject matter better. You can find coverage of modifications with TOP in Chapter 7 and of the BULK rowset provider, tempdb, and transactions in *Inside Microsoft SQL Server 2005: T-SQL Programming* (Microsoft Press, 2006). I organized this chapter in sections based on the three main types of data modification activities: inserting data, deleting data, and updating data. As usual for this book, I'll cover solutions in Microsoft SQL Server 2000 as well as enhancements in SQL Server 2005.

Inserting Data

In this section, I'll cover several subjects related to inserting data, including: SELECT INTO, INSERT EXEC, inserting new rows, INSERT with OUTPUT, and sequence mechanisms.

SELECT INTO

SELECT INTO is a statement that creates a new table containing the result set of a query, instead of returning the result set to the caller. For example, the following statement creates a temporary table called #MyShippers and populates it with all rows from the Shippers table in the Northwind database:

```
SELECT ShipperID, CompanyName, Phone
INTO #MyShippers
FROM Northwind.dbo.Shippers;
```

SELECT INTO is a BULK operation. (See the "Other Performance Considerations" section at the end of the chapter for details.) Therefore, when the database recovery model is not FULL, it's very fast compared to the alternative of creating a table and then using INSERT INTO.

The columns of the new table inherit their names, datatypes, nullability, and IDENTITY property from the query's result set. SELECT INTO doesn't copy constraints, indexes, or triggers from the query's source. If you need the results in a table with the same indexes, constraints, and triggers as the source, you have to add them afterwards.

If you need a "fast and dirty" empty copy of some table, SELECT INTO allows you to obtain such a copy very simply. You don't have to script the CREATE TABLE statement and change the table's name. All you need to do is issue the following statement:

```
SELECT * INTO target_table FROM source_table WHERE 1 = 2;
```

The optimizer is smart enough to realize that no source row will satisfy the filter *1=2*. Therefore, SQL Server won't bother to physically access the source data; rather, it will create the target table based on the schema of the source. Here's an example that creates a table called MyOrders in tempdb, based on the schema of the Orders table in Northwind:

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.MyOrders') IS NOT NULL
  DROP TABLE dbo.MyOrders;
GO

SELECT *
INTO dbo.MyOrders
FROM Northwind.dbo.Orders
WHERE 1 = 2;
```

Keep in mind that if a source column has the IDENTITY property, the target will have it as well. For example, the *OrderID* column in the Orders table has the IDENTITY property. If you don't want the IDENTITY property to be copied to the target column, simply apply any type of manipulation to the source column. For example, you can use the expression *OrderID + 0 AS OrderID* as follows:

```
IF OBJECT_ID('dbo.MyOrders') IS NOT NULL
  DROP TABLE dbo.MyOrders;
GO

SELECT OrderID+0 AS OrderID, CustomerID, EmployeeID, OrderDate,
```

```
  RequiredDate, ShippedDate, ShipVia, Freight, ShipName,
  ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry
INTO dbo.MyOrders
FROM Northwind.dbo.Orders
WHERE 1 = 2;
```

In this case, the *OrderID* column in the target MyOrders table doesn't have the IDENTITY property.

Tip Suppose you want to insert the result set of a stored procedure or a dynamic batch into a new table, but you don't know what the schema is that you need to create. You can use a SELECT INTO statement, specifying OPENQUERY in the FROM clause, referring to your own server as if it were a linked server:

```
EXEC sp_serveroption <your_server>, 'data access', true;
SELECT * INTO <target_table>
FROM OPENQUERY(<your_server>,
  'EXEC {<proc_name> | (<dynamic_batch>)}') AS O;
```

INSERT EXEC

The INSERT EXEC statement allows you to direct a table result set returned from a stored procedure or dynamic batch to an existing table:

```
INSERT INTO <target_table> EXEC {<proc_name> | (<dynamic_batch>)};
```

This statement is very handy when you need to set aside the result set of a stored procedure or dynamic batch for further processing at the server, as opposed to just returning the result set back to the client.

I'll demonstrate practical uses of the INSERT EXEC statement through an example. Recall the discussion about paging techniques in Chapter 7. I provided a stored procedure called *usp_firstpage*, which returns the first page of orders based on *OrderDate, OrderID* ordering. I also provided a stored procedure called *usp_nextpage*, which returns the next page of orders based on an input key (*@anchor*) representing the last row in the previous page. In this section, I will use slightly revised forms of the stored procedures, which I'll call *usp_firstrows* and *usp_nextrows*. Run the code in Listing 8-1 to create both procedures.

Listing 8-1: Creation script for paging stored procedures

```
USE Northwind;
GO

-- Index for paging problem
IF INDEXPROPERTY(OBJECT_ID('dbo.Orders'),
      'idx_od_oid_i_cid_eid', 'IndexID') IS NOT NULL
  DROP INDEX dbo.Orders.idx_od_oid_i_cid_eid;
GO
CREATE INDEX idx_od_oid_i_cid_eid
  ON dbo.Orders(OrderDate, OrderID, CustomerID, EmployeeID);
GO

-- First Rows
IF OBJECT_ID('dbo.usp_firstrows') IS NOT NULL
  DROP PROC dbo.usp_firstrows;
GO
CREATE PROC dbo.usp_firstrows
  @n AS INT = 10 -- num rows
AS

SELECT TOP(@n) ROW_NUMBER() OVER(ORDER BY OrderDate, OrderID) AS RowNum,
  OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.Orders
ORDER BY OrderDate, OrderID;
GO

-- Next Rows
IF OBJECT_ID('dbo.usp_nextrows') IS NOT NULL
  DROP PROC dbo.usp_nextrows;
GO
CREATE PROC dbo.usp_nextrows
```

```
  @anchor_rownum AS INT = 0, -- row number of last row in prev page
  @anchor_key     AS INT, -- key of last row in prev page,
  @n              AS INT = 10 -- num rows
AS
SELECT TOP(@n)
  @anchor_rownum
    + ROW_NUMBER() OVER(ORDER BY O.OrderDate, O.OrderID) AS RowNum,
  O.OrderID, O.OrderDate, O.CustomerID, O.EmployeeID
FROM dbo.Orders AS O
  JOIN dbo.Orders AS A
    ON A.OrderID = @anchor_key
    AND (O.OrderDate >= A.OrderDate
        AND (O.OrderDate > A.OrderDate
            OR O.OrderID > A.OrderID))
ORDER BY O.OrderDate, O.OrderID;
GO
```

Note The stored procedures use new features in SQL Server 2005, so you won't be able to create them in SQL Server 2000.

The stored procedure *usp_firstrows* returns the first *@n* rows of Orders, based on *OrderDate* and *OrderID* ordering. In addition to the columns that *usp_firstpage* returned, *usp_firstrows* (as well as *usp_nextrows*) also returns *RowNum*, a column representing the global logical position of the row in the full Orders table under the aforementioned ordering. Because *usp_firstrows* returns the first page of rows, *RowNum* is just the row number within the result set.

The stored procedure *usp_nextrows* returns the *@n* rows following an anchor row, whose key is provided as input (*@anchor_key*). For a row in the result set of *usp_nextrows*, *RowNum* equals the anchor's global row number (*@anchor_rownum*) plus the result row's logical position within the qualifying set. If you don't want the stored procedure to return a global row number—rather, just the row number within the qualifying set—don't specify a value in the input parameter. In such a case, the default 0 will be used as the anchor row number, and the minimum row number that will be assigned will be 1.

Suppose you want to allow the user to request any range of rows without limiting the solution to forward-only paging. You also want to avoid rescanning large portions of data from the Orders table. You need to develop some caching mechanism where you set aside a copy of the rows you already scanned, along with row numbers representing their global logical position throughout the pages. Upon a request for a range of rows (a page), you first check whether rows are missing from the cache. In such a case, you insert the missing rows into the cache. You then query the cache to return the requested page. Here's an example of how you can implement a server-side solution of such a mechanism.

Run the following code to create the #CachedPages temporary table:

```
IF OBJECT_ID('tempdb..#CachedPages') IS NOT NULL
  DROP TABLE #CachedPages;
GO
CREATE TABLE #CachedPages
(
  RowNum INT NOT NULL PRIMARY KEY,
  OrderID INT NOT NULL UNIQUE,
  OrderDate DATETIME,
  CustomerID NCHAR(5),
  EmployeeID INT
);
```

The caching logic is encapsulated in the stored procedure *usp_getpage*, which you create by running the code in Listing 8-2.

Listing 8-2: Creation script for the stored procedure *usp_getpage*

```
IF OBJECT_ID('dbo.usp_getpage') IS NOT NULL
  DROP PROC dbo.usp_getpage;
GO
CREATE PROC dbo.usp_getpage
  @from_rownum AS INT, -- row number of first row in requested page
  @to_rownum   AS INT, -- row number of last row in requested page
  @rc          AS INT OUTPUT -- number of rows returned
```

```
AS

SET NOCOUNT ON;

DECLARE
  @last_key    AS INT, -- key of last row in #CachedPages
  @last_rownum AS INT, -- row number of last row in #CachedPages
  @numrows     AS INT; -- number of missing rows in #CachedPages

-- Get anchor values from last cached row
SELECT @last_rownum = RowNum, @last_key = OrderID
FROM (SELECT TOP(1) RowNum, OrderID
     FROM #CachedPages ORDER BY RowNum DESC) AS D;

-- If temporary table is empty insert first rows to #CachedPages
IF @last_rownum IS NULL
   INSERT INTO #CachedPages
     EXEC dbo.usp_firstrows
       @n = @to_rownum;
ELSE

BEGIN
SET @numrows = @to_rownum - @last_rownum;

IF @numrows > 0
  INSERT INTO #CachedPages
    EXEC dbo.usp_nextrows
      @anchor_rownum = @last_rownum,
      @anchor_key    = @last_key,
      @n             = @numrows;
END

-- Return requested page
SELECT *
FROM #CachedPages
WHERE RowNum BETWEEN @from_rownum AND @to_rownum
ORDER BY RowNum;

