Apress™
Books for Professionals by Professionals™

Sample Chapter: "Tips and Tricks"
(pre-production "galley" stage)

Advanced Transact-SQL for SQL Server 2000

Practical T-SQL solutions (with code) to common problems

by Itzik Ben-Gan, MVP and Tom Moreau, Ph.D.
ISBN # 1-893115-82-8

Copyright ©2000 Apress, L.P., 901 Grayson St., Suite 204, Berkeley, CA 94710. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

CHAPTER 17

Tips and Tricks

THIS CHAPTER IS DEVOTED TO tips and tricks that solve common needs of T-SQL users and programmers. The solutions presented here were collected from our experience and also from the experience of our colleagues, who were kind enough to contribute their unique solutions to problems they encountered.

We would like to express our thanks to all those who contributed their tips.

Unlearning an Old Trick

Before you begin learning new tricks, there is one you should unlearn. Consider the query in Listing 17-1, which determines the first and last order dates in the Northwind database.

Listing 17-1: Query Using MIN() and MAX() on Same Column

```
SELECT
  MIN (OrderDate),
  MAX (OrderDate)
FROM
  Orders
```

Prior to version 7.0, this query would have produced a table scan instead of using the index on OrderDate. The optimizer wasn't bright enough to figure out that it could tap the index twice and come up with the desired information.

The workaround for this problem involved a nested subquery, as shown in Listing 17-2.

Listing 17-2: Using a Nested Subquery to Use an Index

```
SELECT
  MIN (OrderDate),
  (SELECT MAX (OrderDate) FROM Orders)
FROM
  Orders
```

For this query, the optimizer first performed the nested subquery, which was a quick tap on the index. It then did the outer query, which again made a quick tap on the index. If you run these queries in SQL Server 7.0 or 2000, you get identical performance in I/O and speed, as well as identical query plans. What is even more stunning is the fact that the Query Analyzer uses a `JOIN`. This is because it goes to the index twice and then needs to meld the two pieces of information—the `MIN()` and the `MAX()`—to return the results.

At this point, you don't really need to change any existing code because the results and performance are the same. However, if there is no index you can use, and you use the nested subquery trick, you will get two table scans or two clustered index scans. If you eliminate the subquery, you get only one table scan or clustered index scan. Therefore, in versions of SQL Server before version 7.0, the subquery trick would do no harm and would have the same performance as the non-subquery version if there is no index it could use. However, using this trick in SQL Server 7.0 and 2000 gives worse performance.

Keep in mind that indexes can sometimes get dropped, and if this code is not converted, you could be in for a surprise.

Getting NULLs to Sort Last Instead of First

`NULL`s are special. They are like spoiled children that always need special attention. When you compare them with other values, the result is `UNKNOWN` even when you compare them with other `NULL`s. You always need to take special steps, like using the `IS NULL` operator instead of an equality operator when you look for `NULL`s. However, in some situations, `NULL`s are considered to be equal. Those situations include the `UNIQUE` constraint, `GROUP BY`, and `ORDER BY`.

`ORDER BY` considers `NULL`s to be equal to each other, but the ANSI committee does not define whether they should have a lower or higher sort value than all other known values, so you might find different implementations in different systems. SQL Server sorts `NULL`s before all other values.

What do you do if you want to sort them after all other values? For example, suppose you want to return all customers from the Northwind sample database, ordered by Region. If you issue the query shown in Listing 17-3, you will get `NULL`s first.

Listing 17-3: NULLs Sort First

```
SELECT
  *
FROM
  Customers
ORDER BY
  Region
```

You can use the CASE expression in the ORDER BY clause to return 1 when the region is NULL and 0 when it is not NULL (for similar uses of the CASE expression, please refer to Chapter 4). You can use this result as the first sort value, and the Region as the second. This way, 0s representing known values will sort first, and 1s representing NULLs will sort last. Listing 17-4 shows the query.

Listing 17-4: NULLs Sort Last

```
SELECT
  *
FROM
  Customers
ORDER BY
  CASE
    WHEN Region IS NULL THEN 1
    ELSE 0
  END,
  Region
```

This is all nice and well, but now that the ORDER BY clause uses an expression and not an explicit column, the optimizer will not consider using the index on the region column (which might improve the query performance by not performing an explicit sort operation). Prior to SQL Server 2000, there was not much you could do about it, but with SQL Server 2000, you can create an indexed view with the CASE expression as one of its columns. You can also have the CASE expression as a computed column in the Customers table and create a composite index on the computed column and the original Region column, as Listing 17-5 shows.

Listing 17-5: Adding a Computed Column and an Index on It to the Customers Table

```
ALTER TABLE Customers
  ADD RegionNullOrder AS
    CASE
      WHEN region IS NULL THEN 1
      ELSE 0
    END
GO

CREATE INDEX idx_nci_RegionNullOrder_Region ON
  Customers (RegionNullOrder, Region)
GO
```

Now you can rewrite your query as Listing 17-6 shows, and if you turn SHOWPLAN on to display the execution plan, you will see that it makes use of the new index, as shown in Listing 17-7. The SHOWPLAN option is covered in Appendix C.

Listing 17-6: NULLs Sort Last and an Index Is Used

```
SET SHOWPLAN_TEXT ON
GO

SELECT
  *
FROM
  Customers
ORDER BY
  RegionNullOrder,
  Region
GO
```

Listing 17-7: SHOWPLAN's Output, the Index on the Computed Column Is Used

```
|--Bookmark Lookup(BOOKMARK:([Bmk1000]),
     OBJECT:([Northwind].[dbo].[Customers]))
   |--Index Scan(OBJECT:(
     [Northwind].[dbo].[Customers].[idx_nci_RegionNullOrder_Region]),
       ORDERED FORWARD)
```

Using a Parameter for the Column in the ORDER BY Clause (by Bruce P. Margolin)

The ORDER BY clause accepts only explicit column names or expressions; it won't accept a column name stored in a variable. Suppose you want to write a stored procedure that returns an ordered output of the authors in the Authors table in the pubs sample database, but you want to pass it a parameter that tells it which column to ORDER BY. There are a few ways to approach this problem. You can use either the column number or name as a parameter and use a CASE expression to determine the column, or you can use the column number or name with dynamic execution.

Using the Column Number as Parameter and CASE to Determine the Column

You can pass the column number as a parameter to a stored procedure and use a CASE expression in the ORDER BY clause to pick the relevant column, as Listing 17-8 shows.

Listing 17-8: Passing the ORDER BY Column as a Parameter, Using a Column Number, First Try

```
CREATE PROC GetAuthors1
  @colnum AS int
AS

SELECT
  *
FROM
  authors
ORDER BY
  CASE @colnum
    WHEN 1 THEN au_id
    WHEN 2 THEN au_lname
    WHEN 3 THEN au_fname
    WHEN 4 THEN phone
    WHEN 5 THEN address
    WHEN 6 THEN city
    WHEN 7 THEN state
    WHEN 8 THEN zip
    WHEN 9 THEN contract
    ELSE NULL
  END
GO
```

Notice, however, what happens when you try to execute the `GetAuthors1` stored procedure, providing 1 as an argument to indicate that you want the output to be sorted by au_id, as shown in Listing 17-9.

