The Baker's Dozen: 13 Productivity Tips for Transact-SQL 2005

SQL Server 2005 offers T-SQL language features that can improve your productivity.

by Kevin S. Goff

Microsoft implemented many new features in SQL Server 2005, including an impressive set of language enhancements. From new language statements for SQL-99 compatibility to new features in response to customer requests, Transact-SQL 2005 helps to increase developer productivity. This article covers most of the new language features by posing a statement/scenario and then provide some code samples to show how you can use T-SQL 2005 to address the problem. You'll also get a brief glance at Visual Studio Team Edition for Database Professionals, a product that helps a development team to manage databases. Finally, I'll give you a sneak preview of some features in the next scheduled version of SQL Server (SQL Server 2008, "Katmai").

Beginning with the End in Mind

I speak at community events (MSDN Code Camp, user groups, etc.) at least once a month. One of the more popular sessions (as well as one of my favorites) is called "T-SQL for Developers."

I'm a big believer in plenty of code samples to demonstrate functionality, and so my goal is to provide a healthy number of code samples for each of the following:

- The PIVOT statement
- Common table expressions (CTEs) and recursive queries (part 1 of 2)
- CTEs and recursive queries (part 2 of 2)
- OUTPUT and OUTPUT INTO, to gain immediate access to the system INSERTED and DELETED tables
- Isolation levels (part 1 of 2)
- Isolation levels (part 2 of 2)
- New XQUERY capabilities to handle variable number of selections
- Variable TOP N APPLY and table-valued UDFs
- RANKING and PARTITIONING
- New TRY…CATCH capabilities and RAISING errors
- INTERSECT/EXCEPT
- Flexible UPDATE procedures

All code samples will work using the AdventureWorks database that comes with SQL Server 2005.

Tip 1: PIVOT

Scenario: You want to query a table of vendor orders and group the order amounts by quarter for each vendor.

Application developers often need to convert raw data into some type of analytical view, such as sales by month or quarter or the brackets of an aging report. Prior to SQL Server 2005, you would often have to examine each row with a CASE statement to place raw data into a column.

SQL Server 2005 introduced the PIVOT statement, arguably the most well-known new language feature. PIVOT allows you to (as the name implies) turn rows of raw data into columns. The code below shows a basic example for PIVOT: a query against the Purchase

Application developers often need to convert raw data into a

Order tables in AdventureWorks that summarizes order amounts by quarter:

result set that represents an analytical view. The PIVOT statement makes this task much easier.

```
USE AdventureWorks
GO
WITH OrdCTE AS (
SELECT VendorID,  DatePart(q,OrderDate) AS OrderQtr,
   (OrderQty * UnitPrice)  AS OrderTot
FROM Purchasing.PurchaseOrderHeader POHdr
   JOIN Purchasing.PurchaseOrderDetail PODtl
ON POHdr.PurchaseOrderID = PODtl.PurchaseOrderID )
SELECT VendorID,[1] AS Q1,[2] AS Q2,[3] AS Q3,[4] AS Q4 FROM OrdCTE
   PIVOT (SUM(OrderTot) FOR OrderQtr IN ([1],[2],[3],[4])) AS X
-- You can use MAX instead, if you want the top order for each Qtr
```

Note the syntax for the PIVOT statement: You essentially need to tell PIVOT three pieces of information:

- Which column you are pivoting on (OrderTot).
- Which column you want to examine (OrderQtr, from the Quarter DatePart of the OrderDate), to determine how to pivot.
- The possible values of the column you want to examine (the only possible values of a Quarter DatePart are 1, 2, 3, or 4):

```
SELECT VendorID,
  [1] AS Q1,[2] AS Q2,
  [3] AS Q3,[4] AS Q4
FROM OrdCTE
  PIVOT (SUM(OrderTot) FOR OrderQtr IN
      ([1],[2],[3],[4])) AS X
```

A few additional notes on PIVOT:

- The list of values in the IN clause must be static. Microsoft's implementation of PIVOT does not directly support dynamic queries. If you need to determine these values dynamically at runtime, you must construct the entire SQL statement as a string and then use Dynamic SQL. If you frequently need to generate PIVOT tables dynamically, you may want to look at GeckoWare's SQL CrossTab Builder product.
- You must specify the column you are pivoting on (Ordertot in this case) as a scalar expression (e.g. MAX(), SUM(), etc.).

Note that summary example in this section contained a new language construct: WITH (name). This is a common table expression (CTE), which you can think of as a dynamic view. I'll cover CTEs in the next few tips.

Tip 2: CTEs and Recursive Queries (1 of 2)

Scenario: You have a hierarchy of product items, groups, and brands, all in a single table. Each row has a parent key that points to the row's parent. For any single row, you want to know all the row's parent rows, or all the row's child rows.