SET @rc = @@rowcount;
GO
```

The stored procedure accepts the row numbers representing the first row in the requested page (*@from_rownum*) and the last (*@to_rownum*) as inputs. Besides returning the requested page of rows, the stored procedure also returns an output parameter holding the number of rows returned (*@rc*). You can inspect the output parameter to determine whether you've reached the last page.

The stored procedure's code first queries the #CachedPages temporary table in order to store aside in the local variables *@last_rownum* and *@last_key* the row number and key of the last cached row, respectively. If the temporary table is empty (*@last_rownum IS NULL*), the code invokes the *usp_firstrows* procedure with an INSERT EXEC statement to populate #CachedPages with the first rows up to the requested high boundary row number. If the temporary table already contains rows, the code checks whether rows from the requested page are missing from it (*@to_rownum – @last_rownum > 0*). In such a case, the code invokes the *usp_nextrows* procedure to insert all missing rows up to the requested high boundary row number to the temporary table.

Finally, the code queries the #CachedPages temporary table to return the requested range of rows, and it stores the number of returned rows in the output parameter *@rc*.

To get the first page of rows, assuming a page size of 10, run the following code:

```
DECLARE @rc AS INT;

EXEC dbo.usp_getpage
  @from_rownum = 1,
  @to_rownum   = 10,
  @rc          = @rc OUTPUT;
```

```
IF @rc = 0
  PRINT 'No more pages.'
ELSE IF @rc < 10
  PRINT 'Reached last page.';
```

You will get back the first 10 rows based on *OrderDate* and *OrderID* ordering. Notice in the code that you can inspect the output parameter to determine whether there are no more pages (*@rc = 0*), or whether you've reached the last page (*@rc < 10*).

Query the #CachedPages temporary table, and you will find that 10 rows were cached:

```
SELECT * FROM #CachedPages;
```

Further requests for rows that were already cached will be satisfied from #CachedPages without the need to access the Orders table. Querying #CachedPages is very efficient because the table contains a clustered index on the *RowNum* column. Only the requested rows are physically accessed.

If you now run the preceding code specifying row numbers 21 to 30 as inputs, the *usp_getpage* procedure will add rows 11 through 30 to the temporary table, and return rows 21 through 30. Following requests for rows up to row 30 will be satisfied solely from the temporary table.

Once you're done experimenting with this paging technique, run the following code for cleanup:

```
IF OBJECT_ID('tempdb..#CachedPages') IS NOT NULL
  DROP TABLE #CachedPages;
GO
IF INDEXPROPERTY(OBJECT_ID('dbo.Orders'),
      'idx_od_oid_i_cid_eid', 'IndexID') IS NOT NULL
  DROP INDEX dbo.Orders.idx_od_oid_i_cid_eid;
GO
IF OBJECT_ID('dbo.usp_firstrows') IS NOT NULL
  DROP PROC dbo.usp_firstrows;
GO
IF OBJECT_ID('dbo.usp_nextrows') IS NOT NULL
  DROP PROC dbo.usp_nextrows;
GO
IF OBJECT_ID('dbo.usp_getpage') IS NOT NULL
  DROP PROC dbo.usp_getpage;
GO
```

Inserting New Rows

The problem that is the focus of this section involves inserting rows from some source table into a target table, but filtering only rows whose keys do not exist yet in the target. You might face this problem when you need to update a master table from a table of additions and changes—for example, updating a central data warehouse with information from regional centers. In this section, I'll focus on the part of the problem involving inserting new rows.

There are several techniques you can choose from. The appropriate technique for a given task, in terms of simplicity and performance, will depend on several factors. Does the source table contain rows with duplicate values in attributes that correspond to the target table's key? If so, what is their density? And are the rows with the duplicate values guaranteed to be completely identical, or are the identical parts only the attributes making the target key? The different scenarios that I mentioned might not be clear at the moment, but I'll provide more details as I explain the scenarios in context.

To demonstrate different techniques for solving the task at hand, first run the code in Listing 8-3, which creates and populates the tables MyOrders, MyCustomers, StageCusts, and StageOrders.

Listing 8-3: Create and populate sample tables

```
USE tempdb;
GO
IF OBJECT_ID('dbo.MyOrders') IS NOT NULL
  DROP TABLE dbo.MyOrders;
GO
IF OBJECT_ID('dbo.MyCustomers') IS NOT NULL
  DROP TABLE dbo.MyCustomers;
GO
```

```
IF OBJECT_ID('dbo.StageCusts') IS NOT NULL
  DROP TABLE dbo.StageCusts;
GO
IF OBJECT_ID('dbo.StageOrders') IS NOT NULL
  DROP TABLE dbo.StageOrders;
GO

SELECT *
INTO dbo.MyCustomers
FROM Northwind.dbo.Customers
WHERE CustomerID < N'M';

ALTER TABLE dbo.MyCustomers ADD PRIMARY KEY(CustomerID);

SELECT *
INTO dbo.MyOrders
FROM Northwind.dbo.Orders
WHERE CustomerID < N'M';

ALTER TABLE dbo.MyOrders ADD
  PRIMARY KEY(OrderID),
  FOREIGN KEY(CustomerID) REFERENCES dbo.MyCustomers;

SELECT *
INTO dbo.StageCusts
FROM Northwind.dbo.Customers;

ALTER TABLE dbo.StageCusts ADD PRIMARY KEY(CustomerID);

SELECT C.CustomerID, CompanyName, ContactName, ContactTitle,
  Address, City, Region, PostalCode, Country, Phone, Fax,
  OrderID, EmployeeID, OrderDate, RequiredDate, ShippedDate,
  ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion,
  ShipPostalCode, ShipCountry
INTO dbo.StageOrders
FROM Northwind.dbo.Customers AS C
  JOIN Northwind.dbo.Orders AS O
    ON O.CustomerID = C.CustomerID;

CREATE UNIQUE CLUSTERED INDEX idx_cid_oid
  ON dbo.StageOrders(CustomerID, OrderID);
ALTER TABLE dbo.StageOrders ADD PRIMARY KEY NONCLUSTERED(OrderID);
```

Let's start with the simplest scenario. You just imported some updated and new customer data into the staging table StageCusts. You now need to add to MyCustomers any customers in StageCusts that are not already in MyCustomers. There are no duplicate customers in the source data. The solution is to simply use the NOT EXISTS predicate to verify that you're inserting rows from StageCusts with keys that do not yet exist in MyCustomers as follows:

```
INSERT INTO dbo.MyCustomers(CustomerID, CompanyName, ContactName,
     ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax)
   SELECT CustomerID, CompanyName, ContactName,
     ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax
   FROM dbo.StageCusts AS S
   WHERE NOT EXISTS
     (SELECT * FROM dbo.MyCustomers AS T
     WHERE T.CustomerID = S.CustomerID);
```

Now suppose you're not given the StageCusts table; rather, you're given a StageOrders table that contains both order and customer data in a denormalized form. A new customer might appear in many StageOrders rows but must be inserted only once to MyCustomers. The techniques available to you to isolate only one row for each customer depend on whether all customer attributes are guaranteed to be duplicated identically, or whether there might be differences in the non-key attributes (for example, the format of phone numbers). If rows with the same *CustomerID* are guaranteed to have the same values in all other customer attributes, you can use a NOT EXISTS query similar to the one I showed earlier, adding a DISTINCT clause to the customer attributes you query from the StageOrders table:

```
INSERT INTO dbo.MyCustomers(CustomerID, CompanyName, ContactName,
     ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax)
   SELECT DISTINCT CustomerID, CompanyName, ContactName,
```

```
    ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax
  FROM dbo.StageOrders AS S
  WHERE NOT EXISTS
    (SELECT * FROM dbo.MyCustomers AS T
    WHERE T.CustomerID = S.CustomerID);
```

If customer attributes other than *CustomerID* might vary among rows with the same *CustomerID*, you will need to isolate only one row per customer. Naturally, DISTINCT won't work in such a case because it eliminates only completely identical row duplicates. Furthermore, the technique using DISTINCT requires a full scan of the source table, so it's slow. You can use the source table's key (*OrderID* in our case) to identify a single row per customer, because the key is unique. For example, you can use a subquery returning the minimum *OrderID* for the outer customer:

```
INSERT INTO dbo.MyCustomers(CustomerID, CompanyName, ContactName,
    ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax)
  SELECT CustomerID, CompanyName, ContactName,
    ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax
  FROM dbo.StageOrders AS S
  WHERE NOT EXISTS
    (SELECT * FROM dbo.MyCustomers AS T
    WHERE T.CustomerID = S.CustomerID)
    AND S.OrderID = (SELECT MIN(OrderID) FROM dbo.StageOrders AS S2
                     WHERE S2.CustomerID = S.CustomerID);
```

In SQL Server 2005, you can rely on the ROW_NUMBER function to get the fastest solution available among the ones that I demonstrated:

```
INSERT INTO dbo.MyCustomers(CustomerID, CompanyName, ContactName,
    ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax)
  SELECT CustomerID, CompanyName, ContactName,
    ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax
  FROM (SELECT
          ROW_NUMBER() OVER(PARTITION BY CustomerID ORDER BY OrderID) AS rn,
          CustomerID, CompanyName, ContactName, ContactTitle, Address, City,
          Region, PostalCode, Country, Phone, Fax
        FROM dbo.StageOrders) AS S
  WHERE NOT EXISTS
    (SELECT * FROM dbo.MyCustomers AS T
    WHERE T.CustomerID = S.CustomerID)
    AND rn = 1;
```

The query calculates row numbers partitioned by *CustomerID*, based on *OrderID* ordering, and isolates only rows for new customers, with a row number that is equal to 1. This way you isolate for each new customer only the row with the minimum *OrderID*.

INSERT with OUTPUT

SQL Server 2005 introduces support for returning output from a data modification statement via a new OUTPUT clause. I like to think of this feature as "DML with Results." The OUTPUT clause is supported for INSERT, DELETE, and UPDATE statements. In the OUTPUT clause, you can refer to the special tables *inserted* and *deleted*. These special tables contain the rows affected by the data modification statement—in their new, or after-modification and old, or before-modification versions, respectively. You use the inserted and deleted tables here much like you do in triggers. I will cover triggers at length in *Inside T-SQL Programming*; for now, it suffices to say that the inserted and deleted tables in a trigger hold the new and old images of the rows modified by the statement that fired the trigger. With INSERTs, you refer to the inserted table to identify attributes from the new rows. With DELETEs, you refer to the deleted table to identify attributes from the old rows. With UPDATEs, you refer to the deleted table to identify the attributes from the updated rows before the change, and you refer to the inserted table to identify the attributes from the updated rows after the change. The target of the output can be the caller (client application), a table, or even both. The feature is probably best explained through examples. In this section, I'll give an INSERT example, and later in the chapter I'll also provide DELETE and UPDATE examples.

An example of an INSERT statement in which the OUTPUT clause can be very handy is when you issue a multirow INSERT into a table with an identity column and want to know what the new identity values are. With single-row INSERTs, it's not a problem: the SCOPE_IDENTITY function provides the last identity value generated by your session in the current scope. But for a multirow INSERT statement, how do you find the new identity values? You use the OUTPUT clause and direct the new identity values back to the caller or into some target (for example, a table variable).

To demonstrate this technique, first run the following code, which generates the Customers-Dim table:

```
USE tempdb;
GO
IF OBJECT_ID('dbo.CustomersDim') IS NOT NULL
DROP TABLE dbo.CustomersDim;
GO

CREATE TABLE dbo.CustomersDim
(
  KeyCol      INT           NOT NULL IDENTITY PRIMARY KEY,
  CustomerID  NCHAR(5)      NOT NULL,
  CompanyName NVARCHAR(40)  NOT NULL
  /* ... other columns ... */
);
```

Imagine that this table represents a customer dimension in your data warehouse. You now need to insert into the CustomersDim table the UK customers from the Customers table in the Northwind database. Notice that the target has an identity column called *KeyCol* that contains surrogate keys for customers. I won't get into the reasoning behind the common use of surrogate keys in dimension tables in data warehouses (as opposed to relying on natural keys only); that's not the focus of my discussion here. I just want to demonstrate a technique that uses the OUTPUT clause. Suppose that after each insert you need to do some processing of the newly added customers and identify which surrogate key was assigned to each customer.

The following code declares a table variable (*@NewCusts*), issues an INSERT statement inserting UK customers into *CustomersDim* and directing the new *CustomerID* and *KeyCol* values into *@NewCusts*, and queries the table variable:

```
DECLARE @NewCusts TABLE
(
   CustomerID NCHAR(5) NOT NULL PRIMARY KEY,
   KeyCol INT NOT NULL UNIQUE
);

INSERT INTO dbo.CustomersDim(CustomerID, CompanyName)
    OUTPUT inserted.CustomerID, inserted.KeyCol
    INTO @NewCusts
    -- OUTPUT inserted.CustomerID, inserted.KeyCol
  SELECT CustomerID, CompanyName
  FROM Northwind.dbo.Customers
  WHERE Country = N'UK';

SELECT CustomerID, KeyCol FROM @NewCusts;
```

This code generates the output shown in Table 8-1, where you can see the new identity values in the column *KeyCol*.

Table 8-1: Contents of @NewCusts Table Variable

CustomerID	KeyCol
AROUT	1
BSBEV	2
CONSH	3
EASTC	4
ISLAT	5
NORTS	6
SEVES	7

Notice the commented second OUTPUT clause in the code, which isn't followed by an INTO clause. Uncomment it if you also want to send the output to the caller; you will have two OUTPUT clauses in the INSERT statement.

Sequence Mechanisms

Sequence mechanisms produce numbers that you usually use as keys. SQL Server provides a sequencing mechanism via the IDENTITY column property. The IDENTITY property has several limitations that might cause you to look for an

alternative sequencing mechanism. In this section, I'll describe some of these limitations and alternative mechanisms to generate keys—some that use built-in features, such as global unique identifiers (GUIDs), and some that you can develop yourself.

Identity Columns

The IDENTITY property can be convenient when you want SQL Server to generate single column keys in a table. To guarantee uniqueness, create a PRIMARY KEY or UNIQUE constraint on the identity column. Upon INSERT, SQL Server increments the table's identity value and stores it in the new row.

However, the IDENTITY property has several limitations that might make it an impractical sequencing mechanism for some applications.

One limitation is that the IDENTITY property is table dependent. It's not an independent sequencing mechanism that assigns new values that you can then use in any manner you like. Imagine that you need to generate sequence values that will be used as keys that cannot conflict across tables.

Another limitation is that an identity value is generated when an INSERT statement is issued, not before. There might be cases where you need to generate the new sequence value and then use it in an INSERT statement, and not the other way around.

Another aspect of the IDENTITY property that can be considered a limitation in some cases is that identity values are assigned in an asynchronous manner. This means that multiple sessions issuing multirow inserts might end up getting nonsequential identity values. Moreover, the assignment of a new identity value is not part of the transaction in which the INSERT was issued. These facts have several implications. SQL Server will increment the table's identity value regardless of whether the insert succeeds or fails. You might end up with gaps in the sequence that were not generated by deletions. Some systems cannot allow missing values that cannot be accounted for (for example, invoicing systems). Try telling the Internal Revenue Service (IRS) that some of the missing invoice IDs in your system are a result of the asynchronous manner in which identity values are managed.

Custom Sequences

I'll suggest a couple of solutions to the problem of maintaining a custom sequencing mechanism. I'll show both synchronous and asynchronous solutions.

Synchronous Sequence Generation You need a synchronous sequence generator when you must account for all values in the sequence. The classic scenario for such a sequence is generating invoice numbers. The way to guarantee that no gaps occur is to lock the sequence resource when you need to increment it and release the lock only when the transaction is finished. If you think about it, that's exactly how exclusive locks behave when you modify data in a transaction—that is, a lock is acquired to modify data, and it's released when the transaction is finished (committed or rolled back). To maintain such a sequence, create a table with a single row and a single column holding the last sequence value used. Initially, populate it with a zero if you want the first value in the sequence to be 1:

```
USE tempdb;
GO
IF OBJECT_ID('dbo.SyncSeq') IS NOT NULL
  DROP TABLE dbo.SyncSeq;
GO

CREATE TABLE dbo.SyncSeq(val INT);
INSERT INTO dbo.SyncSeq VALUES(0);
```

Now that the sequence table is in place, I'll describe how you get a single sequence value or a block of sequence values at once.

Single Sequence Value To get a single sequence value, you increment the sequence value by 1 and return the resulting value. You can achieve this by beginning a transaction, modifying the sequence value, and then retrieving it. Or you can both increment and retrieve the new sequence value in a single atomic operation using a specialized UPDATE syntax. Run the following code to create a stored procedure that uses the specialized T-SQL UPDATE syntax, increments the sequence value, and returns the new value as an output parameter:

```
IF OBJECT_ID('dbo.usp_SyncSeq') IS NOT NULL
  DROP PROC dbo.usp_SyncSeq;
GO
```

```
CREATE PROC dbo.usp_SyncSeq
  @val AS INT OUTPUT
AS
UPDATE dbo.SyncSeq
  SET @val = val = val + 1;
GO
```

The assignment *SET @val = val = val + 1* is equivalent to *SET val = val + 1, @val = val + 1*. Note that SQL Server will first lock the row exclusively and then increment *val*, retrieve it, and release the lock only when the transaction is completed.

Whenever you need a new sequence value, use the following code:

```
DECLARE @key AS INT;
EXEC dbo.usp_SyncSeq @val = @key
OUTPUT; SELECT @key;
```

To reset the sequence—for example, when the sequence value is about to overflow—set its value to zero:

```
UPDATE dbo.SyncSeq SET val = 0;
```

Block of Sequence Values If you want a mechanism to allocate a block of sequence values all at once, you need to slightly alter the stored procedure's implementation as follows:

```
ALTER PROC dbo.usp_SyncSeq
  @val AS INT OUTPUT,
  @n AS INT = 1
AS
UPDATE dbo.SyncSeq
  SET @val = val + 1, val = val + @n;
GO
```

In the additional argument (*@n*) you specify the block size (how many sequence values you need). The stored procedure increments the current sequence value by *@n* and returns the first value in the block via the *@val* output parameter. This procedure allocates the block of sequence values from *@val* to *@val + @n − 1*.

The following code provides an example of acquiring and using a whole block of sequence values:

```
IF OBJECT_ID('tempdb..#CustsStage') IS NOT NULL
  DROP TABLE #CustsStage
GO

DECLARE @key AS INT, @rc AS INT;

SELECT CustomerID, 0 AS KeyCol
INTO #CustsStage
FROM Northwind.dbo.Customers
WHERE Country = N'UK';

SET @rc = @@rowcount;
EXEC dbo.usp_SyncSeq @val = @key OUTPUT, @n = @rc;

SET @key = @key -1;
UPDATE #CustsStage SET @key = KeyCol = @key + 1;

SELECT CustomerID, KeyCol FROM #CustsStage;
```

This technique is an alternative to the one shown earlier that used the IDENTITY property to generate surrogate keys for UK customers. This code uses a SELECT INTO statement to insert UK customers into a temporary table called #CustsStage, temporarily assigning 0 as the *Key-Col* value in all target rows. The code then stores the number of affected rows (*@@rowcount*) in the variable *@rc*. Next, the code invokes the *usp_SyncSeq* procedure to request a block of a size *@rc* of new sequence values. The stored procedure stores the first sequence value from the block in the variable *@key* through the output parameter *@val*. Next, the code subtracts 1 from *@key* and invokes a specialized T-SQL UPDATE statement to assign the block of sequence values. The UPDATE makes a single pass over the rows in #CustStage. With every row that the UPDATE visits, it stores the value of *@key + 1* in *KeyCol* and in *@key*. This means that with every new row visited, *@key* is incremented by one and stored in *KeyCol*. You basically distribute the new block of sequence values among the rows in #CustStage. If you run this code after resetting the sequence value to 0, as instructed earlier, #CustStage will contain seven UK customers, with *KeyCol* values ranging from 1 through 7. Run this code multiple times to

see how you get a new block of sequence values every time (1 through 7, 8 through 14, 15 through 21, and so on).

The specialized T-SQL UPDATE statement is not standard and doesn't guarantee it will access the rows in #CustStage in any particular order. Therefore, by using it, you cannot control the order in which SQL Server will assign the block of sequence values. There's a technique for assigning the block of sequence values where you can control the order of assignment, but it is new to SQL Server 2005. Substitute the specialized UPDATE statement just shown with an UPDATE against a CTE that calculates row numbers based on a desired order (for example, *CustomerID*) as follows:

```
WITH CustsStageRN AS
(
   SELECT KeyCol, ROW_NUMBER() OVER(ORDER BY CustomerID) AS RowNum
   FROM #CustsStage
)
UPDATE CustsStageRN SET KeyCol = RowNum + @key;
```

You can use similar techniques when you just want to assign a sequence of unique values starting with 1 (and increasing by 1 from there) that is unrelated to any existing sequence. I will describe such techniques later in the chapter in the "Assignment UPDATE" section.

So far, I demonstrated inserting a result set into a target table and modifying the rows with new sequence values. Instead, you can use a SELECT INTO statement to populate a temporary table with the target rows, and also to assign row numbers starting with 1 and increasing by 1 from there. (Let's call the column *rn*.) To generate the row numbers, you can use the IDENTITY function in SQL Server 2000 and the ROW_NUMBER function in SQL Server 2005. Right after the insert, store the *@@rowcount* value in a local variable (*@rc*) and invoke the procedure *usp_SyncSeq* to increment the sequence by *@rc* and assign the first sequence value in the new block to your variable (*@key*). Finally, query the temporary table, calculating the new sequence values as *rn* + *@key* − *1*. Here's the complete code sample demonstrating this technique:

```
IF OBJECT_ID('tempdb..#CustsStage') IS NOT NULL
  DROP TABLE #CustsStage
GO

DECLARE @key AS INT, @rc AS INT;

SELECT CustomerID, IDENTITY(int, 1, 1) AS rn
  -- In 2005 can use ROW_NUMBER() OVER(ORDER BY CustomerID)
INTO #CustsStage
FROM Northwind.dbo.Customers
WHERE Country = N'UK';

SET @rc = @@rowcount;
EXEC dbo.usp_SyncSeq @val = @key OUTPUT, @n = @rc;

SELECT CustomerID, rn + @key - 1 AS KeyCol FROM #CustsStage;
```

Run the code several times to see how you get a new block of sequence values every time.

Asynchronous Sequence Generation The synchronous sequencing mechanism doesn't allow gaps, but it might cause concurrency problems. Remember that you must exclusively lock the sequence to increment it, and then you must maintain the lock until the transaction finishes. The longer the transaction is, the longer you lock the sequence. Obviously, this solution can cause queues of processes waiting for the sequence resource to be released. But there's not much you can do if you want to maintain a synchronous sequence.

However, there are cases where you might not care about having gaps. For example, suppose that all you need is a key generator that will guarantee that you will not generate the same key twice. Say that you need those keys to uniquely identify rows across tables. You don't want the sequence resource to be locked for the duration of the transaction. Rather, you want the sequence to be locked for a fraction of a second while incrementing it, just to prevent multiple processes from getting the same value. In other words, you need an asynchronous sequence, one that will work much faster than the synchronous one, allowing better concurrency.

One option that would address these requirements is to use built-in functions that SQL Server provides you to generate GUIDs. I'll discuss this option shortly. However, GUIDs are long (16 bytes). You might prefer to use integer sequence values, which are substantially smaller (4 bytes). To achieve such an asynchronous sequencing mechanism, you create a table (AsyncSeq) with an identity column as follows:

```
USE tempdb;
```

```
GO
IF OBJECT_ID('dbo.AsyncSeq') IS NOT NULL
  DROP TABLE dbo.AsyncSeq;
GO
CREATE TABLE dbo.AsyncSeq(val INT IDENTITY(1,1));
```

Create the following *usp_AsyncSeq* procedure to generate a new sequence value and return it through the *@val* output parameter:

```
IF OBJECT_ID('dbo.usp_AsyncSeq') IS NOT NULL
  DROP PROC dbo.usp_AsyncSeq;
GO

CREATE PROC dbo.usp_AsyncSeq
  @val AS INT OUTPUT
AS
BEGIN TRAN
  SAVE TRAN S1;
  INSERT INTO dbo.AsyncSeq DEFAULT VALUES;
  SET @val = SCOPE_IDENTITY()
  ROLLBACK TRAN S1;
COMMIT TRAN
GO
```

The procedure opens a transaction just for the sake of creating a save point called S1. It inserts a new row to AsyncSeq, which generates a new identity value in the AsyncSeq table and stores it in the *@val* output parameter. The procedure then rolls back the INSERT. But a rollback doesn't undo a variable assignment, nor does it undo incrementing the identity value. Plus, the identity resource is not locked for the duration of an outer transaction; rather, it's locked only for a fraction of a second to increment. This behavior of the IDENTITY property is crucial for maintaining an asynchronous sequence.

> **Note** As of the date of writing this, I haven't found any official documentation from Microsoft that describes this behavior of the IDENTITY property.

Rolling back to a save point ensures that the rollback will not have any effect on an external transaction. The rollback prevents the AsyncSeq table from growing. In fact, it will never contain any rows from calls to *usp_AsyncSec*.

Whenever you need the next sequence value, run the *usp_AsyncSeq*, just like you did with the synchronous one:

```
DECLARE @key AS INT;
EXEC dbo.usp_AsyncSeq @val = @key
OUTPUT; SELECT @key;
```

Only this time, the sequence will not block if you increment it within an external transaction. This asynchronous sequence solution can generate only one sequence value at a time.

If you want to reset the sequence value, you can do one of two things. You can truncate the table, which resets the identity value:

```
TRUNCATE TABLE dbo.AsyncSeq;
```

Or you can issue the DBCC CHECKIDENT statement with the RESEED option, as follows:

```
DBCC CHECKIDENT('dbo.AsyncSeq', RESEED, 0);
```

Globally Unique Identifiers

SQL Server provides you with the NEWID function, which generates a new globally unique identifier (GUID) every time it is invoked. The function returns a 16-byte value typed as UNIQUEIDENTIFIER. If you need an automatic mechanism that assigns unique keys in a table, or even across different tables, you can create a UNIQUEIDENTIFIER column with the default value *NEWID*. The downside of a UNIQUEIDENTIFIER column used as a key is that it's pretty big—16 bytes. This, of course, has an impact on index sizes, join performance, and so on.

Note that the NEWID function does not guarantee that a newly generated GUID will be greater than any previously generated one in the same computer. If you need such a guarantee, use the new NEWSEQUENTIALID function, introduced in SQL Server 2005 particularly for this purpose.

Deleting Data

In this section, I'll cover different aspects of deleting data, including TRUNCATE vs. DELETE, removing rows with duplicate data, DELETE using joins, large DELETEs, and DELETE with OUTPUT.

TRUNCATE vs. DELETE

If you need to remove all rows from a table, use TRUNCATE TABLE and not DELETE without a WHERE clause. DELETE is always fully logged, and with large tables it can take a while to complete. TRUNCATE TABLE is always minimally logged, regardless of the recovery model of the database, and therefore, it is always significantly faster than a DELETE. Note though, that TRUNCATE TABLE will not fire any DELETE triggers on the table. To give you a sense of the difference, using TRUNCATE TABLE to clear a table with millions of rows can take a matter of seconds, while clearing the table with DELETE can take a few hours.

> **Tip** SQL Server will reject DROP TABLE attempts if there's a schema-bound object pointing to the target table. It will reject both DROP TABLE and TRUNCATE TABLE attempts if there's a foreign key pointing to the target table. This limitation applies even when the foreign table is empty, and even when the foreign key is disabled. If you want to prevent accidental TRUNCATE TABLE and DROP TABLE attempts against sensitive production tables, simply create dummy tables with foreign keys pointing to them.

In addition to the substantial performance difference between TRUNCATE TABLE and DELETE, there's also a difference in the way they handle the IDENTITY property. TRUNCATE TABLE resets the IDENTITY property to its original seed, while DELETE doesn't.

Removing Rows with Duplicate Data

Duplicates typically arise from inadequate data-integrity enforcement. If you don't create a PRIMARY KEY or UNIQUE constraint where you need one, you can expect to end up with unwanted duplicate data. Data integrity is important; enforce it through constraints and design applications to protect your data.

Having said that, if you have duplicates in your data and need to get rid of them, there are a number of techniques that you can use. Your options and the efficiency of the techniques depend on several factors. One is the number of duplicates with respect to unique rows (density). Another is whether the whole row is guaranteed to be duplicated, or whether you can rely only on the set of attributes that will form the key once duplicates are removed. For example, suppose you have a table containing information about orders. A primary key was not created on the *OrderID* column, and several *OrderID* values appear more than once. You need to keep only one row per each unique *OrderID* value.

To demonstrate techniques to remove duplicates, run the following code, which creates the OrdersDups table and populates it with 83,000 rows containing many duplicates:

```
USE tempdb;
GO
IF OBJECT_ID('dbo.OrdersDups') IS NOT NULL
  DROP TABLE dbo.OrdersDups
GO

SELECT OrderID+0 AS OrderID, CustomerID, EmployeeID, OrderDate,
  RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress,
  ShipCity, ShipRegion, ShipPostalCode, ShipCountry
INTO dbo.OrdersDups
FROM Northwind.dbo.Orders, dbo.Nums
WHERE n <= 100;
```

Remember that the Nums table is an auxiliary table of numbers. I discussed it in detail in Chapter 4.

Rerun this code before testing each technique so that you have common base data in all your tests.

Before I cover set-based techniques, I should first mention that you could use a cursor, scanning the rows in the order of the attributes that determine duplicates (*OrderID* in the Orders-Dups case) and deleting all rows that you identify as duplicates. Just remember that such a cursor involves a lot of overhead and will typically be slower than set-based solutions, especially when the density of duplicates is high (a lot of duplicates). Though, with a low density of duplicates, it might be worthwhile to compare the cursor-based solution to the set-based ones, especially when there's no way to uniquely identify a row.

Let's start with a technique that you can apply when complete rows are duplicated and you have a high density of duplicates. Use a SELECT DISTINCT … INTO statement to copy distinct rows to a new table. Drop the original table,

rename the new one to the original table's name, and then create all constraints, indexes, and triggers:

```
SELECT DISTINCT * INTO dbo.OrdersTmp FROM dbo.OrdersDups;
DROP TABLE dbo.OrdersDups;
EXEC sp_rename 'dbo.OrdersTmp', 'OrdersDups';
-- Add constraints, indexes, triggers
```

This code runs for about 5 seconds on my system against the sample data I provided. The nice thing about this technique is that it doesn't require an existing unique identifier in the table.

Some other techniques require an existing unique identifier. If you already have one, your solution will not incur the cost involved with adding such a column. If you don't, you can add an identity column for this purpose, but adding such a column can take a while. If you have an existing numeric column that you can overwrite, you can use one of the techniques that I will show later to assign a sequence of values to an existing column. The process of overwriting an existing numeric column is substantially faster because it doesn't involve the restructuring and expansion of rows.

To allow a fast duplicate removal process, you want an index on the attributes that determine duplicates (*OrderID*) plus the unique identifier (call it *KeyCol*). Here's the code you need to run to add an identity column to the OrdersDups table and create the desired index:

```
ALTER TABLE dbo.OrdersDups
  ADD KeyCol INT NOT NULL IDENTITY;
CREATE UNIQUE INDEX idx_OrderID_KeyCol
  ON dbo.OrdersDups(OrderID, KeyCol);
```

Then use the following DELETE statement to get rid of the duplicates:

```
DELETE FROM dbo.OrdersDups
WHERE EXISTS
  (SELECT *
  FROM dbo.OrdersDups AS O2
  WHERE O2.OrderID = dbo.OrdersDups.OrderID
  AND O2.KeyCol > dbo.OrdersDups.KeyCol);
```

This statement deletes all orders for which another order can be found with the same *OrderID* and a higher *KeyCol*. If you think about it, you will end up with one row for each *OrderID*—the one with the highest *KeyCol*. This technique runs for 14 seconds on my system, including adding the identity column and creating the index.

One advantage this technique has over the previous DISTINCT technique is that you rely only on the attributes that determine duplicates (*OrderID*) and the surrogate key (*KeyCol*). It works even when other attributes among the redundant rows with the same *OrderID* value are not equal. However, this technique can be very slow when there's a high density of duplicates. To optimize the solution in a high-density scenario, you can use similar logic to the last solution; that is, keep rows with the maximum *KeyCol* per *OrderID*. But insert those unique rows into a new table using a SELECT INTO statement. You can then get rid of the original table; rename the new table to the original table name; and re-create all indexes, constraints, and triggers. Here's the code that applies this approach, which ran for 2 seconds on my system in total:

```
ALTER TABLE dbo.OrdersDups
  ADD KeyCol INT NOT NULL IDENTITY;
CREATE UNIQUE INDEX idx_OrderID_KeyCol
  ON dbo.OrdersDups(OrderID, KeyCol);
GO

SELECT O.OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate,
  ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity,
  ShipRegion, ShipPostalCode, ShipCountry
INTO dbo.OrdersTmp

FROM dbo.OrdersDups AS O
  JOIN (SELECT OrderID, MAX(KeyCol) AS mx
       FROM dbo.OrdersDups
       GROUP BY OrderID) AS U
    ON O.OrderID = O.OrderID
    AND O.KeyCol = U.mx;

DROP TABLE dbo.OrdersDups;
EXEC sp_rename 'dbo.OrdersTmp', 'OrdersDups';
-- Recreate constraints, indexes
```

You've seen several solutions to deleting rows with duplicate values, and I recommended the scenarios where I find that each is adequate. But all of them were limited in one way or another. The DISTINCT technique requires equivalence of complete rows among duplicates, and the other two techniques require a unique identifier in the table. In SQL Server 2005, you can use a CTE and the ROW_NUMBER function to generate a fast solution without these shortcomings:

```
WITH Dups AS
(
  SELECT *,
    ROW_NUMBER() OVER(PARTITION BY OrderID ORDER BY OrderID) AS rn
  FROM dbo.OrdersDups
)
DELETE FROM Dups WHERE rn > 1;
```

The query defining the CTE Dups generates row numbers starting with 1 for each partition of rows with the same *OrderID*, meaning that each set of rows with the same *OrderID* value will be assigned row numbers starting with 1 independently. Each row number here represents the duplicate number. The ROW_NUMBER function requires you to specify an ORDER BY clause, even when you don't really care how row numbers are assigned within each partition. You can specify the same column you use in the PARTITION BY clause (*OrderID*) also in the ORDER BY clause. Such an ORDER BY clause will have no effect on the assignment of row numbers within each partition. More importantly, while the row numbering is nondeterministic, there will be exactly one row within each partition with *rn* equal to 1.

Finally, the outer query simply deletes rows that have a duplicate number greater than 1 through the CTE, leaving only one row for each *OrderID* value.

This solution runs for only 1 second on my system; it doesn't require a unique identifier in the table; and it allows you to identify duplicates based on any attribute or attributes that you like.

DELETE Using Joins

T-SQL supports a proprietary syntax for DELETE and UPDATE based on joins. Here I'll cover DELETEs based on joins, and later, in the UPDATE section, I'll cover UPDATEs based on joins.

> **Note** This syntax is not standard and should be avoided unless there's a compelling benefit over the standard syntax, as I will describe in this section.

I'll first describe the syntax, and then show examples where it provides functionality not available with standard syntax.

You write a DELETE based on a join in a similar manner to writing a SELECT based on a join. You substitute the SELECT clause with a DELETE FROM <*target_table*>, where <*target_table*> is the table from which you want to delete rows. Note that you should specify the table alias if one was provided.

The typical use of this feature is to make it easier to delete rows that meet an EXISTS or NOT EXISTS condition, to avoid having to specify a subquery for the matching condition twice. Some people also like the fact that it allows you to write a SELECT query first, and then change SELECT to DELETE.

As an example of how a SELECT join query and a DELETE join query are similar, here's a query that returns order details for orders placed on or after May 6, 1998:

```
USE Northwind;

SELECT OD.*
FROM dbo.[Order Details] AS OD
  JOIN dbo.Orders AS O
    ON OD.OrderID = O.OrderID
WHERE O.OrderDate >= '19980506';
```

If you want to delete order details for orders placed on or after May 6, 1998, simply replace *SELECT OD.** in the preceding query with *DELETE FROM OD*:

```
BEGIN TRAN

DELETE FROM OD
FROM dbo.[Order Details] AS OD
  JOIN dbo.Orders AS O
    ON OD.OrderID = O.OrderID
WHERE O.OrderDate >= '19980506';
```

```
ROLLBACK TRAN
```

In some of my examples, I use a transaction and roll back the modification so that you can try out the examples without permanently modifying the sample tables. This particular nonstandard DELETE query can be rewritten as a standard one using a subquery:

```
BEGIN TRAN

DELETE FROM dbo.[Order Details]
WHERE EXISTS

   (SELECT *
     FROM dbo.Orders AS O
     WHERE O.OrderID = dbo.[Order Details].OrderID
       AND O.OrderDate >= '19980506');

ROLLBACK TRAN
```

In this case, the nonstandard DELETE has no advantage over the standard one—neither in performance nor in simplicity, so I don't see any point in using it. However, you will find cases in which it is hard to get by without using the proprietary syntax. For example, suppose you need to delete from a table variable, and you must refer to the table variable from a subquery. T-SQL doesn't support qualifying a column name with a table variable name.

The following code declares a table variable called *@MyOD* and populates it with some order details, identified by (*OrderID, ProductID*). The code then attempts to delete all rows from *@MyOD* with keys that already appear in the Order Details table:

```
DECLARE @MyOD TABLE
(
   OrderID   INT NOT NULL,
   ProductID INT NOT NULL,
   PRIMARY KEY(OrderID, ProductID)
);

INSERT INTO @MyOD VALUES(10001, 14);
INSERT INTO @MyOD VALUES(10001, 51);
INSERT INTO @MyOD VALUES(10001, 65);
INSERT INTO @MyOD VALUES(10248, 11);
INSERT INTO @MyOD VALUES(10248, 42);

DELETE FROM @MyOD
WHERE EXISTS
    (SELECT * FROM dbo.[Order Details] AS OD
     WHERE OD.OrderID = @MyOD.OrderID
      AND OD.ProductID = @MyOD.ProductID);
```

This code fails with the following error:

```
Msg 137, Level 15, State 2, Line 17
Must declare the scalar variable "@MyOD".
```

Essentially, the reason for the failure is that T-SQL doesn't support qualifying a column name with a table variable name. Moreover, T-SQL doesn't allow you to alias the target table directly; rather, it requires you to do so via a second FROM clause like so:

```
DELETE FROM MyOD
FROM @MyOD AS MyOD
WHERE EXISTS
   (SELECT * FROM dbo.[Order Details] AS OD
     WHERE OD.OrderID = MyOD.OrderID
       AND OD.ProductID = MyOD.ProductID);
```

Note If you want to test this code, make sure you run it right after declaring and populating the table variable in the same batch. Otherwise, you will get an error saying that the variable *@MyOD* was not declared. Like any other variable, the scope of a table variable is the local batch.

Another solution is to use a join instead of the subquery, where you can also alias tables:

```
DELETE FROM MyOD
```

```
FROM @MyOD AS MyOD
   JOIN dbo.[Order Details] AS OD
     ON OD.OrderID = MyOD.OrderID
    AND OD.ProductID = MyOD.ProductID;
```

In SQL Server 2005, you can use a CTE as an alternative to aliasing the table variable, allowing a simpler solution:

```
WITH MyOD AS (SELECT * FROM @MyOD)
DELETE FROM MyOD
WHERE EXISTS
    (SELECT * FROM dbo.[Order Details] AS OD
      WHERE OD.OrderID = MyOD.OrderID
        AND OD.ProductID = MyOD.ProductID);
```

CTEs are extremely useful in other scenarios where you need to modify data in one table based on data that you inspect in another. It allows you to simplify your code and, in many cases, avoid relying on modification statements that use joins.

DELETE with OUTPUT

In Chapter 7, I described a technique to delete large volumes of data from an existing table in batches to avoid log explosion and lock escalation problems. Here I will show how you can use the new OUTPUT clause to archive the data that you purge. To demonstrate the technique, first run the following code, which creates the LargeOrders tables and populates it with a little over two million orders placed in years 2000 through 2006:

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.LargeOrders') IS NOT NULL
  DROP TABLE dbo.LargeOrders;
GO
SELECT IDENTITY(int, 1, 1) AS OrderID, CustomerID, EmployeeID,
  DATEADD(day, n-1, '20000101') AS OrderDate,
  CAST('a' AS CHAR(200)) AS Filler
INTO dbo.LargeOrders
FROM Northwind.dbo.Customers AS C,
  Northwind.dbo.Employees AS E,
  dbo.Nums
WHERE n <= DATEDIFF(day, '20000101', '20061231') + 1;

CREATE UNIQUE CLUSTERED INDEX idx_od_oid
  ON dbo.LargeOrders(OrderDate, OrderID);

ALTER TABLE dbo.LargeOrders ADD PRIMARY KEY NONCLUSTERED(OrderID);
```

> **Note** It should take the code a few minutes to run, and it will require about a gigabyte of space in your tempdb database. Also, the code refers to the Nums auxiliary table, which was covered in Chapter 4.