Listing 17-9: Error When Trying to Invoke the GetAuthors1 Stored Procedure

```
EXEC GetAuthors1
  @colnum = 1

Server: Msg 245, Level 16, State 1, Procedure GetAuthors1, Line 5
Syntax error converting the varchar value '172-32-1176'
to a column of data type bit.
```

The reason for this error is that the `CASE` expression's return value's datatype is determined by the highest datatype, according to the datatypes precedence rules (see the Books Online for details). In this case, the `bit` datatype of the contract column has the highest precedence, so it determines the datatype of the result of the `CASE`

expression. Error 245 indicates that the author ID `'172-32-1176'`, which is of the datatype `varchar`, cannot be converted to `bit`, of course.

You can get around this problem by casting the problematic contract column to a `char` datatype, as Listing 17-10 shows.

Listing 17-10: Passing the ORDER BY Column as a Parameter, Using a Column Number, Second Try

```
ALTER PROC GetAuthors1
  @colnum AS int
AS

SELECT
  *
FROM
  authors
ORDER BY
  CASE @colnum
    WHEN 1 THEN au_id
    WHEN 2 THEN au_lname
    WHEN 3 THEN au_fname
    WHEN 4 THEN phone
    WHEN 5 THEN address
    WHEN 6 THEN city
    WHEN 7 THEN state
    WHEN 8 THEN zip
    WHEN 9 THEN CAST(contract AS CHAR(1))
    ELSE NULL
  END
GO
```

Note that you will have the same problem with numeric columns, but simply casting a numeric column to a character datatype won't be sufficient. You will also need to prefix the cast values with the proper number of zeros so that they will sort properly. For example, suppose you have a qty column holding the value `10` in one row and `2` in another row. Simply casting those values to the character strings `'10'` and `'2'`, respectively, will result in `2` being sorted after `'10'`, because a character sort will be performed here instead of a numeric sort.

To avoid this problem, you can prefix the qty column with the proper number of zeros, causing all of the cast values to have the same length as the maximum possible length of the qty column, say ten digits in our example. Listing 17-11 shows how this can be achieved.

Listing 17-11: Prefixing a Numeric Value with Zeros

```
RIGHT (REPLICATE ('0', 10) + CAST (qty AS varchar (10)), 10)
```

Using Dynamic Execution

Another option using the column number is to construct the `SELECT` statement in a variable, and execute it dynamically with the `EXEC` command, as Listing 17-12 shows.

Listing 17-12: Passing the ORDER BY Column as a Parameter, Using Dynamic Execution

```
CREATE PROC GetAuthors2
  @colnum AS int
AS

DECLARE
  @cmd AS varchar (8000)

SET @cmd =
  'SELECT *'     + CHAR (13) + CHAR(10) +
  'FROM authors' + CHAR (13) + CHAR(10) +
  'ORDER BY '    + CAST (@colnum AS varchar (4))

EXEC(@cmd)
GO
```

Using the Column Name as Parameter and CASE to Determine the Column

Finally, you can pass the column name as a `sysname` (`nvarchar(128)`), which is used for identifiers, and check it using a `CASE` expression similar to the first example, but now it checks for column names rather than column numbers, as Listing 17-13 shows.

Listing 17-13: Passing the ORDER BY Column as a Parameter, Using a Column Name

```
CREATE PROC GetAuthors3
  @colname AS sysname
AS
```

```
SELECT
  *
FROM
  authors
ORDER BY
  CASE @colname
    WHEN 'au_id' THEN au_id
    WHEN 'au_lname' THEN au_lname
    WHEN ...
    WHEN 'contract' THEN CAST (contract AS CHAR (1))
    ELSE NULL
  END
GO
```

Formatting Output that May Be Null (by Robert Skoglund)

Sometimes you need a SQL trick because of the requirements of the front end. For example, suppose your client-side GUI uses a multiline edit box for displaying the customer's address. Some of your customers have region information, such as state or province, and for others this is NULL. Concatenating NULL with anything will give you a NULL. What is needed is a way to provide the formatted address information while handling NULL information.

Using the Northwind database, Robert creates a view of the form as shown in Listing 17-14.

Listing 17-14: A View to Build a String and Handle NULLs

```
CREATE VIEW MailingList
AS
SELECT
  CustomerID,
  CompanyName + CHAR (13) + CHAR (10) +
  Address     + CHAR (13) + CHAR (10) +
  City        + CHAR (13) + CHAR (10) +
  CASE WHEN Region IS NOT NULL THEN Region + CHAR (13) + CHAR (10)
       ELSE ''
  END +  Country AS ContactAddress
FROM
  Customers
```

The carriage returns and line feeds—CHAR(13) + CHAR(10)—are provided to format the text for the multiline edit box. When Region is not NULL, then Region plus the carriage return and line feed are added to the address. Otherwise, an empty

string is added. Robert's original design involved the use of the SUBSTRING() function, which has been replaced here with a CASE construct.

Embedding Stored Procedure Rowsets in SELECT and SELECT INTO Statements

Chapter 11 covers user-defined functions and shows you how to build table-valued functions that return a rowset. You can embed a call to such a function in a SELECT query that performs a join, for example, or even in a SELECT INTO statement to create a target table and populate it with the result of the function. User-defined functions (UDFs) are not available in SQL Server 7.0. You can create a stored procedure that accepts parameters and returns a rowset as a result, but you can't embed it naturally in SELECT or SELECT INTO statements. However, you insist! Of course, you refuse to be left empty handed.

Suppose you wanted to embed the result of the stored procedure shown in Listing 17-15, which returns authors from the pubs sample database for a given state, into your SELECT or SELECT INTO statements.

Listing 17-15: Creation Script for Stored Procedure AuthorsInState

```
USE pubs

GO

CREATE PROC AuthorsInState
  @state char(2)
AS

SELECT
  *
FROM
  authors
WHERE
  state = @state
GO
```

You have three options: using the INSERT EXEC statement, the OPENROWSET() function, or the OPENQUERY() function. These options are presented in the next three sections. Later, in the section "Using OPENROWSET() in a View," you will build on your skill with OPENROWSET. Finally, in the section "Choosing between SQL and OPENQUERY()," you will see how to make the decision to use OPENQUERY().

Using the INSERT EXEC Statement

You can create a temporary table with the same structure as the rowset returned from the stored procedure, as Listing 17-16 shows.

Listing 17-16: Schema Creation Script for the #TmpAuthors Table

```
CREATE TABLE #TmpAuthors
(
  au_id    varchar(11) NOT NULL,
  au_lname varchar(40) NOT NULL,
  au_fname varchar(20) NOT NULL,
  phone    char(12)    NOT NULL,
  address  varchar(40) NULL,
  city     varchar(20) NULL,
  state    char(2)     NULL,
  zip      char(5)     NULL,
  contract bit         NOT NULL
)
```

You can then use the `INSERT EXEC` statement to populate it, as Listing 17-17 shows.

Listing 17-17: Using the INSERT EXEC Statement

```
INSERT INTO #TmpAuthors
  EXEC AuthorsInState 'CA'
```

Now you can use the temporary table in `SELECT` or `SELECT INTO` statements just like any other table, as Listings 17-18 and 17-19 show.