SQL Server 2000 developers commonly requested the ability to more easily query against hierarchical data. Many databases store hierarchical data, such as databases for menu options, bill of materials, and geographic hierarchies. E-commerce applications in particular will often store a company's product hierarchy in a single table, where each row contains a pointer key to the row's parent. The table might look something like the contents of Table 1.

Recursive queries are powerful—so powerful that you may not always see all their power at once. Next time you think you need to write code to loop through data, consider a CTE and a recursive query instead.

Table 1: The table shows a sample product hierarchy for an E-commerce application.

PrimaryKey	ParentKey	Description

1	NULL	Brand A
2	NULL	Brand B
3	NULL	Brand C
4	1	Brand A, Group 1
5	1	Brand A, Group 2
6	2	Brand B, Group 3
7	4	Brand A, Group 1, Item a
8	6	Brand B, Group 3, Item b

Imagine writing even a simple query to determine all the parent or child records from the table, especially when there is no limit to the number of levels. Fortunately, the common table expression and recursive query capabilities in SQL Server 2005 make this very easy. The key is to use a set of language constructs that allow you to first retrieve your main or "core" selection(s), and then a second set of queries or processes that takes the result of your core selections and repeatedly traverses (either up or down) a hierarchy to find subsequent matches.

Listing 1 populates a simple product hierarchy, and shows how to use recursive query capabilities to determine all the parent and child rows for a specific row.

A basic recursive query consists of three parts:

First, you must define a common table expression (CTE). Note the semicolon at the beginning of the statement:

```
; WITH ProductCTE  AS
```

Second, you construct your main or core (or "anchor") query:

```
(SELECT ID, Name, ParentID FROM @tProducts
WHERE Name = @cSearch
```

So far, easy stuff. The major work occurs in the third step. You want to add a UNION ALL statement and construct a query that references the CTE. Notice that the query does a JOIN on the ProductCTE based on a match between each ID and the ParentID. SQL Server 2005 will continue to repeat this query (hence the name "recursive" until it exhausts all searches):

```
-- Recursive query
SELECT Prod.ID, Prod.Name, Prod.ParentID
FROM @tProducts Prod
    INNER JOIN ProductCTE ON ProductCTE.ID =
    Prod.parentID )
```

Now that you've seen a basic example of recursive queries, I'll move on to a second example that shows their power.

Tip 3: CTEs and Recursive Queries (1 of 2)
Scenario: You need to take a table of vendor orders and produce a flat result set for a weekly report or chart for the first quarter of 2004. The result set must contain one row for each vendor and week-ending date, with the sum of orders for that week (even if zero).

If you've ever written queries for weekly reports or weekly charts, you know that creating the full result set can be the trickiest part. Some reporting and charting systems require a matrix or even some form of a Cartesian product as the data source.

In the example above, you need to create a result set with one record per week per vendor account, even if the account did not have any orders for the week. You may be tempted to write a loop (either in T-SQL or in the application layer) to produce a table with the accounts and dates, and then join that against the orders themselves while searching for any orders between each week. However, you can also construct a recursive query to perform all the work in a stored procedure.

Before you write such a recursive query, you need to construct a function that will convert any date to an end of the week date. Orders may take place any day of the week, but you need to convert any date to a week-ending date (normally Saturday, as most businesses treat a business week as Sunday through Saturday. The following code is a SQL user-defined function (UDF) to convert any day to a Saturday date—so if you pass '6-13-2006' as a parameter, you'll get '6-16-2006' as the return value from the UDF:

```
ALTER FUNCTION [dbo].[GetEndOfWeek]
   (@dDate DateTime)
-- Converts date to the Saturday date for the week
-- So 6-12-07 becomes 6-16-07
RETURNS DateTime AS
BEGIN
DECLARE @dRetDate DateTime
SET @dRetDate =
                @dDate + ( 7-DATEPART(WeekDay, @dDate))
RETURN @dRetDate
END
```

Next, you need to construct a matrix of dates and vendors. The report calls for a weekly result set by vendor for the first quarter of 2004. Since you may not know the first Saturday in 2004 (nor the last Saturday), you can simply declare two variables for the '1-1-2004' and '3-31-2004' strings, and use the UDF to convert them to the actual Saturday dates:

```
DECLARE @StartDate DATETIME, @EndDate DATETIME
SET @StartDate = dbo.GetEndOfWeek('1-1-2004')
-- First Saturday
SET @EndDate = dbo.GetEndOfWeek('3-31-2004')
-- Last Saturday
```

After that, you'll next create two CTEs, one for a range of dates and a second for a list of vendors with at least one order. You can actually combine CTE declarations into a single large statement and delimit them with commas. The first CTE, DateCTE, will contain a row for each Saturday date in the date range. Again here is where you may be accustomed to writing some kind of loop to create a list. But now you can use recursive queries. To do so, you declare your CTE and set your main query, which simply does a SELECT of the single starting date:

```
; WITH DateCTE (WeekEnding)  AS
     (SELECT @StartDate AS WeekEnding
```