As a reminder, you would use the following technique to delete all rows with an *OrderDate* older than 2001 in batches of 5000 rows (but don't run it yet):

```
WHILE 1 = 1
BEGIN
  DELETE TOP (5000) FROM dbo.LargeOrders WHERE OrderDate < '20010101';
  IF @@rowcount < 5000 BREAK;
END
```

Remember that in SQL Server 2005 you use DELETE TOP instead of the older SET ROWCOUNT option. Earlier in the chapter, I introduced the SQL Server 2005 support for the new OUTPUT clause, which allows you to return output from a statement that modifies data. Remember that you can direct the output to a temporary or permanent table, to a table variable, or back to the caller. I showed an example using an INSERT statement, and here I will show one using a DELETE statement. Suppose you wanted to enhance the solution that purges historic data in batches by also archiving the data that you purge. Run the following code to create the OrdersArchive table, where you will store the archived orders:

```
CREATE TABLE dbo.OrdersArchive
(
   OrderID          INT      NOT NULL PRIMARY KEY NONCLUSTERED,
   CustomerID       NCHAR(5) NOT NULL,
   EmployeeID       INT NOT  NULL,
   OrderDate        DATETIME NOT NULL,
```

```
   Filler CHAR(200) NOT      NULL
);

CREATE UNIQUE CLUSTERED INDEX idx_od_oid
   ON dbo.OrdersArchive(OrderDate, OrderID);
GO
```

Using the new OUTPUT clause, you can direct the deleted rows from each batch into the OrdersArchive table. For example, the following code is the enhanced solution, which purges orders with an *OrderDate* before 2001 in batches and also archives them:

```
WHILE 1=1
BEGIN
  BEGIN TRAN
    DELETE TOP(5000) FROM dbo.LargeOrders
      OUTPUT deleted.* INTO dbo.OrdersArchive
    WHERE OrderDate < '20010101';

      IF @@rowcount < 5000
      BEGIN
        COMMIT TRAN
        BREAK;
      END
    COMMIT TRAN
END
```

Note It should take this code a few minutes to run.

The OrdersArchive table now holds archived orders placed before 2001.

Note When using the OUTPUT clause to direct the output to a table, the table cannot have enabled triggers or CHECK constraints, nor can it participate on either side of a foreign key constraint. If the target table doesn't meet these requirements, you can direct the output to a temporary table or a table variable, and then copy the rows from there to the target table.

There are important benefits to using the OUTPUT clause when you want to archive data that you delete. Without the OUTPUT clause, you need to first query the data to archive it, and then delete it. Such a technique is slower and more complex. To guarantee that new rows matching the filter will not be added between the SELECT and the DELETE (also known as *phantoms*), you must lock the data you archive using a serializable isolation level. With the OUTPUT clause, you will not only get better performance, you won't need to worry about phantoms, as you are guaranteed to get exactly what you deleted back from the OUTPUT clause.

Updating Data

This section covers several aspects of updating data, including UPDATEs using joins, UPDATE with OUTPUT, and SELECT and UPDATE statements that perform assignments to variables.

UPDATE Using Joins

Earlier in this chapter, I mentioned that T-SQL supports a nonstandard syntax for modifying data based on a join, and I showed DELETE examples. Here I'll cover UPDATEs based on joins, focusing on cases where the nonstandard syntax has advantages over the supported standard syntax. I'll show that SQL Server 2005 introduces simpler alternatives that practically eliminate the need for the older UPDATE syntax that uses joins.

I'll start with one of the cases where an UPDATE based on a join had a performance advantages over the standard UPDATE supported by T-SQL. Suppose you wanted to update the shipping information for orders placed by USA customers, overwriting the *ShipCountry*, *ShipRegion*, and *ShipCity* attributes with the customer's *Country*, *Region*, and *City* attributes from the Customers table. You could use one subquery for each of the new attribute values, plus one in the WHERE clause to filter orders placed by USA customers as follows:

```
USE Northwind;

BEGIN TRAN

  UPDATE dbo.Orders
    SET ShipCountry = (SELECT C.Country FROM dbo.Customers AS C
```

```
                        WHERE C.CustomerID = dbo.Orders.CustomerID),
          ShipRegion = (SELECT C.Region FROM dbo.Customers AS C
                        WHERE C.CustomerID = dbo.Orders.CustomerID),
            ShipCity = (SELECT C.City FROM dbo.Customers AS C
                        WHERE C.CustomerID = dbo.Orders.CustomerID)
WHERE CustomerID IN
    (SELECT CustomerID FROM dbo.Customers WHERE Country = 'USA');

ROLLBACK TRAN
```

Again, I'm rolling back the transaction so that the change will not take effect in the Northwind database. Though standard, this technique is very slow. Each such subquery involves separate access to return the requested attribute from the Customers table. I wanted to provide a figure with the graphical execution plan for this UPDATE, but it's just too big! Request a graphical execution plan in SSMS to see for yourself.

You can write an UPDATE based on a join to perform the same task as follows:

```
BEGIN TRAN

   UPDATE O
     SET ShipCountry = C.Country,
         ShipRegion = C.Region,
         ShipCity = C.City
   FROM dbo.Orders AS O
     JOIN dbo.Customers AS C
         ON O.CustomerID = C.CustomerID
     WHERE C.Country = 'USA';

ROLLBACK TRAN
```

This code is shorter and simpler, and the optimizer generates a more efficient plan for it, as you will notice if you request the graphical execution plan in SSMS. You will find in the execution plan that the Customers table is scanned only once, and through that scan, the query processor accesses all the customer attributes it needs. This plan reports half the estimated execution cost of the previous one. In practice, if you compare the two solutions against larger tables, you will find that the performance difference is substantially higher. Alas, the UPDATE with a join technique is nonstandard.

ANSI supports syntax called *row value constructors* that allows you to simplify queries like the one just shown. This syntax allows you to specify vectors of attributes and expressions and eliminates the need to issue a subquery for each attribute separately. The following example shows this syntax:

```
UPDATE dbo.Orders
  SET (ShipCountry, ShipRegion, ShipCity) =
    (SELECT Country, Region, City
     FROM dbo.Customers AS C
     WHERE C.CustomerID = dbo.Orders.CustomerID);
  WHERE C.Country = 'USA';
```

However, T-SQL doesn't yet support row value constructors. Such support would allow for simple standard solutions and naturally also lend itself to good optimization. However, hope is not lost. By using a CTE, you can come up with a simple solution yielding an efficient plan very similar to the one that uses a join UPDATE. Simply create a CTE out of the join, and then UPDATE the CTE like so:

```
BEGIN TRAN;

WITH UPD_CTE AS
(
  SELECT
    O.ShipCountry AS set_Country, C.Country AS get_Country,
    O.ShipRegion  AS set_Region,  C.Region  AS get_Region,
    O.ShipCity    AS set_City,    C.City    AS get_City
  FROM dbo.Orders AS O
    JOIN dbo.Customers AS C
      ON O.CustomerID = C.CustomerID
  WHERE C.Country = 'USA'
)
UPDATE UPD_CTE
  SET set_Country = get_Country,
      set_Region  = get_Country,
      set_City    = get_City;
```

```
ROLLBACK TRAN
```

Note Even though CTEs are defined by ANSI SQL:1999, the DELETE and UPDATE syntax against CTEs implemented in SQL Server 2005 is not standard.

This UPDATE generates an identical plan to the one generated for the UPDATE based on a join.

There's another issue you should be aware of when using the join-based UPDATE. When you modify the table on the "many" side of a one-to-many join, you might end up with a nondeterministic update. To demonstrate the problem, run the following code, which creates the tables Customers and Orders and populates them with sample data:

```
USE tempdb;
GO
IF OBJECT_ID('dbo.Orders') IS NOT NULL
  DROP TABLE dbo.Orders;

IF OBJECT_ID('dbo.Customers') IS NOT NULL
  DROP TABLE dbo.Customers;
GO

CREATE TABLE dbo.Customers
(
   custid VARCHAR(5) NOT NULL PRIMARY KEY,
   qty    INT        NULL
);

INSERT INTO dbo.Customers(custid) VALUES('A');
INSERT INTO dbo.Customers(custid) VALUES('B');

CREATE TABLE dbo.Orders
(
   orderid INT        NOT NULL PRIMARY KEY,
   custid  VARCHAR(5) NOT NULL REFERENCES dbo.Customers,
   cqty    INT        NOT NULL
);

INSERT INTO dbo.Orders(orderid, custid, qty) VALUES(1, 'A', 20);
INSERT INTO dbo.Orders(orderid, custid, qty) VALUES(2, 'A', 10);
INSERT INTO dbo.Orders(orderid, custid, qty) VALUES(3, 'A', 30);
INSERT INTO dbo.Orders(orderid, custid, qty) VALUES(4, 'B', 35);
INSERT INTO dbo.Orders(orderid, custid, qty) VALUES(5, 'B', 45);
INSERT INTO dbo.Orders(orderid, custid, qty) VALUES(6, 'B', 15);
```

There's a one-to-many relationship between Customers and Orders. Notice that each row in Customers currently has three related rows in Orders. Now, examine the following UPDATE and see if you can guess how Customers would look after the UPDATE:

```
UPDATE Customers
  SET qty = O.qty
FROM dbo.Customers AS C
  JOIN dbo.Orders AS O
   ON C.custid = O.custid;
```

The truth is that the UPDATE is nondeterministic. You can't guarantee which of the values from the related Orders rows will be used to update the *qty* value in Customers. Remember that you cannot assume or rely on any physical order of the data. For example, run the following query against Customers after running the preceding UPDATE:

```
SELECT custid, qty FROM dbo.Customers;
```

You might get the output shown in Table 8-2.