Listing 17-18: Embedding the Result of the INSERT EXEC Statement in a SELECT Statement

```
SELECT
  *
FROM
  #TmpAuthors
GO
```

Listing 17-19: Embedding the Result of the INSERT EXEC Statement in a SELECT INTO Statement

```
SELECT
  *
INTO
  #TmpAuthors2
FROM
  #TmpAuthors
GO
```

Using the OPENROWSET() Function

The `OPENROWSET()` function is used to issue ad hoc queries against heterogeneous data sources. You can also use it to invoke remote stored procedures. As a result, you can use it to invoke the `AuthorsInState` stored procedure as if the local server were a remote one. Listing 17-20 shows how to embed the `OPENROWSET()` function in a `SELECT` statement.

Listing 17-20: Embedding the Result of the OPENROWSET Function in a SELECT Statement

```
SELECT
  T1.*
FROM
  OPENROWSET('SQLOLEDB','<server>';'<user>';'<pass>',
           'EXEC pubs..AuthorsInState ''CA''') AS T1
```

Listing 17-21 shows you how to embed the `OPENROWSET()` function in a `SELECT INTO` statement.

Listing 17-21: Embedding the Result of the OPENROWSET Function in a SELECT INTO Statement

```
SELECT
  *
INTO
  #TmpAuthors
FROM
  OPENROWSET('SQLOLEDB', <server>';'<user>';'<pass>,
           'EXEC pubs..AuthorsInState ''CA''') AS T1
```

Using the OPENQUERY() Function

The OPENQUERY() function is used to issue pass-through queries against a linked server. You can also use it to invoke a remote stored procedure from the linked server. You can refer to your own local server as a linked server by turning on the 'Data Access' server option as Listing 17-22 shows.

Listing 17-22: Turning On the 'Data Access' Server Option

```
EXEC sp_serveroption '<server>', 'Data Access', 'true'
```

Now you can embed the OPENQUERY() function in a SELECT statement as Listing 17-23 shows.

Listing 17-23: Embedding the Result of the OPENQUERY Function in a SELECT Statement

```
SELECT
  T1.*
FROM
  OPENQUERY([<server>],
            'EXEC pubs..AuthorsInState ''CA''') AS T1
```

You can also embed the OPENQUERY() function in a SELECT INTO statement, as Listing 17-24 shows.

Listing 17-24: Embedding the Result of the OPENQUERY Function in a SELECT INTO Statement

```
SELECT
  *
INTO
  #TmpAuthors
FROM OPENQUERY([<server>],
               'EXEC pubs..AuthorsInState ''CA''')
```

Using OPENROWSET() in a View

From time to time, you have to use vendor products that do not support stored procedures. Rather, they can do SELECTs on tables and views. This is no problem; with the OPENROWSET() function, you can create a view that acts as a wrapper for the stored procedure call. Check out Listing 17-25.

Listing 17-25: Creating a View on a Stored Procedure Call

```
CREATE VIEW StoredProcWrapper
AS
SELECT
  *
FROM
    OPENROWSET
    (
      'SQLOLEDB',
      'SERVER=.;Trusted_Connection=yes',
      'SET FMTONLY OFF EXEC sp_who2'
    )
```

Here, the `SERVER=.` piece refers to the default instance of SQL Server on a machine on which SQL Server 2000 is running. You will have to specify the correct instance if you are not using the default, e.g., `SERVER=BMCI03\BMCI03_02`.

Choosing between SQL and OPENQUERY()

The server closest to the data is the one that should work with that data. Consider the following scenario, using the Northwind database. You have a Customers table on your local server, but you want to know the quantity of product ID 17 purchased by your local Canadian customers on the remote server, and you want it broken down by customer ID. The code in Listing 17-26 shows you how to do this with an SQL statement that uses the four-part naming convention to reference the tables.

Listing 17-26: Using Remote Tables with the Four-Part Naming Convention

```
SELECT
  C.CustomerID,
  SUM (OD.Quantity) AS Quantity
FROM
    Customers                          C
  JOIN
    Remote.Northwind.dbo.Orders        O  ON O.CustomerID = C.CustomerID
  JOIN
    Remote.Northwind.dbo.[Order Details] OD ON OD.OrderID   = O.OrderID
WHERE
    C.Country    = 'Canada'
  AND
    OD.ProductID = 17
GROUP BY
  C.CustomerID
```

This query did all of the work on the local server, even though the remote server had most of the data. Now compare this to using the `OPENQUERY()` function, as shown in Listing 17-27.

Listing 17-27: Using OPENQUERY() to Do the Majority of the Work on the Remote Server

```
SELECT
  C.CustomerID,
  O.Quantity
FROM
    Customers                   C
  JOIN
    OPENQUERY
    (
      Remote,
      'SELECT
         O.CustomerID,
         SUM (OD.Quantity) AS Quantity
       FROM
           Northwind..Orders          O
         JOIN
           Northwind..[Order Details] OD ON OD.OrderID = O.OrderID
       WHERE
           OD.ProductID = 17
       GROUP BY
         O.CustomerID'
    )                           O ON O.CustomerID = C.CustomerID
WHERE
    C.Country = 'Canada'
```

Here, the heavy work was done on the remote server where you had most of the data. Even the aggregation could be done there. Also, consider where you want the query to execute. In this case, the local server had the least amount of the data but it was the one managing the query.

You can also execute the query on the “remote” server and reference the Customers table from the “local” server, as shown in Listing 17-28.

Listing 17-28: Running the Query on the Remote Server

```
SELECT
  C.CustomerID,
  SUM (OD.Quantity) AS Quantity
FROM
    Local.Northwind.dbo.Customers C
  JOIN
    Orders                  O  ON O.CustomerID = C.CustomerID
  JOIN
    [Order Details]         OD ON OD.OrderID   = O.OrderID
WHERE
    C.Country    = 'Canada'
  AND
    OD.ProductID = 17
GROUP BY
  C.CustomerID
```

The trick here is to experiment with where you run the query and whether you directly access the remote table (via four-part naming) or use the `OPENQUERY()` function.

Using CASE in a JOIN (by Robert Vieira)

Suppose that you have a "Provider" institution that may have multiple children that are all "Company" institutions. Companies have Regions and Sites. Also, a Provider has both restricted and unrestricted users. The problem is to figure out the algorithm.

If a Provider user is unrestricted, then they will only have the Provider institution in their institution list. If they are a restricted user, then they will have the Provider plus all the companies to which they have rights. Therefore, the query needed is a bit different depending on whether they have just the provider or a list of companies.

The code shown in Listing 17-29 gets the institution list that the user can see (all institutions under the provider if it only finds the Provider row; just the list of company rows if it finds more than the provider). It works quite well. It does the query without a stored procedure, and all in a single round trip to the server.