At this point, the DateCTE has only one row in it, containing the '1-3-2004' value. The recursive query will repeatedly query DateCTE for values greater than 7 days beyond each entry, which are less than the ending date. So the recursive query will eventually build a row in DateCTE for '1-10-2004', '1-17-2004', etc., all the way to the ending date:

```
UNION ALL
SELECT WeekEnding + 7 AS WeekEnding
FROM DateCTE
WHERE WeekEnding < @EndDate ),
```

Now that you've built the DateCTE, you can build a simple CTE for each vendor with at least one order:

```
VendorList (VendorID)   AS
  (SELECT VendorID FROM
     Purchasing.PurchaseOrderHeader
GROUP BY VendorID)
```

Finally, you can query against the two CTEs and the PO table to build the final result. Notice that the final query calls the GetEndOfWeek UDF to convert each OrderDate to a Saturday date, to match it up against the corresponding end of week date in DateCTE. Listing 2 contains the complete source code for the CTEs and the final query, but here's the relevant portion:

```
SELECT Matrix.VendorID,WeekEnding,
    SUM(COALESCE(Orders.TotalDue,0)) AS WeeklySales
FROM (SELECT VendorID, WeekEnding FROM
   DateCTE, VendorList) AS Matrix
LEFT JOIN Purchasing.PurchaseOrderHeader Orders
ON dbo.GetEndOfWeek(Orders.OrderDate) = Matrix.WeekEnding
   AND Orders.VendorID = Matrix.VendorID
GROUP BY Matrix.VendorID,Matrix.WeekEnding
```

Recursive queries are powerful—so powerful that you may not always see all their power at once. Next time you think you need to write code to loop through data, consider a CTE and a recursive query instead.

Tip 4: *OUTPUT* and *OUTPUT INTO*

Scenario: You are inserting rows into a table and want to immediately retrieve the entire row, without the need for another SELECT *statement, or a round trip to the server, and without using* SCOPE_IDENTITY*. Additionally, when you* UPDATE *rows in a table, you want to immediately know the contents of the row both before and after the* UPDATE*—again, without the need for another set of queries.*

Have you ever needed to do a SELECT statement after an INSERT or UPDATE, just to retrieve a fresh copy of the row you added/changed? Related question: Have you ever been frustrated that you can't access the INSERTED and DELETED system tables outside of a database trigger?

If you answered yes to either question, then you may like the next tip. SQL Server 2005 introduced a programming shortcut called OUTPUT that you can include in any INSERT/UPDATE/DELETE statement. OUTPUT allows you to specify values from the INSERTED and DELETED system tables so that you can include any or all "old value - new value" pieces of data.

OUTPUT and OUTPUT INTO are nice capabilities to get immediate feedback on what has been added or changed. But don't use them in place of audit trail triggers.

Here are two examples that show how to use OUTPUT for both INSERT and UPDATE statement:

```
DECLARE @Invoice TABLE
    (PK int identity, InvDate DateTime, InvAmt Decimal(14,2))
DECLARE @InvDate DATETIME
SET @InvDate = GETDATE()
INSERT INTO @Invoice OUTPUT INSERTED.* VALUES ( GETDATE(),1000)
DECLARE @InsertedRows TABLE
    (PK int, InvDate DateTime, InvAmt Decimal(14,2))
INSERT INTO @Invoice OUTPUT INSERTED.* INTO @InsertedRows
VALUES ( GETDATE(),2000)
INSERT INTO @Invoice OUTPUT INSERTED.* INTO @InsertedRows
VALUES ( GETDATE(),3000)
SELECT * FROM @InsertedRows



DECLARE @tTest1 TABLE ( TestCol int,Amount decimal(10,2))
INSERT INTO @ttest1 VALUES (1, 100)
INSERT INTO @ttest1 VALUES (2, 200)
INSERT INTO @ttest1 VALUES (3, 300)
UPDATE     TOP(2)  @tTest1     SET Amount = Amount * 10
OUTPUT DELETED.*,inserted.*
-- This allows you to access the DELETED and INSERTED system tables
-- that were previously available only inside triggers
```

For an INSERT, you just add the OUTPUT statement before the VALUES statement. If you are inserting into a table with an identity column, the OUTPUT result will include the new identity column value, as well as any default values or calculated columns from the table. This is much simpler than using SCOPE_IDENTITY to retrieve the full record after the insert:

```
INSERT INTO @Invoice OUTPUT INSERTED.*
VALUES ( GETDATE(),1000)
```

If you are inserting several records, you can use OUTPUT INTO to insert the results to a table variable, one insert at a time:

```
DECLARE @InsertedRows TABLE
 (PK int, InvDate DateTime, InvAmt
Decimal(14,2))
INSERT INTO @Invoice
OUTPUT INSERTED.* INTO @InsertedRows
VALUES ( GETDATE(),2000)
INSERT INTO @Invoice
OUTPUT INSERTED.* INTO @InsertedRows
VALUES ( GETDATE(),3000)
```

As for UPDATE statements, The second OUTPUT example above updates the first two rows in a table and outputs both the DELETED and INSERTED system tables:

```
UPDATE     TOP(2)  @tTest1
SET Amount = Amount * 10
OUTPUT DELETED.*,inserted.*
```

A note about OUTPUT: If you're using database triggers for audit trail logging, you may be tempted to think that the new OUTPUT functionality can replace audit trail logging. Strong piece of advice: resist the temptation. Database triggers provide the best mechanism for logging database changes because they will fire regardless of how an INSERT/UPDATE occurs. So restrict the use of OUTPUT to instances where you want immediate feedback on changes.