Table 8-2: Possible Contents of Customers After Nondeterministic UPDATE

custid	qty
A	20
B	35

Table 8-3: Another Possible Contents of cutomers After Nondeterministic UPDATE

custid	qty
A	10
B	15

Once you're done experimenting with nondeterministic UPDATEs, run the following code to drop Orders and Customers:

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

UPDATE with OUTPUT

As with INSERT and DELETE statements, UPDATE statements also support an OUTPUT clause, allowing you to return output when you update data. UPDATE is the only statement out of the three where there are both new and old versions of rows, so you can refer to both deleted and inserted. UPDATEs with the OUTPUT clause have many interesting applications. I will give an example of managing a message or event queue.

SQL Server 2005 introduces a whole new queuing infrastructure and a platform called Service Broker that is based on that infrastructure.

More Info For details about programming with Service Broker please refer to *Inside Microsoft SQL Server 2005: T-SQL Programming.*

You can use Service Broker to develop applications that manage queues in your database. However, when you need to manage queues on a much smaller scale, without delving into the new queuing infrastructure and platform, you can do so by using the new OUTPUT clause. To demonstrate managing a queue, run the following code, which creates the Messages table:

```
USE tempdb;
GO
IF OBJECT_ID('dbo.Messages') IS NOT NULL
  DROP TABLE dbo.Messages;
GO

CREATE TABLE dbo.Messages
(
  msgid   INT          NOT NULL IDENTITY ,
  msgdate DATETIME     NOT NULL DEFAULT(GETDATE()),
  msg     VARCHAR(MAX) NOT NULL,

  status  VARCHAR(20)  NOT NULL DEFAULT('new'),
  CONSTRAINT PK_Messages
    PRIMARY KEY NONCLUSTERED(msgid),
  CONSTRAINT UNQ_Messages_status_msgid
    UNIQUE CLUSTERED(status, msgid),
  CONSTRAINT CHK_Messages_status
    CHECK (status IN('new', 'open', 'done'))
);
```

For each message, you store a message ID, an entry date, message text, and a status indicating whether the message wasn't processed yet (*'new'*), is being processed (*'open'*), or has already been processed (*'done'*).

The following code simulates a session that generates messages by using a loop that inserts a message with random text every second. The status of newly inserted messages is *'new'* because the status column was assigned with the default value *'new'*. Run this code from multiple sessions at the same time:

```
SET NOCOUNT ON;
USE tempdb;
GO
DECLARE @msg AS VARCHAR(MAX);
WHILE 1=1
BEGIN
  SET @msg = 'msg' + RIGHT('000000000'
    + CAST(CAST(RAND()*2000000000 AS INT)+1 AS VARCHAR(10)), 10);
  INSERT INTO dbo.Messages(msg) VALUES(@msg);
  WAITFOR DELAY '00:00:01';
END
```

Of course, you can play with the delay period as you like.

The following code simulates a session that processes messages using the following steps:

1. Form an endless loop that constantly processes messages.
2. Lock one available new message using an UPDATE TOP(@n) statement with the READPAST hint to skip locked rows, and change its status to *'open'*. *@n* represents a configurable input that determines the maximum number of messages to process in each iteration.
3. Store the attributes of the messages in the *@Msgs* table variable using the OUTPUT clause.
4. Process the messages.
5. Set the status of the messages to *'done'* by joining the Messages table and the *@Msgs* table variable.
6. If no new message was found in the Messages table, wait for one second.

```
SET NOCOUNT ON;
USE tempdb;
GO

DECLARE @Msgs TABLE(msgid INT, msgdate DATETIME, msg VARCHAR(MAX));
DECLARE @n AS INT;
SET @n = 3;

WHILE 1 = 1
BEGIN
  UPDATE TOP(@n) dbo.Messages WITH(READPAST) SET status = 'open'
    OUTPUT inserted.msgid, inserted.msgdate, inserted.msg INTO @Msgs
    OUTPUT inserted.msgid, inserted.msgdate, inserted.msg
  WHERE status = 'new';

  IF @@rowcount > 0
  BEGIN
    PRINT 'Processing messages...';
    /* ...process messages here... */

    WITH UPD_CTE AS
    (
      SELECT M.status
      FROM dbo.Messages AS M
        JOIN @Msgs AS N
          ON M.msgid = N.msgid
    )
    UPDATE UPD_CTE
      SET status = 'done';

    DELETE FROM @Msgs;
  END
  ELSE
  BEGIN
```

```
    PRINT 'No messages to process.';
    WAITFOR DELAY '00:00:01';
  END
END
```

You can run this code from multiple sessions at the same time. You can increase the number of sessions that would run this code based on the processing throughput that you need to accommodate.

Note that just for demonstration purposes, I included in the first UPDATE statement a second OUTPUT clause, which returns the messages back to the caller. I find this UPDATE statement particularly beautiful because it encompasses four different T-SQL enhancements in SQL Server 2005: UPDATE TOP, TOP with an input expression, the OUTPUT clause, and the READPAST hint in data modification statements. The READPAST hint was available in SQL Server 2000, but only to SELECT queries.

SELECT and UPDATE Statement Assignments

This section covers statements that assign values to variables and, in the case of UPDATE, can modify data at the same time. There are some tricky issues with such assignments that you might want to be aware of. Being familiar with the way assignments work in T-SQL is important to program correctly—that is, to program what you intended to.

Assignment SELECT

I'll start with assignment SELECT statements. T-SQL supports assigning values to variables using a SELECT statement, but the ANSI form of assignment, which is also supported by T-SQL, is to use a SET statement. So, as a rule, unless there's a compelling reason to do otherwise, it's a good practice to stick to using SET. I'll describe cases where you might want to use SELECT because it has advantages over SET in those cases. However, as I will demonstrate shortly, you should be aware that when using SELECT, your code is more prone to errors.

As an example of the way an assignment SELECT works, suppose you need to assign the employee ID whose last name matches a given pattern (*@pattern*) to the *@EmpID* variable. You assume that only one employee will match the pattern. The following code, which uses an assignment SELECT, doesn't accomplish the requirement:

```
USE Northwind;

DECLARE @EmpID AS INT, @Pattern AS NVARCHAR(100);

SET @Pattern = N'Davolio'; -- Try also N'Ben-Gan', N'D%';
SET @EmpID = 999;

SELECT @EmpID = EmployeeID
FROM dbo.Employees
WHERE LastName LIKE @Pattern;

SELECT @EmpID;
```

Given *N'Davolio'* as the input pattern, you get the employee ID 1 in the *@EmpID* variable. In this case, only one employee matched the filter. However, if you're given a pattern that does not apply to any existing last name in the Employees table (for example, *N'Ben-Gan'*), the assignment doesn't take place even once. The content of the *@EmpID* variable remains as it was before the assignment—999. (This value is used for demonstration purposes.) If you're given a pattern that matches more than one last name (for example, *N'D%'*), this code will issue multiple assignments, overwriting the previous value in *@EmpID* with each assignment. The final value of *@EmpID* will be the employee ID from the qualifying row that SQL Server happened to access last.

A much safer way to assign the qualifying employee ID to the *@EmpID* variable is to use a SET statement as follows:

```
DECLARE @EmpID AS INT, @Pattern AS NVARCHAR(100);

SET @Pattern = N'Davolio'; -- Try also N'Ben-Gan', N'D%';
SET @EmpID = 999;

SET @EmpID = (SELECT EmployeeID
              FROM dbo.Employees
              WHERE LastName LIKE @Pattern);

SELECT @EmpID;
```

If only one employee qualifies, you will get the employee ID in the *@EmpID* variable. If no employee qualifies, the subquery will set *@EmpID* to NULL. When you get a NULL, you know that you had no matches. If multiple employees qualify, you will get an error saying that the subquery returned more than one value. In such a case, you will realize that there's something wrong with your assumptions or with the design of your code. But the problem will surface as opposed to eluding you.

When you understand how an assignment SELECT works, you can use it to your advantage. For example, a SET statement can assign only one variable at a time. An assignment SELECT can assign values to multiple variables within the same statement. With well-designed code, this capability can give you performance benefits. For example, the following code assigns the first name and last name of a given employee to variables:

```
DECLARE @FirstName AS NVARCHAR(10), @LastName AS NVARCHAR(20);

SELECT @FirstName = NULL, @LastName = NULL;

SELECT @FirstName = FirstName, @LastName = LastName
FROM dbo.Employees
WHERE EmployeeID = 3;

SELECT @FirstName, @LastName;
```

Notice that this code uses the primary key to filter an employee, meaning that you cannot get more than one row back. The code also initializes the *@FirstName* and *@LastName* variables with NULLs. If no employee qualifies, the variables will simply retain the NULLs. This type of assignment is especially useful in triggers when you want to read attributes from the special tables inserted and deleted into your own variables, after you verify that only one row was affected.

Technically, you could rely on the fact that an assignment SELECT performs multiple assignments when multiple rows qualify. For example, you could do aggregate calculations, such as concatenating all order IDs for a given customer:

```
DECLARE @Orders AS VARCHAR(8000), @CustomerID AS NCHAR(5);
SET @CustomerID = N'ALFKI';
SET @Orders = '';

SELECT @Orders = @Orders + CAST(OrderID AS VARCHAR(10)) + ';'
FROM dbo.Orders
WHERE CustomerID = @CustomerID;

SELECT @Orders;
```

However, this code is far from being standard, and you can't guarantee the order of assignment. I've seen attempts programmers made to control the order of assignment by introducing an ORDER BY clause like so:

```
DECLARE @Orders AS VARCHAR(8000), @CustomerID AS NCHAR(5);
SET @CustomerID = N'ALFKI';
SET @Orders = '';

SELECT @Orders = @Orders + CAST(OrderID AS VARCHAR(10)) + ';'
FROM dbo.Orders
WHERE CustomerID = @CustomerID
ORDER BY OrderDate, OrderID;

SELECT @Orders;
```

But you cannot force the optimizer to sort before applying the assignments. If the optimizer chooses to sort after the assignments, the ORDER BY clause here fails to have the desired effect. You realize that when you specify an ORDER BY clause you might get an intermediate state of the variable. In short, it's better to not rely on such techniques. There are enough supported and guaranteed techniques that you can choose from for such calculations, many of which I covered in Chapter 6.