Listing 17-29: Using CASE in a JOIN

```
USE RiskNet

-- GET INSTITUTION CHILDREN
SELECT
  v.InstitutionID,
  v.Name,
  v.HierarchyLevelID,
  v.HierarchyLevelName,
  v.Disabled,
  v.CompanyID,
  v.ParentInstitutionID
FROM
    vInstitution v
  JOIN
    Security.dbo.AccountInstSecurityRole s
  ON s.InstitutionID =
     CASE -- If count is 1 or less, then is unrestricted,
          -- otherwise, different join
       WHEN (SELECT
               COUNT(*)
             FROM
                 vInstitution v
               JOIN
                 Security.dbo.AccountInstSecurityRole s
               ON (v.InstitutionID = s.InstitutionID)
             WHERE
                 v.ParentInstitutionID = @ProviderInstitution
               AND
                 s.AccountID = @LoginID
               AND
                 v.HierarchyLevelID > 1
               AND
                 v.Disabled = 0
            ) <= 1
         THEN v.ParentInstitutionID
       ELSE v.InstitutionID
     END
```

```
WHERE
    v.ParentInstitutionID = @ProviderInstitution
  AND
    s.AccountID = @LoginID
  AND
    v.HierarchyLevelID > 1
  AND
    v.Disabled = 0
ORDER BY
   v.Name
```

Using COALESCE() with a LEFT JOIN

Consider the following scenario. You have a table of customers and another table containing their phone numbers. They can have a home and/or a business phone number. You wish to produce a phone list consisting of the customer's name and either the business number or home number, but not both. You also prefer to see the business number in the event that the customer has both a business and a home number. The list must show whether the number is a business or home phone number. Finally, you do not wish to see any customers who have no phones—after all, this is a phone list. The tables and some sample data are presented in Listing 17-30.

Listing 17-30: Tables and Sample Data for the Phone List

```
CREATE TABLE Customers
(
  CustomerNo int          NOT NULL
                          PRIMARY KEY,
  LastName   varchar (10) NOT NULL,
  FirstName  varchar (10) NOT NULL
)
GO

INSERT Customers VALUES (1, 'Smith', 'John')
INSERT Customers VALUES (2, 'Jones', 'Jim')
INSERT Customers VALUES (3, 'Stockwell', 'Mary')
INSERT Customers VALUES (4, 'Harris', 'Mike')
GO
```

```
CREATE TABLE Telephones
(
  CustomerNo  int      NOT NULL
                       REFERENCES Customers (CustomerNo),
  TelType     char (1) NOT NULL
                       CHECK (TelType IN ('H', 'B')),
  TelephoneNo int      NOT NULL,
                       PRIMARY KEY (CustomerNo, TelType)
)
GO

INSERT Telephones VALUES (1, 'H', 5550000)
INSERT Telephones VALUES (1, 'B', 5550001)
INSERT Telephones VALUES (2, 'H', 5550002)
INSERT Telephones VALUES (3, 'H', 5550003)
INSERT Telephones VALUES (3, 'B', 5550004)
GO
```

The solution requires two `LEFT JOIN`s—both from the Customers table to the Telephones table. One `LEFT JOIN` will pick out the business numbers while the other will pick out the home numbers. The traps associated with `LEFT JOIN`s are outlined in Chapter 1. The filter criteria for the unpreserved table, Telephones, are placed with the `ON` clauses to ensure that only those rows conforming to the filter criteria are chosen.

The trick is to determine which customers have phones. This is where the `COALESCE()` function comes to the rescue. This function takes a comma-delimited list of values and returns the first non-`NULL` value in the list. If all of the values are `NULL`, it returns a `NULL`. For this problem, you can list one column from each of the unpreserved tables. If `COALESCE()` returns a non-`NULL` value, then you have found a customer with a phone.

You also need to present the correct number according to the selection criteria—business numbers are preferred over home numbers. Here, too, you use the `COALESCE()` function and place the columns from the Telephones table in the order (business, home). The final solution is presented in Listing 17-31.

Listing 17-31: Generating the Phone List

```
SELECT
  C.CustomerNo,
  C.LastName,
  C.FirstName,
  COALESCE (TB.TelephoneNo, TH.TelephoneNo) AS TelephoneNo,
  COALESCE (TB.TelType, TH.TelType)         AS TelType
```

```
FROM
    Customers  C
  LEFT JOIN
    Telephones TB ON  C.CustomerNo = TB.CustomerNo
                  AND TB.TelType   = 'B'
  LEFT JOIN
    Telephones TH ON  C.CustomerNo = TH.CustomerNo
                  AND TH.TelType   = 'H'
WHERE
    COALESCE (TB.TelephoneNo, TH.TelephoneNo) IS NOT NULL
```

This example can be extended to add cell phone numbers.

Case-Sensitive Searches (by Umachandar Jayachandran)

If you have a case-insensitive installation but you still want to issue a few case-sensitive queries, then the following trick is for you. Consider the case-insensitive query shown in Listing 17-32, which retrieves authors with the last name "green" from the pubs sample database.

Listing 17-32: Authors with the Last Name "green", Case-Insensitive Search

```
SELECT
  *
FROM
  Authors
WHERE
  au_lname = 'green'
```

In a case-insensitive installation, this query returns one row, although the actual last name stored in the row is "Green" and not "green". This is, by definition, how case-insensitivity should work. To run a case-sensitive search, you can cast both the searched value and the au_lname column to a binary datatype. Since the letter *G* has a different binary value than the letter *g*, "Green" will not be equal to "green", and so the query shown in Listing 17-33 will find no match.

Listing 17-33: Authors with the Last Name "green", Case-Sensitive Search

```
SELECT
  *
FROM
  Authors
WHERE
  CAST (au_lname AS varbinary (40)) = CAST ('green' AS varbinary(40))
```

The problem is that using the CONVERT and CAST functions preclude the use of an index because the searched column is now inside a function. However, by adding a redundant equality comparison between the au_lname column and the searched value as is, as shown in Listing 17-34, the optimizer can use an index on the au_lname column, and *then* check for the case.

Listing 17-34: Authors with the Last Name "green", Case-Sensitive Search Using an Index

```
SELECT
  *
FROM
  Authors
WHERE
    au_lname = 'green'
  AND
    CAST (au_lname AS varbinary (40)) = CAST ('green' AS varbinary (40))
```

Getting Correct Values from @@ Functions

System functions starting with @@ supply very useful information. For example, @@IDENTITY holds the last identity value inserted by the session (see Chapter 6 for details), @@ROWCOUNT holds the number of rows affected by the last statement, and @@ERROR holds an integer number representing the way the last statement that was run finished (see Chapter 7 for details).

The problem with system functions is that most of them are very volatile—almost every statement can change their values, and thus you lose the previous value that was stored in them. For this reason, it is a good practice to store their values in local variables for safekeeping, and later inspect the local variables.

This will be better explained with an example. The script shown in Listing 17-37 creates the T1 table which will be used to generate some errors that you will try to trap.

Listing 17-37: Schema Creation Script for the T1 Table

```
CREATE TABLE T1
(
  pk_col    int NOT NULL PRIMARY KEY CHECK (pk_col > 0),
  ident_col int NOT NULL IDENTITY (1,1)
)
```

Now run the code shown in Listing 17-38, which inserts new rows and checks whether there was a duplicate key violation (error 2627) or a CHECK constraint violation (error 547).

Listing 17-38: Unsuccessful Attempt to Trap Both Primary Key and CHECK Constraint Violations

```
INSERT INTO T1 VALUES(0) -- violate the check constraint

IF @@ERROR = 2627
  PRINT 'PRIMARY KEY constraint violation'
ELSE IF @@ERROR = 547
  PRINT 'CHECK constraint violation'
GO
```

The first `IF` that checks for a `PRIMARY KEY` violation is a statement in its own right. It runs successfully, and thus `@@ERROR` will return 0 when the second `IF` that checks for a `CHECK` constraint violation is run. This code will never trap a `CHECK` constraint violation.