Tip 5: Isolation Levels (part 1 of 2)
Scenario: You've just started using SQL Server and want to understand how SQL Server locks rows during queries and updates. This is a two-part tip: the first will cover the four options that developers had prior to SQL Server 2000, and the next tip will cover an option that is specific to SQL Server 2005.

Some developers who come from different database environments (e.g. Visual FoxPro) ask how locking works in SQL Server 2000. Some developers who have started writing SQL queries later discover that implementing those queries leads to deadlocks.

The SQL Server paradigm uses the concept of *isolation*. In other words, when you query against data, when you perform transactions, how "isolated" (hence the name) are you from other processes that are working with the same data or logically related data? (Keep a mental note of the term *logically related data*—I'll come back to that later.)

SQL Server 2000 contains four isolation levels, each progressively more restrictive. They are as follows:

- READ UNCOMMITTED (dirty read)
- READ COMMITTED (the default level)
- REPEATABLE READ
- SERIALIZABLE READ

You can take advantage of isolation levels to avoid or minimize instances of deadlocks. SQL Server 2005 adds a fifth isolation level that I'll cover in the next tip, but it's important to understand the first four.

The code examples in this section, as well as Listing 9 and Listing 10, show examples of each isolation level. Each presents a "User A", "User B" scenario to show what happens when one user wants to update data while the other user queries it.

The first level, READ UNCOMMITTED, offers the least amount of isolation against other actions on the same row. If User A begins to update a record, and User B queries the record before User A commits the transaction, User B will still pick up the change. Why? Because READ UNCOMMITTED is just that—the ability to read uncommitted changes. This isolation level offers the best protection against deadlocks but also carries the greatest risk of querying bad data. If User A's update does not commit correctly (violates a constraint or a system error, etc.), then User B will be looking at not only "dirty" but also invalid data. The following code shows an example of this level:

```
-- USER A
USE AdventureWorks
GO
BEGIN TRAN
-- change from Trey Research to  'Trey & Wilson Research, Inc.'
UPDATE Purchasing.Vendor
SET Name = 'Trey & Wilson Research, Inc.'  WHERE VendorID = 10
-- USER B
USE AdventureWorks
GO
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
SELECT Name FROM Purchasing.Vendor WHERE VendorID = 10
-- will return 'Trey & Wilson Research, Inc'
--  even before USER A does a 'COMMIT TRANSACTION'
-- because we are doing a dirty read  (reading uncommitted)
```

Normally you would use this first isolation level for maximum query performance in situations where there is little or no risk of invalid uncommitted data.

The second level, READ COMMITTED is the default SQL Server isolation level. As the name implies, you can query only against committed data. So if another user is in the middle of a transaction that updates data, your query against the same record will be locked out until the original transaction completes. So in the example above, if User A updates the record and then User B tries to query before User A's transaction is completed, User B's query will be in deadlock until User A is done. At the end, User B's query will return the newly committed value. Here's an example of this level:

```
-- USER A
USE AdventureWorks
GO
BEGIN TRAN
-- change from Trey Research to  'Trey & Wilson Research, Inc.'
UPDATE Purchasing.Vendor
SET Name = 'Trey & Wilson Research, Inc.'  WHERE VendorID = 10
-- USER B
USE AdventureWorks
GO
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
SELECT Name FROM Purchasing.Vendor WHERE VendorID = 10
-- This query will remain in DEADLOCK, because USER A
-- has an exclusive lock on the ROW
-- The query will finish when USER A does a COMMIT or a ROLLBACK
```

The third level is REPEATABLE READ. In some stored procedures, a developer may need to query the same row or set of rows more than once. It's possible that another user may do something to update a row in between, which means that the original query might report a different value for the row. The process querying the data wants to ensure that the data remains the same between the two queries, even if someone tries to change the data during the process—in other words, the query wants to REPEAT the same read. So the user trying to update the record will be deadlocked out until the stored procedure finished. Listing 3 shows an example of this level.

The REPEATABLE READ level would seem to protect a long stored procedure that queries and re-queries data from any other processes that change the same data. However, there's one fly in the ointment. Remember earlier when I asked to make a mental note about "logically related data"? Suppose you have a long stored procedure that queries a set of data based on a condition, and then re-executes the same or similar query later in the procedure. Suppose someone INSERTS a new row in between, where the row meets the condition of the stored procedure?