Assignment UPDATE

T-SQL also supports a nonstandard UPDATE syntax that can assign values to variables in addition to modifying data. To demonstrate the technique, first run the following code, which creates the table T1 and populates it with sample data:

```
USE tempdb;
GO
IF OBJECT_ID('dbo.T1') IS NOT NULL
  DROP TABLE dbo.T1;
```

```
GO

CREATE TABLE dbo.T1
(
  col1 INT         NOT NULL,
  col2 VARCHAR(5) NOT NULL
);

INSERT INTO dbo.T1(col1, col2) VALUES(0, 'A');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'B');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'C');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'C');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'C');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'B');

INSERT INTO dbo.T1(col1, col2) VALUES(0, 'A');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'A');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'C');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'C');
```

Currently, the T1 table has no primary key and there's no way to uniquely identify the rows. Suppose that you wanted to assign unique integers to *col1* and then make it the primary key. You can use the following assignment UPDATE to achieve this task:

```
DECLARE @i AS INT;
SET @i = 0;
UPDATE dbo.T1 SET @i = col1 = @i + 1;
```

This code declares the variable *@i* and initializes it with 0. The UPDATE statement then scans the data and, for each row, sets the current *col1* value to *@i + 1*, and then sets *@i*'s value to *col1*'s new value. Logically, the SET clause is equivalent to *SET col1 = @i + 1, @i = @i + 1*. However, in such an UPDATE statement, there's no way for you to control the order in which the rows in T1 will be scanned and modified. For example, Table 8-4 shows how the contents of T1 might look after the assignment.

Table 8-4: Contents of T1 After Applying Assignment UPDATE

col1	col2
1	A
2	B
3	C
4	C
5	C
6	B
7	A
8	A
9	C
10	C

But keep in mind that it might be different. As long as you don't care about the order in which the data is scanned and modified, you might be happy with this technique. It is very fast, as it scans the data only once.

SQL Server 2005 allows you to achieve the task in an elegant manner, where you can guarantee the result row numbers will be based on a requested ordering. To do so, issue an UPDATE against a CTE that calculates row numbers based on any desired order:

```
WITH T1 RN AS
(
```

```
  SELECT col1, ROW_NUMBER() OVER(ORDER BY col2) AS RowNum
  FROM dbo.T1
)
UPDATE T1 RN SET col1 = RowNum;
```

Table 8-5 shows the contents of T1 after the UPDATE.

Table 8-5: Contents of T1 After Applying UPDATE Against CTE

col1	col2
1	A
4	B
6	C
7	C
8	C
5	B
2	A
3	A
9	C
10	C

By now, you have probably figured out why my favorite features in SQL Server 2005 are the ROW_NUMBER function and CTEs.

Other Performance Considerations

In this section, I'll provide a brief background about designing and architecting write-intensive systems. I thought that such background might be interesting for programmers, particularly in terms of realizing that there's a lot involved in designing systems when aiming for good modification performance. If you're not familiar with the terms described in this section, feel free to skip it.

More Info You can find more thorough coverage of the subject and more detailed explanations about the terms discussed here in *Inside Microsoft SQL Server 2005: The Storage Engine* (Microsoft Press, 2006) by Kalen Delaney.

You need to take several important things into consideration when designing write-intensive systems. Unless I explicitly mention otherwise, the discussion here applies to any type of data modification (inserting, deleting, and updating data).

In terms of physical design of your database system, remember that modifications affect two sections of the database: the data and the transaction log. When a modification takes place, SQL Server first looks for the pages that need to be modified in cache. If the target pages are already in cache, SQL Server modifies them there. If not, SQL Server will first load the target pages from the data portion of the database into cache, and then modify them there. SQL Server writes the change to the transaction log. (I'll discuss aspects of logging shortly.) SQL Server periodically runs a checkpoint process, which flushes dirty (changed) pages from cache to the data portion of the database on disk. However, SQL Server will flush only dirty pages for which the change was already written to the transaction log. The transaction log essentially provides the durability facet of transactions (the D part of the ACID facets of a transaction), allowing rollback and roll-forward capabilities.

Multiple write activities to the data portion of the database can be done simultaneously by using multiple disk drives. Therefore, striping the data over multiple disk drives is a key performance factor. The more spindles, the better. RAID 10 is typically the optimal type of RAID system for write-intensive environments. RAID 10 both stripes the data and mirrors every drive. It doesn't involve any parity calculations. Its main downside is that it's expensive because half the drives in the array are used to mirror the other half. RAID 5 is usually a poor choice for write-intensive systems because it involves parity

calculations with every write. Its main advantage is its low cost, because for an array with *n* drives, only 1/*n* is used for parity. However, in many cases RAID 10 can yield more than a 50 percent write performance improvement over RAID 5, so its higher cost is usually worthwhile. For systems that mainly read data (for example, data warehouses), RAID 5 is fine for the data portion as long as the extract, transform, and load (ETL) process has a sufficient window of time (typically during the night). While during the day, applications only read data from the data warehouse.

As for the transaction log, its architecture is such that it can be written only synchronously— that is, only in a sequential manner. Therefore, striping the log over multiple disk drives provides no benefit unless you also have processes that read from the transaction log. One example of a process that reads from the transaction log is transaction log replication. Another example is accessing the deleted and inserted views in triggers in SQL Server 2000.

These views in SQL Server 2000 reflect the portion of the log containing the change that fired the trigger. In SQL Server 2005, deleted and inserted rows are based on the new rowversioning technology, which maintains row versions in the tempdb database. This means that when you access the inserted and deleted views, SQL Server 2005 will scan data in tempdb and not the transaction log. At any rate, it's a good practice to separate the transaction log into its own drive to avoid interfering with its activity. Any interference with the transaction log activity ultimately postpones writes to the data portion of the database. Unless you have processes reading from the transaction log, RAID 1 is sufficient. RAID 1 just mirrors a drive and doesn't stripe data. But if you also have intensive processes reading from the log, RAID 10 would be a better choice.

As for tempdb, it's always a good idea to stripe it using RAID 10. That's true for both online transaction processing (OLTP) and data warehouse systems, because SQL Server spools data in tempdb for many background activities related to both reads and writes. Tuning tempdb becomes even more important in SQL Server 2005 because the new row-versioning technology that is used by several activities stores and reads row versions from tempdb. Activities that use row-versioning include online index operations, constructing the deleted and inserted tables in triggers, new snapshot isolations, and multiple active result sets (MARS).

The synchronous manner in which SQL Server writes to the transaction log has a significant performance impact on data modification. The performance of fully logged modifications is often bound by the time it takes to write to the log. Furthermore, once the capacity of writes reaches the throughput of the disk drive on which the log resides, modification will start waiting for log writes. When the transaction log becomes the bottleneck, you should consider splitting the database into multiple ones, each with its own transaction log residing on separate disk drives.

With this background, you can see that in many cases the performance of data modifications in general—and data insertions in particular—will be strongly related to the amount of logging. When designing insert processes, one of your main considerations should be the amount of logging.

SQL Server will always fully log a modification unless two conditions are met: the database recovery model is not FULL, and the operation is considered a BULK operation. A BULK operation issued against a database with a non-FULL recovery model is minimally logged. Minimal logging means recording only the addresses of the extents that were allocated during the operation to support rollback capabilities, as opposed to logging the whole change. BULK operations include creation or rebuilding of an index, inserts that use the BULK engine (for example, BULK INSERT), SELECT INTO, Large Object (LOB) manipulation. Examples for LOB manipulation in SQL Server 2000 include WRITETEXT and UPDATETEXT. SQL Server 2005 also supports new LOB manipulation techniques, including using the BULK rowset provider and the WRITE method. A minimally logged modification is typically dramatically faster than a fully logged modification.

So, when designing insert processes, your first choice should be to use a BULK operation. The second choice should be to use a set-based multirow INSERT (INSERT SELECT), and the last choice should be individual INSERTs. The basic rule is the amount of logging. Individual INSERTs incur substantially more logging than multirow INSERTs, which in turn incur more logging than BULK operations that are run in a database with a non-FULL recovery model.

Another factor that will affect the performance of your inserts is the transaction size. This is a tricky issue.

The worst-case scenario is inserting individual rows, each in its own transaction. In such a case, not only is the INSERT fully logged, each INSERT causes the writing of three records to the transaction log: BEGIN TRAN, the INSERT itself, and a COMMIT TRAN. Besides the excessive writes to the log, you need to consider the overhead involved with maintaining each transaction (for example, acquiring locks, releasing locks, and so on). To improve insert performance, you want to encapsulate each batch of multiple rows into a single transaction.

The question is, what's the optimal transaction size? That's the tricky part. Things are not black and white in the sense that single-row transactions would be the worst-case scenario and one big transaction would be the best-case scenario. In

practice, as the transaction grows larger, the insert performance improves up to a point. The point at which insert performance starts degrading is when maintaining the huge transaction involves too much overhead for SQL Server. Many factors affect the transaction size that would give you the optimal insert performance. These include the hardware, layout of your database (data, log), index design, and so on. So, in practical terms, the best way to figure out the optimal transaction size for a given table is to use a benchmark. Just test your insert process with different transaction sizes, increasing or decreasing the number of rows you encapsulate in a single transaction based on the performance that you get. After tweaking the transaction size several times like this, you will find the optimal one.

When tuning single row inserts, in your test environment you can form a loop that inserts a row in each iteration, and maintain a counter in your INSERT loop. Open a new transaction before every *n* iterations, where *n* is the number of rows you want to test inserting in a single transaction, and commit the transaction after every *n* iterations. I've seen production environments that collect the data for insertion from the source environments, and form such a loop to insert data to the target tables. Of course, not every system is designed to accommodate such a looping logic.

This process (controlling the insert transaction size) is relevant to any type of insert—that is, single-row INSERTs, multiple row set–based INSERTs (INSERT SELECT), and bulk inserts. With bulk inserts (for example, using BULK INSERT), by default, the whole insert is considered a single transaction. But you can control the number of rows per transaction and the way the insert will be optimized by setting the BATCHSIZE and ROWS_PER_BATCH options.

Conclusion

Data modifications involve many challenges. You need to be familiar with SQL Server's architecture and internals to design systems that can cope with large volumes of data, and largescale modifications. There are also many challenging logical problems related to data modifications, such as maintaining your own custom sequence, deleting rows with duplicate data, and assigning unique values to existing rows. In this chapter, I covered performance aspects of data modifications as well as logical ones. I introduced quite a few key techniques that you will likely find useful.