To avoid this problem, you can save the value of `@@ERROR` in a variable, as Listing 17-39 shows.

Listing 17-39: Successful Attempt to Trap Both Primary Key and CHECK Constraint Violations

```
DECLARE
  @myerror    AS int

INSERT INTO T1 VALUES(0) -- violate the check constraint

SET @myerror = @@ERROR

IF @myerror = 2627
  PRINT 'PRIMARY KEY constraint violation'
ELSE IF @myerror = 547
  PRINT 'CHECK constraint violation'
GO
```

Now, suppose you want to capture the number of rows affected by the statement, and the last identity value inserted, by storing `@@ROWCOUNT` and `@@IDENTITY` in your own local variables. You could run the code shown in Listing 17-40.

Listing 17-40: Unsuccessful Attempt to Capture @@IDENTITY, @@ROWCOUNT, and @@ERROR

```
DECLARE
  @myerror    AS int,
  @myrowcount AS int,
  @myidentity AS int

INSERT INTO T1 VALUES(10) -- used to make the next statement cause a PK violation
INSERT INTO T1 VALUES(10) -- PK violation

SET @myidentity = @@IDENTITY
SET @myrowcount = @@ROWCOUNT
SET @myerror    = @@ERROR

PRINT '@myidentity: ' + CAST(@myidentity AS varchar)
PRINT '@myrowcount: ' + CAST(@myrowcount AS varchar)
PRINT '@myerror   : ' + CAST(@myerror AS varchar)
GO
```

The output, shown in Listing 17-41, shows that `@myerror` stores 0 instead of 2627 (primary key violation), and `@myrowcount` mistakenly stores 1 instead of 0. The variable assignment prior to assigning `@@ERROR` to `@myerror` was successful, and thus the original value of `@@ERROR` was lost, and the number of rows affected is 1.

Listing 17-41: Output of Unsuccessful Attempt to Capture @@IDENTITY, @@ROWCOUNT, and @@ERROR

```
@myidentity: 3
@myrowcount: 1
@myerror   : 0
```

To make sure none of the environment variables are lost, you can assign values to all of them in one statement using a `SELECT` statement instead of multiple `SET` statements, as shown in Listing 17-42.

Listing 17-42: Successful Attempt to Capture @@IDENTITY, @@ROWCOUNT, and @@ERROR

```
DECLARE
  @myerror    AS int,
  @myrowcount AS int,
  @myidentity AS int
```

```
INSERT INTO T1 VALUES(10) -- PK violation

SELECT @myidentity = @@IDENTITY,
       @myrowcount = @@ROWCOUNT,
       @myerror    = @@ERROR

PRINT '@myidentity: ' + CAST(@myidentity AS varchar)
PRINT '@myrowcount: ' + CAST(@myrowcount AS varchar)
PRINT '@myerror   : ' + CAST(@myerror AS varchar)
GO
```

The output in Listing 17-43 shows that all of the environment variables were successfully captured.

Listing 17-43: Output of Successful Attempt to Capture @@IDENTITY, @@ROWCOUNT, and @@ERROR

```
@myidentity: 3
@myrowcount: 0
@myerror   : 2627
```

Using PWDCOMPARE() and PWDENCRYPT() in SQL Server 6.5 and 7.0 (by Brian Moran)

This tip demonstrates how to use the undocumented `PWDCOMPARE()` and `PWDENCRYPT()` functions when moving between SQL Server 6.5 and SQL Server 7.0, but the same techniques can be used to build your own password encryption tools in SQL Server 2000. The problem at hand has to do with the fact that SQL Server 6.5's versions of `PWDENCRYPT()` and `PWDCOMPARE()` are not supported in SQL 7.0. Passwords created in 6.5 can't be decrypted in SQL Server 7.0 using `PWDCOMPARE`.

`PWDENCRYPT()` and `PWDCOMPARE()` are internal, undocumented features used by SQL Server to manage passwords. `PWDENCRYPT()` is a one-way hash that takes a clear string and returns an encrypted version of the string. `PWDCOMPARE()` is used to compare an unencrypted string to its encrypted representation to see if it matches. Microsoft cautions people against using internal undocumented features, but sometimes you just can't help yourself. With that said, yes there is a secret, undocumented, relatively unknown way to make "strings" encrypted with the SQL Server 6.5 version of `PWDENCRYPT()` work with the SQL Server 7.0 version of `PWDCOMPARE()`.

Let's assume you've built an application that stores a four-character PIN number that is used within the application for a simple password check. You could have spent a bunch of time writing your own encryption algorithms, or you could have used the Microsoft Cryptography API, but you're a rebel so you decided to use the undocumented and unsupported `PWDENCRYPT()` and `PWDCOMPARE()` functions. You use

PWDENCRYPT() to encrypt the PIN and then use PWDCOMPARE() to check a clear text version of the PIN against the encrypted version to see if they match. Everything worked perfectly until you tried to upgrade to SQL Server 7.0, at which time the PWDCOMPARE() function started returning FALSE even when the clear text and encrypted versions of the PIN string did match. How do you make the old encrypted PIN numbers work with PWDCOMPARE() when you upgrade to a newer version of SQL Server?

> **NOTE** *Listing 17-47 at the end of this section includes a code sample for using these functions in a SQL Server 7.0 or 2000 environment.*

SQL Server 7.0 must have a way to compare the old SQL Server 6.5 passwords encrypted with PWDENCRYPT() since the passwords from an upgraded SQL Server 6.5 database work fine. Doing a little detective work with the T-SQL source code behind sp_addlogin and sp_password can give you all the answers you are looking for. (Reading system procedures is always a great way to learn new tricks!) These two stored procedures both make internal use of the SQL Server encryption functions, and they both need to deal with SQL Server 6.5 versions of passwords.

Reading the sp_addlogin T-SQL code and supporting Books Online documentation shows a possible value of 'skip_encryption_old' for the @PWDENCRYPT() parameter, and the Books Online tell us this value means, "The password is not encrypted. The supplied password was encrypted by an earlier version of SQL Server. This option is provided for upgrade purposes only."

Reading further through the T-SQL code for sp_addlogin clearly shows that the SQL Server 7.0 version of PWDENCRYPT() is not applied to the @passwd string if @encryptopt = 'skip_encryption_old'. But SQL Server 7.0 does apply some CONVERT() gymnastics to the @passwd parameter to store the string in the "new" SQL Server 7.0 datatype format for passwords. The relevant snippet of T-SQL code from sp_addlogin is shown in Listing 17-44.

Listing 17-44: Excerpt from sp_addlogin

```
ELSE IF @encryptopt = 'skip_encryption_old'
BEGIN
    SELECT @xstatus = @xstatus | 0x800,    -- old-style encryption
    @passwd = CONVERT(sysname,CONVERT(varbinary(30),
                        CONVERT(varchar(30), @passwd)))
```

Pay close attention to the three-step CONVERT() process that SQL Server makes the @passwd parameter jump through. You'll be reusing it shortly.

Now take a look at the T-SQL code for `sp_password`. One of the checks is to see if the old password matches. You'll see the snippet of code shown in Listing 17-45.