This newly inserted record is known as a "phantom" record and you may not wish to see it until your stored procedure finishes. In this case the REPEATABLE READ doesn't protect you; the stored procedure would return a different result the second time.

However, the forth transaction level (SERIALIZABLE) protects this, by using "key-range locks" to lock out any insertions that would otherwise affect the conditions of queries performed inside a SERIALIZABLE transaction. SERIALIZABLE offers you the greatest level of isolation from other processes—l;and as you also can see-runs greater risk of other processes being locked out until you finish. Listing 4 shows an example of this level.

Granted, the enhancements with common table expressions and other query constructs will reduce the number of instances where a stored procedure will have to query the same data twice.

Many developers requested a finer level of granularity with isolation levels for SQL Server 2005, which I'll cover in the next tip.

Tip 6: Isolation Levels (part 2 of 2)
Scenario: You've reviewed the four isolation levels available in Tip #5 You're looking for the ability to query against committed data, but you don't want the query to implement any shared locks against the data.

In some situations, you may want to execute a series of queries in a stored procedure and be able to guarantee:

- That you're querying against committed data only.
- That you'll return the same results each time.
- That another user can update the data, or insert related data, while your stored procedure is querying the same data, without the user experiencing a deadlock.
- That even when the user commits the update or insert transaction, your query will still continue to use the same version of the data that existed when your stored procedure queries begin.

So, no deadlocks, and everyone is reasonably happy. Does it sound too good to be true? This scenario describes the SNAPSHOT isolation level in SQL Server 2005.

The key word in the scenario above is "version" because SNAPSHOT isolation works by internally creating a snapshot version of data into the system tempdb database. Listing 5 shows an example of the new SNAPSHOT isolation level. Note that you must configure the database to support this level:

```
-- First, you must configure the database to
-- support SNAPSHOT isolation
ALTER DATABASE AdventureWorks
SET ALLOW_SNAPSHOT_ISOLATION ON
GO
```

Of course, any option with this much power can come at a cost. You should use this isolation level (or any isolation level) only judiciously—abusing the SNAPSHOT level can become very resource intensive with the tempdb database.

Tip 7: XML and XQUERY
Scenario: You want to utilize XML to query against a variable number of key selections that you pass as parameters.

A common challenge I've faced in querying is supporting multiple/variable selections. A user might want to run a report for one account, or ten, or a hundred, or more. Since this might occur in several different places in an application, I've always searched for a reusable methodology.

Prior to SQL Server 2000, you generally had two choices:

- Pass a CSV list of select keys and convert the CSV list to a table variable/temp table.
- Pass an XML string and use the system stored procedure sp_xml_preparedocument and the OPENXML function.

Either of these two options still works in SQL 2005, and the CSV approach benefits from the fact that a VARCHAR parameter can now be much larger (i.e., VARCHAR(MAX)). However, the new XML capabilities in SQL 2005 allow you to pass an XML string of selections, without the overhead of sp_xml_preparedocument and OPENXML. You can do this by using the new XQUERY capabilities in SQL Server 2005 to "shred" XML strings to read individual nodes.

The XML features in SQL Server 2005 are especially helpful in processing variable-length lists of keys

Here's a table-valued UDF called XML2TableVar that you can use to convert any XML string of integer key values to a table variable:

```
ALTER FUNCTION [dbo].[XML2TableVar]
(  @XMLString XML)
RETURNS
   @tPKList TABLE   ( PKValue int )
AS
BEGIN
INSERT INTO @tPKList
SELECT Tbl.Col.value('.','int')
FROM   @XMLString.nodes('//PKValue') Tbl(Col)
RETURN
```

```
END
```

Listing 6 demonstrates the use of this UDF. You simply need to define an XML string with a column called PKValue:

```
DECLARE @cVendorXML XML
SET @cVendorXML ='<Vendors>
<Vendor>
<PKValue>73</PKValue>
<PKValue>76</PKValue>
</Vendor>
</Vendors>'
```

Then you can query a table and JOIN the results of XML2TableVar:

```
SELECT Orders.* FROM
 Purchasing.PurchaseOrderHeader Orders
    JOIN dbo.XML2TableVar(@cVendorXML) VendorList
ON VendorList.PKValue = Orders.VendorID
```

One additional point: The first query in Listing 6 basically requires that the XML string must contain at least one entry. Suppose the user selects "all vendors" in the application, and the system contains thousands of vendors. Does the application need to deal with an XML string that contains a row for each Vendor PK?

