Microsoft TechNet

Introduction to New T-SQL Programmability Features in SQL Server 2008

SQL Server Technical Article

Writer: Itzik Ben-Gan, SolidQ.com [http://www.solidq.com/default.aspx] /InsideTSQL.com [http://www.insidetsql.com/default.aspx]

Technical Reviewers: Michael Wang, Microsoft Corp.

Umachandar Jayachandran (UC), Microsoft Corp.

Jim Hogg, Microsoft Corp.

Michael Rys, Microsoft Corp.

Published: July 2008

Applies to: SQL Server 2008

Summary: This paper introduces key new Transact-SQL programmability features in Microsoft SQL Server 2008 as well as SQL/Common Language Runtime (CLR) enhancements. New Transact-SQL features provide improved performance, increased functionality, and enhanced globalization support. Transact-SQL programmability enhancements in SQL Server 2008 address the needs of both OLTP and data warehouse environments.

Acknowledgments: I would like to thank the following members of Solid Quality Mentors who provided feedback about this white paper: Kathy Blomstrom, Ken Spencer, Greg Low, Allan Hirt, Aaron Johal, James Winters, Gustavo Larriera, Andrew J. Kelly, Gianluca Hotz, Davide Mauri, Francisco A González Díaz, Adam Machanic, Jesús López, Randy Dyess, and Eladio Rincón.

Introduction

Microsoft SQL Server 2008 introduces several important new Transact-SQL programmability features and enhances some existing ones. This paper covers the new features and enhancements and demonstrates them through code samples. It will familiarize you with the new features by providing a high-level functionality overview but does not cover all technical details. You can find details in SQL Server Books Online.

This paper covers the following key new features

- Declaring and initializing variables
- Compound assignment operators
- Table value constructor support through the VALUES clause
- Enhancements to the CONVERT function
- New date and time data types and functions
- Large UDTs
- The HIERARCHYID data type
- Table types and table-valued parameters
- The MERGE statement, grouping sets enhancements

- DDL trigger enhancements
- Sparse columns
- Filtered indexes
- Large CLR user-defined aggregates
- Multi-input CLR user-defined aggregates
- The ORDER option for CLR table-valued functions
- Object dependencies
- Change data capture
- Collation alignment with Microsoft® Windows®
- Deprecation

Transact-SQL Delighters

Transact-SQL delighters are small enhancements that, for the most part, help make the programming experience more convenient. This section covers the following Transact-SQL delighters: declare and initialize variables, compound assignment operators, table value constructor support through the VALUES clause, and the enhanced CONVERT function.

Declare and Initialize Variables

Microsoft SQL Server® 2008 enables you to initialize variables inline as part of the variable declaration statement instead of using separate DECLARE and SET statements. This enhancement helps you abbreviate your code. The following code example demonstrates inline initializations using a literal and a function:

```
DECLARE @i AS INT = 0, @d AS DATETIME = CURRENT_TIMESTAMP;

SELECT @i AS [@i], @d AS [@d];
```

Compound Assignment Operators

Compound assignment operators help abbreviate code that assigns a value to a column or a variable. The new operators are:

- += (plus equals)
- -= (minus equals)
- *= (multiplication equals)
- /= (division equals)
- %= (modulo equals)

You can use these operators wherever assignment is normally allowed—for example, in the SET clause of an UPDATE statement or in a SET statement that assigns values to variables. The following code example demonstrates the use of the += operator:

```
DECLARE @price AS MONEY = 10.00;

SET @price += 2.00;

SELECT @price;
```

This code sets the variable @price to its current value, 10.00, plus 2.00, resulting in 12.00.

Table Value Constructor Support through the VALUES Clause

SQL Server 2008 introduces support for table value constructors through the VALUES clause. You can now use a single VALUES clause to construct a set of rows. One use of this feature is to insert multiple rows based on values in a single INSERT statement, as follows:

```
USE tempdb;

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

CREATE TABLE dbo.Customers

(

  custid      INT         NOT NULL,

  companyname VARCHAR(25) NOT NULL,

  phone       VARCHAR(20) NOT NULL,

  address     VARCHAR(50) NOT NULL,

  CONSTRAINT PK_Customers PRIMARY KEY(custid)

);

INSERT INTO dbo.Customers(custid, companyname, phone, address)

  VALUES

  (1, 'cust 1', '(111) 111-1111', 'address 1'),

  (2, 'cust 2', '(222) 222-2222', 'address 2'),

  (3, 'cust 3', '(333) 333-3333', 'address 3'),

  (4, 'cust 4', '(444) 444-4444', 'address 4'),

  (5, 'cust 5', '(555) 555-5555', 'address 5');
```

Note that even though no explicit transaction is defined here, this INSERT statement is considered an atomic operation. So if any row fails to enter the table, the entire INSERT operation fails.