Listing 17-45: Excerpt from sp_password

```
PWDCOMPARE(@old, password,
          (CASE WHEN xstatus & 2048 = 2048 THEN 1 ELSE 0 END))
```

This shows that the SQL Server 7.0 version of `PWDCOMPARE()` now takes three parameters rather than two parameters like the SQL Sever 6.5 version used. Some experimentation helped me understand that the third parameter is optional and defaults to 0. When set to 0, `PWDCOMPARE()` uses the "new" SQL Server 7.0 algorithm, but setting this parameter to 1 tells SQL Server 7.0 to use the "old" SQL Server 6.5 version of the `PWDCOMPARE()` algorithm.

Listing 17-46 has a stored procedure sample that shows how you can leverage these tricks to use effectively "old" SQL Server 6.5 encrypted strings with the SQL Server 7.0 version of `PWDCOMPARE()`. This stored procedure assumes your application asks a user to provide their Social Security Number (SSN) and "secret" PIN, which were stored in a table called MyTable. The value of the PIN had previously been encrypted using the SQL Server 6.5 version of `PWDENCRYPT()`.

Listing 17-46: Creation Script for the CompareSQL65EncryptedString Stored Procedure

```
CREATE PROCEDURE CompareSQL65EncryptedString
(
  @SSN char(9),
  @pin  char(4),
  @return int OUTPUT)
AS
IF EXISTS
(
SELECT
  *
FROM MyTable (NOLOCK)
WHERE
    SSN = @ssn
  AND
    PWDCOMPARE(@pin,
      CONVERT(sysname,
        CONVERT(varbinary(30),CONVERT(varchar(30),pin))),
      1) = 1
)
  SELECT @return = 1
```

```
ELSE
  SELECT @return = 0
GO
```

For those of you running SQL Server 7.0 or 2000, the code snippet in Listing 17-47 shows how to use the current versions of `PWDENCRYPT()` and `PWDCOMPARE()` to create your own one-way hash password-management algorithms.

Listing 17-47: Using the PWDENCRYPT() and PWDCOMPARE() Functions

```
DECLARE
  @ClearPIN     varchar (255),
  @EncryptedPin varbinary(255)

SELECT
  @ClearPin = 'test'

SELECT
  @EncryptedPin = CONVERT (varbinary(255), PWDENCRYPT (@ClearPin))

SELECT
  PWDCOMPARE (@ClearPin, @EncryptedPin, 0)
```

The final `SELECT` statement will return 1, indicating `TRUE`. In other words, `@ClearPin` is put through a one-way encryption hash, and SQL Server tells you the unencrypted string matches the encrypted version.

Creating Sorted Views

You cannot sort a view, right? Not until SQL Server 7.0! You can now use the `TOP n PERCENT` feature of the `SELECT` statement to take all of the rows. How? Just take `TOP 100 PERCENT`. Check out the code in Listing 17-48.

Listing 17-48: Creating a Sorted View

```
CREATE VIEW SortedView
AS
SELECT TOP 100 PERCENT
  C.CompanyName,
  O.OrderDate
```

```
FROM
    Customers C
  JOIN
    Orders    O ON O.CustomerID = C.CustomerID
ORDER BY
  O.OrderDate
GO
```

See Chapter 8 for more details on the effects of this technique.

Getting Rows in Order

In the relational model, table rows don't have any specific order. According to the ANSI standards, a query that uses the `ORDER BY` clause doesn't return a table; rather, it returns a cursor. This is why an `ORDER BY` clause is not allowed in a view, and why the `TOP` clause is not ANSI compliant. Both deal with rows in a specific order.

The `TOP` T-SQL extension and the special needs it can answer are covered in Chapter 4. There are still other needs, though, dealing with rows with a specific order, that simple `TOP` queries cannot answer. The following sections deal with examples of such needs.

Getting Rows m to n

If you order the authors in the pubs sample database by author ID, you can use a simple `TOP` query to ask for the first five authors. However, if you want the second group of five authors—authors six to ten—things become a little bit more complex.

You can use the ANSI-compliant query shown in Listing 17-49. For each author, this query performs a correlated subquery that calculates the number of authors with author IDs that are smaller than or equal to the current author ID.

Listing 17-49: Getting Rows m to n in One Query

```
SELECT
  *
FROM
  Authors AS A1
```

```
WHERE
  (
    SELECT
      COUNT(*)
    FROM
      Authors AS A2
    WHERE
      A2.au_id <= A1.au_id
  ) BETWEEN 6 AND 10
ORDER BY
  au_id
```

Note that this query has poor performance, as it needs to scan the Authors table as many times as there are authors in the table. This query also requires the column that you order by, in this case the au_id column, to be unique. We can improve our query's performance by splitting our solution into two steps. First we can place the authors' rows in a temporary table, along with their ordinal position according to the author ID column. We can achieve this by using the `IDENTITY()` function (the `IDENTITY()` function is discussed in detail in Chapter 6), as Listing 17-50 shows.

Listing 17-50: Placing the Authors in a Temporary Table Along with Their Ordinal Positions

```
SELECT
  IDENTITY (int, 1, 1) AS rownum,
  *
INTO
  #TmpAuthors
FROM
  Authors
ORDER BY
  au_id
```

You can now issue a simple query to retrieve authors six to ten, as Listing 17-51 shows.

Listing 17-51: Retrieving Authors Six to Ten from the Temporary Table

```
SELECT
  *
FROM
  #TmpAuthors
WHERE
  rownum BETWEEN 6 AND 10
```

Note that this technique did not always work prior to SQL Server 2000 (tested on SQL Server 7.0, Service Pack 2). When using the `SELECT INTO` statement, sometimes the `Identity` values were calculated prior to the sort, making this solution improper for the problem at hand. SQL Server 2000 solved this problem, and if you examine the execution plan of the `SELECT INTO` statement, you can actually see that a sort is performed prior to calculating the `IDENTITY` value. In both versions, however, if you create the temporary table manually with an additional `IDENTITY` column and use an `INSERT SELECT` statement to populate it, the execution plans show that a sort is performed prior to calculating the `IDENTITY` values, making this solution valid for both versions. Hopefully this bug will be resolved in SQL Server 7.0 in one of the next service packs.

If you want to do everything in a single statement, you can use the `TOP` feature of the `SELECT` statement. First, you need to determine the first ten authors, which you do with a `TOP 10`, and you `ORDER BY au_id ASC`. This `SELECT` then acts as a derived table from which you can do a `TOP 5`, this time with `ORDER BY au_id DESC`. This gives you the second group of five; however, it is sorted in reverse order to what is desired. This result is then used as a derived table, where you do a regular `SELECT` and just sort the rows with `au_id ASC`. The solution is presented in Listing 17-52.

Listing 17-52: Retrieving Authors Six through Ten

```
SELECT
  *
FROM
(
  SELECT TOP 5
    *
  FROM
  (
    SELECT TOP 10
      *
    FROM
      Authors
    ORDER BY
      au_id ASC
  ) X
  ORDER BY
    au_id DESC
) Y
ORDER BY
  au_id ASC
```

Getting the First n Rows for Each Occurrence of…

Things can get even more complex than the previous example. Suppose you want to provide the first three orders for each customer for all orders shipped to the U.S. (order data appears in the Orders table in the Northwind sample database). You can use a query similar to, but slightly more complex than, the one used in the previous section to supply a solution using a single query. The solution is shown in Listing 17-53.