Well, it shouldn't have to. You can modify the query in Listing 6 (the code appears at the bottom of Listing 6) to use an OUTER APPLY instead of a JOIN, and add a CASE statement to examine if the XML string is empty before deciding to use the WHERE clause on the VendorID:

```
SELECT Orders.* FROM
   Purchasing.PurchaseOrderHeader Orders
-- Must use OUTER APPLY and a CASE statement
   OUTER APPLY dbo.XML2TableVar(@cVendorXML)
       AS VendorList
WHERE
CASE WHEN LEN(
    CAST(@cVendorXML AS VARCHAR(MAX)))
    = 0 THEN 1
WHEN VendorList.PKValue = Orders.VendorID
THEN 1
ELSE 0 END = 1
```

Tip 8: Variable *TOP N*

Scenario: You want to query against the first N number of rows based on some condition, and where N is variable. While you could do this by constructing a dynamic SQL statement, you're looking for an easier way. Additionally, you're also looking for a way to INSERT *or* UPDATE *a variable number of rows, based on* TOP N *syntax.*

Prior to SQL Server 2005, if you wanted to retrieve the first N number of records based on some order, and the N was variable, you either had to use dynamic SQL to construct the query or use dynamic SQL for the ROWCOUNT statement. This was because SQL 2000 treated TOP N as a literal, not a variable.

SQL Server 2005 now treats the N as a true variable. Listing 7 shows several examples of the new TOP N capabilities, including the ability to set a variable for TOP N in an INSERT/UPDATE statement.

Tip 9: *APPLY* and Table-Valued UDFs

Scenario: You previously realized that you and your development team were repeating logic for result sets, and decided to abstract that logic to create table-valued user-defined functions (UDFs). However, you're finding that integrating your table-valued UDFs from your queries sometimes means extra steps and extra queries. You want to call your UDFs cleanly from your queries.

Just as a picture is worth a thousand words, code samples are sometimes worth a thousand words as well. SQL Server 2005's APPLY capability allows developers to apply (once again, *hence the name*) reusable table-valued UDFs with queries.

If you ever tried to incorporate table-valued UDFs into stored procedures in SQL 2000, you know that you sometimes had to create additional temporary tables or additional SQL statements. Ideally, you'd like to integrate a table-valued UDF into a query, and pass a column from the main query table as a parameter into the UDF. Here's one scenario that creates table-valued UDFs and then uses a query to APPLY the UDFs:

```
-- Create a Table-valued UDF
CREATE FUNCTION [dbo].[GetTopNPurchaseOrders]
   (@EmployeeID AS int, @nTop AS INT)
RETURNS TABLE
AS
RETURN
SELECT TOP(@nTop) PurchaseOrderID, EmployeeID,
              OrderDate,  TotalDue
FROM Purchasing.PurchaseOrderHeader
WHERE EmployeeID = @EmployeeID
ORDER BY TotalDue DESC
GO

-- APPLY the table-valued UDF
USE AdventureWorks
GO
DECLARE @nTopCount int
SET @nTopCount = 5
SELECT TOPOrder.EmployeeID , TOPOrder.PurchaseOrderID,
   TOPOrder.OrderDate, TOPOrder.TotalDue
FROM HumanResources.Employee Employee
   CROSS APPLY dbo.GetTopNPurchaseOrders
   (Employee.EmployeeID ,@nTopCount) AS TOPOrder
ORDER BY TOPOrder.EmployeeID, TOPOrder.TotalDue DESC
```

Here's a second scenario:

```
-- Another table-valued UDF
CREATE FUNCTION [dbo].[GetPurchaseOrdersGTx]
   (@EmployeeID AS int, @nThreshold AS INT)
RETURNS @tOrders TABLE (PurchaseOrderID int,
   EmployeeID int, OrderDate datetime,
   TotalDue decimal(14,2))
AS
BEGIN
INSERT INTO @tOrders
SELECT PurchaseOrderID, EmployeeID,
    OrderDate,  TotalDue
FROM Purchasing.PurchaseOrderHeader
WHERE EmployeeID = @EmployeeID AND
    TotalDue  >= @nThreshold
ORDER BY TotalDue DESC
RETURN
END
GO


--APPLY the UDF from a sub-query
DECLARE @nNumOrders int, @nMinAmt decimal(14,2)
SET @nNumOrders = 10
SET @nMinAmt = 75000.00
-- Find Employees with 2 orders at least 5K
SELECT EmployeeID  FROM HumanResources.Employee
WHERE  (SELECT COUNT(*) FROM
     dbo.GetPurchaseOrdersGTx (
     EmployeeID, @nMinAmt)) >=@nNumOrders
GO
-- Now, for these 10 employees with
-- orders greater than 75,000,
-- retrieve those orders
DECLARE @nNumOrders int, @nMinAmt decimal(14,2)
SET @nNumOrders = 10
SET @nMinAmt = 75000.00
-- use a CTE to get the list of employees
;WITH EmpList AS (SELECT EmployeeID
    FROM HumanResources.Employee
WHERE  (SELECT COUNT(*) FROM
     dbo.GetPurchaseOrdersGTx (
      EmployeeID,@nMinAmt)) >=@nNumOrders)
-- then query against that list
SELECT MaxOrders.* FROM EmpList
   CROSS APPLY dbo.GetPurchaseOrdersGTx
       (EmployeeID ,@nMinAmt)
AS MaxOrders
```

Note that both the scenarios above use the CROSS APPLY statement to apply the table-valued UDF across every row in the main table.