Listing 17-53: Getting the First Three Orders for Each U.S. Customer in One Query

```
SELECT
  *
FROM
  Orders AS O1
WHERE
    ShipCountry = 'USA'
  AND
    (
      SELECT
        COUNT(*)
      FROM
        Orders AS O2
      WHERE
          ShipCountry   = 'USA'
        AND
          O2.CustomerID = O1.CustomerID
        AND
          O2.OrderID   <= O1.OrderID
    ) <= 3
ORDER BY
  CustomerID,
  OrderID
```

This query suffers from the same problems as the one in the previous section, mainly from poor performance. It incurred a scan count of 123 and had 2,329 logical reads against the Orders table.

However, the problem can be approached in a totally different way. For example, if you have the exact list of customer IDs, you can perform a `UNION ALL` between a number of queries, each of which retrieves the first three orders for a certain customer. Listing 17-54 shows a template for such a query.

Listing 17-54: Getting the First Three Orders for Each Customer, for a Known List of Customers

```
SELECT
  *
FROM
  (
    SELECT TOP 3
      *
    FROM
      Orders
    WHERE
        ShipCountry = 'USA'
      AND
        CustomerID = <first_cust>
    ORDER BY
      CustomerID,
      OrderID
  ) AS T1

UNION ALL

SELECT
  *
FROM
  (
    SELECT TOP 3
      *
    FROM
      Customers
    WHERE
        ShipCountry = 'USA'
      AND
        CustomerID = <second_cust>
    ORDER BY
      CustomerID,
      OrderID
  ) AS T2

UNION ALL
...
UNION ALL
```

```
SELECT
  *
FROM
  (
    SELECT TOP 3
      *
    FROM
      Customers
    WHERE
        ShipCountry = 'USA'
      AND
        CustomerID = <last_cust>
    ORDER BY
      CustomerID,
      OrderID
  ) AS Tn
```

To make this query dynamic so that it will run for an unknown list of customers simply as they appear in the Orders table, you can use this template to build the query in a variable, and execute it dynamically, as shown in Listing 17-55. This script iterates through all customers in a loop, retrieves a customer with a higher customer ID in each iteration, and adds another `SELECT` statement to the `UNION ALL` query.

Listing 17-55: Getting the First Three Orders for Each Customer, for an Unknown List of Customers

```
DECLARE
  @lastindid AS char (5),
  @i         AS int,
  @cmd       AS varchar (8000)

SET @cmd = ''
SET @i = 1

SELECT
  @lastindid = MIN (CustomerID)
FROM
  Orders
WHERE
  ShipCountry = 'USA'
```

```
WHILE @lastindid IS NOT NULL
BEGIN
  SET @cmd = @cmd +
  'SELECT * FROM ' +
    '(SELECT TOP 3 * FROM Orders ' +
     'WHERE ShipCountry = ''USA'' AND CustomerID = ''' + @lastindid + ''' ' +
     'ORDER BY CustomerID,OrderID) AS T' +
     CAST(@i AS varchar) + CHAR (13) + CHAR(10)

  SELECT
    @lastindid = MIN (CustomerID),
    @i         = @i + 1
  FROM
    Orders
  WHERE
      ShipCountry = 'USA'
    AND
      CustomerID  > @lastindid

  IF @lastindid IS NOT NULL
    SET @cmd = @cmd + 'UNION ALL' + CHAR (13) + CHAR(10)

END

PRINT @cmd -- just for debug
EXEC (@cmd)
```

You might think that I/O performance is improved significantly as the I/O statistics for the dynamically constructed `UNION ALL` query show a scan count of 13 and 99 logical reads, but you need to take into account the I/O generated as a result of the loop that dynamically constructs the `UNION ALL` statement. The total logical reads are very high due to the loop. This solution might not render better performance, but it may give you some ideas about constructing statements dynamically.

Now, for the pièce de résistance. Have a go at the query in Listing 17-56.

Listing 17-56: Getting the First Three Orders for Each Customer for an Unknown List of Customers, Using a Correlated Subquery

```
SELECT
  O1.*
FROM
  Orders O1
WHERE
    O1.ShipCountry = 'USA'
  AND
    O1.OrderID IN
(
  SELECT TOP 3
    O2.OrderID
  FROM
    Orders    O2
  WHERE
      O2.ShipCountry = 'USA'
    AND
      O2.CustomerID  = O1.CustomerID
  ORDER BY
    O2.OrderID
)
ORDER BY
  O1.CustomerID,
  O1.OrderID
```

The scan count is 123 while the logical reads are 927. This query uses the `TOP` feature inside a correlated subquery with an `IN` predicate. The outer query needs to find those OrderIDs that correspond to the first three OrderIDs for each CustomerID. The correlation is on CustomerID. This solution gives you the rows you want for the lowest query cost.

Top Countries per Employee

You have seen, in Chapter 2, the use of correlated subqueries, including those on the `HAVING` predicate of a `GROUP BY` clause. This problem requires you to find the country for which each employee has the most orders shipped. You will use the Orders table of the Northwind database. You can use a correlated subquery on the `HAVING` predicate of the `GROUP BY` clause, as shown in Listing 17-57.

Listing 17-57: Determining the Country that Keeps Each Employee the Busiest

```
SELECT
  O1.EmployeeID,
  O1.ShipCountry,
  COUNT (*) AS Orders
FROM
    Orders O1
GROUP BY
    O1.EmployeeID,
    O1.ShipCountry
  HAVING
    COUNT (*) =
(
  SELECT TOP 1
    COUNT (*) AS Orders
  FROM
    Orders O2
  WHERE
    O2.EmployeeID = O1.EmployeeID
  GROUP BY
    O2.EmployeeID,
    O2.ShipCountry
  ORDER BY
    Orders DESC
)
ORDER BY
  O1.EmployeeID
```

The `COUNT(*)` in the `SELECT` list of the outer query is there just to provide supporting information. The problem simply required finding out who served which country the most.

Are You Being Served?

This next problem is a variation on the previous one. It requires you to find the employee who processes the most shipments for each country. Again, you will use the Orders table in the Northwind database. The first try solves the problem in a single `SELECT`, with a correlated subquery on the `GROUP BY` clause. This is shown in Listing 17-58.

Listing 17-58: Employee Who Serves Each Country the Most, First Try

```
SELECT
  O1.ShipCountry,
  O1.EmployeeID,
  COUNT (*) AS Orders
FROM
    Orders O1
GROUP BY
    O1.ShipCountry,
    O1.EmployeeID
  HAVING
    COUNT (*) =
(
  SELECT TOP 1
    COUNT (*) AS Orders
  FROM
    Orders O2
  WHERE
    O2.ShipCountry = O1.ShipCountry
  GROUP BY
    O2.ShipCountry,
    O2.EmployeeID
  ORDER BY
    Orders DESC
)
ORDER BY
  O1.ShipCountry
```

The inner query has to calculate the count for every occurrence of ShipCountry and EmployeeID. The outer query is also calculating the counts. This gives a scan count of 22 and 36,042 logical reads. It solves the problem, but perhaps there is a way to reduce the I/O.

Since the counts have to be used twice, you can do the calculation once and store the results in a temporary table. This is done in Listing 17-59.

Listing 17-59: Employee Who Serves Each Country the Most, Second Try

```
SELECT
  O1.ShipCountry,
  O1.EmployeeID,
  COUNT (*) AS Orders
INTO
  #Temp
FROM
    Orders O1
GROUP BY
    O1.ShipCountry,
    O1.EmployeeID

SELECT
  T1.*
FROM
  #Temp T1
WHERE
  T1.Orders =
(
  SELECT
    MAX (T2.Orders)
  FROM
    #Temp T2
  WHERE
      T2.ShipCountry = T1.ShipCountry
)
ORDER BY
  T1.ShipCountry
```

The correlated subquery no longer involves the `GROUP BY` clause. The total scan count is 2, while the logical reads are just 24. The relative query cost for this version is 23.04 percent versus 76.96 percent for the previous version. Now you're cookin'!

Can't improve on perfection? You can dispense with the temporary table by casting its query as a derived table. The same derived table appears twice in the statement. However, there is only one scan and 21 counts. Looks like the optimizer is smart enough not to consider it twice. See the code in Listing 17-60.

Listing 17-60: Employee Who Serves Each Country the Most, Third Try

```
SELECT
  T1.*
FROM
  (SELECT
     O1.ShipCountry,
     O1.EmployeeID,
     COUNT (*) AS Orders
   FROM
     Orders O1
   GROUP BY
     O1.ShipCountry,
     O1.EmployeeID) T1
WHERE
  T1.Orders =
(
  SELECT
    MAX (T2.Orders)
  FROM
    (SELECT
       O1.ShipCountry,
       O1.EmployeeID,
       COUNT (*) AS Orders
     FROM
       Orders O1
     GROUP BY
       O1.ShipCountry,
       O1.EmployeeID) T2
  WHERE
      T2.ShipCountry = T1.ShipCountry
)
ORDER BY
  T1.ShipCountry
```

SQL Puzzle 17-1: Top Gun: The Best of the Best

Congratulations! You have reached the final puzzles of this book. By now, you are ready to take on anything. Resist the urge to go to the Answers section, even though it is only a few pages away. It will be worth it.

In the two previous problem scenarios, you saw which country kept each employee the busiest and who the best employee was, broken down by the country

served. This puzzle requires you to find the standings of the top sellers. In other words, you are interested only in those employees who were the number one sellers in a country. The employee who appears as numero uno the most is the top employee.

SQL Puzzle 17-2: Filling a Table with Magic Square Data in T-SQL

This puzzle encapsulates various T-SQL elements and requires creativity. It deals with magic squares, a subject that is the source for many mathematical puzzles. The solution to the magic squares problem will be found in the T-SQL world.

What Is a Magic Square?

A magic square is a square matrix—a matrix with the same number of rows and columns for which if you sum up the values in each row, column, and diagonal you always get the same number. For example, a 3 × 3 magic square might look like this:

8	1	6
3	5	7
4	9	2

Notice that if you sum up the values in each row, column, and diagonal, you always get 15. There is a simple method for filling an odd-sized magic square on paper, and you can use that method to fill the magic square's data in a table with T-SQL. Filling an even-sized magic square is too complex to handle with T-SQL, so we will not consider even-sized magic squares in this puzzle.

The rules for filling an odd-sized magic square are very simple:

1. Write the first number (1) in the middle cell of the first row.

2. Write the next consecutive number in the cell that is one cell to the right, and one cell above the current cell. If this cell is occupied, go to the cell beneath the current cell. When you bump to an edge of the magic square, you go all the way around. Think of the square as a ball where all of the edges of the rows, columns, and diagonals are connected to each other.

3. Repeat the second step until the magic square is full.

Figure 17-1 shows the steps involved in filling the magic square presented earlier.

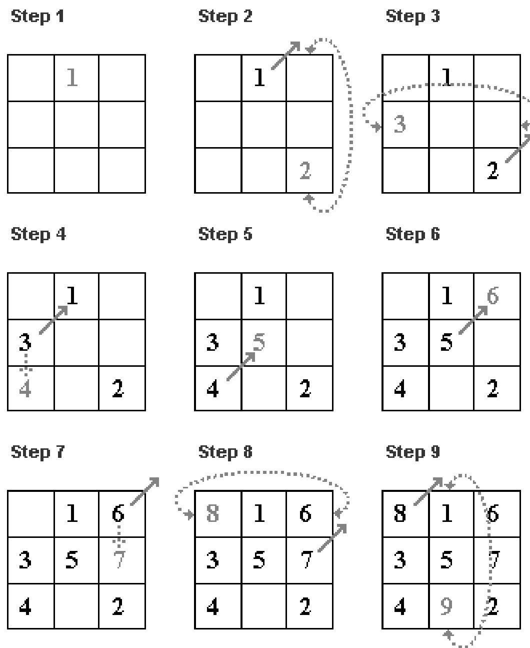

Figure 17-1: Filling the magic square

Representing a Magic Square in a Table

A normalized table will represent the magic square, with each row in the table representing a cell in the square. Each row in the table records the cell's row and column coordinates in the square, as well as its value. This table structure allows you to represent a magic square of any given size.

```
CREATE TABLE MagicSquare
(
  row   int NOT NULL,
  col   int NOT NULL,
  value int NOT NULL,
  CONSTRAINT PK_magicsquare_row_col PRIMARY KEY (row, col),
  CONSTRAINT UNQ_magicsquare_value  UNIQUE      (value)
)
```

SQL Puzzle 17-2-1: Filling a Table with a Magic Square's Data in T-SQL

Your first task is to write a stored procedure that accepts the magic square's size as a parameter and fills your table with its data. First, you need to clear all existing rows in the table. Second, you will insert the first number in the middle cell of the first row. Third and last, you will insert the rest of the numbers in a `WHILE` loop. To make it a real challenge, you will limit yourself to using a non-blocked `WHILE` loop with a single `INSERT` statement.

The template of the `FillMagicSquare` stored procedure looks like this:

```
CREATE PROC FillMagicSquare
  @size AS int
AS

DELETE MagicSquare

-- Insert the first number in the middle cell in the first row
INSERT INTO MagicSquare
  (row, col, value)
VALUES
  (...) -- ?

-- Insert the rest of the numbers in a while loop
WHILE ... -- ?
  INSERT INTO MagicSquare(row, col, value)
    ... -- ?
GO
```

SQL Puzzle 17-2-2: Displaying the Magic Square as a Cross-Tab Table

Your second task is to display the magic square in a more meaningful cross-tab format. For example, if the size of your magic square is 3, the output should look like this:

ol1	Col2	Col3
8	1	6
3	5	7
4	9	2

It shouldn't be too hard to generate a cross-tab query that provides such an output for a known magic square size, but your solution should produce such an output without a prior knowledge of the size of the magic square stored in the table.

SQL Puzzle 17-2-3: Checking whether the Table Represents a Magic Square Correctly

Your last task is to write a single `IF` statement that checks whether the table represents a magic square correctly or not. You are not allowed to use a complex condition, such as `OR`s or `AND`s. The template for your `IF` statement looks like this:

```
IF ... -- ?
  PRINT 'This is a Magic Square. :-)'
ELSE
  PRINT 'This is not a Magic Square. :-('
```

You can assume that the table consists of a complete square—that is, that all cell coordinates exist in the table.

The answers to these puzzles can be found on pages 721–733.