Tip 10: *RANKING* and *PARTITIONING*
Scenario: You want to assign ranking numbers to your result set so that you can physically show the numbers 1 through 10 to annotate the top ten orders for each vendor. You also want to see other applications for partitioning.

Prior to SQL Server 2005, if you ever wanted to assign ranking numbers to a result set, you either had to create an identity column in a result set or manually loop through the results and assign a ranking number. If you needed to assign ranking numbers within a group (e.g. assign ranking number to orders by vendor), you know that the processes became even more involved.

Fortunately, SQL Server 2005 provides functionality to easily create a ranking column in a result set. The code below shows two examples of ranking: one for the entire result set, and a second within a group. You can use the new ROW_NUMBER() function to assign a ranking number, and the OVER (ORDER BY) clause to indicate how SQL Server should order the ranking logic:

```
DECLARE @nTop INT
SET @nTop = 50
SELECT  TOP(@nTop)
   EmployeeID, PurchaseOrderID, OrderDate, TotalDue,
   ROW_NUMBER() OVER ( ORDER BY  TotalDue DESC  ) AS OrderRank
FROM Purchasing.PurchaseOrderHeader

SELECT EmployeeID, PurchaseOrderID, OrderDate, TotalDue,
       ROW_NUMBER() OVER
      (PARTITION BY EmployeeID ORDER BY TotalDue DESC) AS OrderRank
FROM Purchasing.PurchaseOrderHeader
```

Note how the second example uses the PARTITION statement to assign ranking numbers by a certain group (EmployeeID, and then TotalDue descending within EmployeeID), you can use the PARTITION statement:

The PARTITION statement has other uses beyond ranking. While this would be an unusual case, suppose you wanted to execute an in-line summary of a column, or maybe to SUM and GROUP a specific column without dealing with the GROUP rule that all non-aggregate columns must be specified in a GROUP BY. Here's an example that shows how to create aggregate partitions:

```
--Aggregate partitions as an alternative to GROUP BY
SELECT PurchaseOrderID, PurchaseOrderDetailID,ProductID,
     SUM(LineTotal) OVER (PARTITION BY PurchaseOrderID) AS POTotal,
    COUNT(*) OVER (PARTITION BY PurchaseOrderID) AS POItemCount,
    SUM(LineTotal) OVER (PARTITION BY ProductID) AS ProductTotal,
    COUNT(*) OVER (PARTITION BY ProductID) AS ProductItemCount
FROM Purchasing.PurchaseOrderDetail
ORDER BY PurchaseOrderID
```

Finally, you can use the ROW_NUMBER function to implement custom paging of result sets in a web application. If a user query results in 500 rows, and you want to display only 20 at a time and implement custom paging logic, you can use the ranking functions to assign result set numbers and return offset ranges (e.g. rows 21-40, 41-60). I presented a full solution for this in the March-April 2007 issue of *CoDe Magazine*.

Tip 11: *TRY…CATCH*
Scenario: You want to implement TRY…CATCH *capabilities in SQL Server 2005 stored procedures in the*

same way you can implement TRY…CATCH *capabilities in .NET. You also want to be able to* RAISE *an error back to the calling application.*

Both database developers and DBAs will be happy to know that the TRY…CATCH language features in C# and Visual Basic are now available in T-SQL 2005. You can place nested TRY…CATCH statements inside stored procedures (though you cannot use FINALLY).

Listing 8 shows an example of a TRY…CATCH block of code that attempts to delete a row from the PO header table. The code will generate an error because of the constraint between the header and detail files. You want to catch the error and raise an exception back to the calling application so that any TRY…CATCH blocks in the application layer can detect and process the exception.

The CATCH block in Listing 8 reads the error severity, number, message, and error state information from SQL Server. The block of code then uses the RAISERROR function to raise an exception back to the calling application. But in between, the code examines the Error State setting to see if the value is zero. The reason is because some errors (e.g. constraint violations) do not return an error state, and so you must set the error state to a value of one, before raising the error back to the application.

Tip 12: *INTERSECT/EXCEPT*
Scenario: You're looking for a replacement for IN *and* NOT IN *clauses that are simpler to code and better at handling* NULL *values.*

INTERSECT and EXCEPT statements in SQL Server 2005 are lesser-known (but still valuable) language features in SQL Server 2005, as they provide cleaner alternatives for IN and NOT IN. Here's an example of using INTERSECT and EXCEPT:

```
--Using INTERSECT/EXCEPT instead of IN/NOT IN
USE AdventureWorks
GO
SELECT Vendor.VendorID  FROM Purchasing.Vendor Vendor
INTERSECT SELECT VendorID FROM Purchasing.PurchaseOrderHeader
SELECT Vendor.VendorID  FROM Purchasing.Vendor Vendor
EXCEPT SELECT VendorID FROM Purchasing.PurchaseOrderHeader
```

Along with slightly improved readability, INTERSECT and EXCEPT also do not require ISNULL or COALESCE statements to test for NULL values in the sub-query, whereas IN and NOT IN require them.

Tip 13: Flexible *UPDATE* Procedures
Scenario: You want to build UPDATE *stored procedures so that you only have to pass parameters for column values that changed, and you also want to pass explicit* NULL *values for columns that should be* NULL*ed out.*

Unlike all the other code in this article, this tip can apply to SQL Server 2000 as well as SQL Server 2005.

Recently I participated in a debate regarding stored procedures to handle UPDATE statements. Essentially, someone posed the question about whether an UPDATE stored proc could handle the following:

- Optional parameters (for example, a table has twenty columns, but a user only changes three of them, so the application should only have to pass those three parameters, not all twenty).
- Explicit NULL parameters (the ability to set a non-null column value to NULL)

Some people have asked me about the second scenario. You might have a survey application where NULL means "No Response." A user might accidently post a response, click Save, and then realize later that the response should have been "No Response." Another example might be an insurance application that ties individual accounting transactions to a particular policy. A transaction might be shifted from policy A to another policy, but not right away. So the related policy foreign key might need to be set to NULL for a period of time, until the client determines the correct foreign key. Bottom line: sometimes it's necessary to set values from non-null to NULL.

Addressing both conditions proves to be an interesting challenge. A single column parameter might be NULL because the application did not pass it, in which case you don't want the column to change from the current value. On the other hand, a single column parameter might be NULL because the application explicitly passed a NULL value. So how can the stored procedure code determine the "intent" of the NULL?

Some database developers handle this by adding a second set of Boolean parameters to indicate that the NULL parameter is an explicit NULL. While this works, it adds to the stored procedure's API by requiring more parameters. An alternate solution shown below utilizes a default token value for each datatype that would otherwise never be used. You can then check in the UPDATE statement if the parameter is still equal to this token value:

```
--UPDATE stored proc to handle implicit and explicit NULL values
CREATE PROCEDURE [dbo].[UpdateData]
  @PK int,  @FullName char(100)='-999999',
  @LastEvalDate DateTime = '01-01-1800',
  @Salary Decimal(14,0)  = -999999, @Interviewed int = -99
AS
BEGIN
SET NOCOUNT ON;
UPDATE NameTest
SET FullName = CASE
WHEN @Name = '-999999'  THEN Name
ELSE  @Name END, LastEvalDate = CASE

WHEN @LastEvalDate = '01-01-1800' THEN LastEvaldate
ELSE @LastEvalDate END, Salary = CASE

WHEN @Salary= -999999  THEN Salary
ELSE @Salary END, Interviewed = CASE

WHEN @Interviewed = -99  THEN Interviewed
ELSE CAST(@Interviewed AS BIT) END

OUTPUT Inserted.*
WHERE PK = @PK
END
```

Note that the Boolean column for Interviewed is represented as an integer column. This is because it's not possible to set a token value for a bit column, as the only possible values are NULL, 0, or 1. However, you can define the parameter as an integer and then CAST it as a bit if the value does not equal the token value. A developer can still pass a true/false value from the application.

Sneak Preview of Katmai (SQL Server 2008)
As I write, Microsoft has released the June 2007 Community Technology Preview (CTP) for SQL Server 2008 (codenamed "Katmai"). Microsoft provides two links you can follow for more information on Katmai:

- http://www.microsoft.com/sql/prodinfo/futureversion/default.mspx
- https://connect.microsoft.com/SQLServer/content/content.aspx?ContentID=5395

Katmai features a number of language enhancements to the T-SQL language. Here are a few:

- Programming shortcuts for declaring and assigning values to variables
- Table types and table parameters
- Expansion of GROUP BY clause
- A new MERGE statement to perform an INSERT/UPDATE/DELETE in one statement
- Row constructors for INSERT statements

Closing Thoughts:
Have you ever submitted something (an article, a paper, some code, etc.) and thought of some good ideas after the fact? Well, I'm the king of thinking of things afterwards. Fortunately, that's the type of thing that makes blogs valuable. Check my blog for follow-up tips and notes on Baker's Dozen articles…and maybe a few additional treats. You can find the entire source code on my web site.

Kevin S. Goff is the founder and principal consultant of Common Ground Solutions, a consulting group that provides custom Web and desktop software solutions in .NET, Visual FoxPro, SQL Server, and Crystal Reports. Kevin has been building software applications for 17 years. He has received several awards from

the U.S. Department of Agriculture for systems automation. He has also received special citations from Fortune 500 companies for solutions that yielded six-figure returns on investment. He has worked in such industries as insurance, accounting, public health, real estate, publishing, advertising, manufacturing, finance, consumer packaged goods, and trade promotion. In addition, Kevin provides many forms of custom training.