A table value constructor can be used to define table expressions such as key derived tables and CTEs, and can be used where table expressions are allowed (such as in the FROM clause of a SELECT statement or as the source table in a MERGE statement). The following example demonstrates using the VALUES clause to define a derived table in the context of an outer SELECT statement:

```
SELECT *

FROM

  (VALUES

    (1, 'cust 1', '(111) 111-1111', 'address 1'),

    (2, 'cust 2', '(222) 222-2222', 'address 2'),

    (3, 'cust 3', '(333) 333-3333', 'address 3'),

    (4, 'cust 4', '(444) 444-4444', 'address 4'),

    (5, 'cust 5', '(555) 555-5555', 'address 5')

  ) AS C(custid, companyname, phone, address);
```

The outer query can operate on this table expression like any other table expression, including joins, filtering, grouping, and so on.

Enhanced CONVERT Function

The CONVERT function is enhanced in SQL Server 2008 to support new conversion options between character strings and binary data types. You determine the conversion option to use by specifying a style number as the third argument. Style 0 is the default behavior that was supported in previous SQL Server versions—this style translates character strings to the binary representation of the ASCII codes for the characters and vice versa.

Styles 1 and 2 introduce new functionality. These styles convert a hex string literal to a binary value that contains the same digits when presented in hex form and vice versa. If you use Style 1 when converting from a character string to a binary value, the input character string should include the 0x prefix; if you use Style 2, it should not. Similarly, if you use Style 1 when converting from a binary value to a character string, the output character string will contain the 0x prefix; if you use Style 2, it will not. The following example demonstrates using styles 1 and 2:

```
SELECT
  CONVERT(VARCHAR(12) , 0x49747A696B  , 1) AS [Bin to Char 1],
  CONVERT(VARBINARY(5), '0x49747A696B', 1) AS [Char to Bin 1],
  CONVERT(VARCHAR(12) , 0x49747A696B  , 2) AS [Bin to Char 2],
  CONVERT(VARBINARY(5), '49747A696B'  , 2) AS [Char to Bin 2];
```

This code produces the following output:

```
Bin to Char 1 Char to Bin 1 Bin to Char 2 Char to Bin 2
------------- ------------- ------------- -------------
0x49747A696B  0x49747A696B  49747A696B    0x49747A696B
```

Date and Time Data Types

Before SQL Server 2008, date and time improvements were probably at the top of the list of the most requested improvements for SQL Server—especially the request for separate date and time data types, but also for general enhanced support for temporal data. SQL Server 2008 introduces four new date and time data types—including DATE, TIME, DATETIME2, and DATETIMEOFFSET—as well as new functions that operate on the new types and enhancements to existing functions.

New Data Types

The four new date and time data types provide a split between date and time, support for a larger date range, improved accuracy, and support for a time zone element. The new data types have the compliant semantics and are compatible with the SQL standard. The DATE and TIME data types split the date and time, which in previous versions were consolidated. The DATETIME2 data type is an improved version of DATETIME, providing support for a larger date range and better accuracy. The DATETIMEOFFSET data type is similar to DATETIME2 with the addition of a time zone component. Table 1 describes the new data types, showing their storage in bytes, date-range support, accuracy, recommended entry format for literals, and an example.

Data Type	Storage (bytes)	Date Range	Accuracy	Recommended Entry Format and Example
DATE	3	January 1, 0001, through December 31, 9999 (Gregorian calendar)	1 day	'YYYY-MM-DD' '2009-02-12'
TIME	3 to 5		100 nanoseconds	'hh:mm:ss.nnnnnnn' '12:30:15.1234567'
DATETIME2	6 to 8	January 1, 0001, through December 31, 9999	100 nanoseconds	'YYYY-MM-DD hh:mm:ss.nnnnnnn' '2009-02-12 12:30:15.1234567'
DATETIMEOFFSET	8 to 10	January 1, 0001, through December 31, 9999	100 nanoseconds	'YYYY-MM-DD hh:mm:ss.nnnnnnn [+|-]hh:mm'

'2009-02-12
12:30:15.1234567 +02:00'

Table 1: New Date and Time Data Types

Note that the format 'YYYY-MM-DD' is language neutral for the new data types, but it is language dependent for the DATETIME and SMALLDATETIME data types. The language-neutral format for those data types is 'YYYYMMDD'.

The three new types that contain a time component (TIME, DATETIME2, and DATETIMEOFFSET) enable you to specify the fractional seconds precision in parentheses following the type name. The default is 7, meaning 100 nanoseconds. If you need a fractional second accuracy of milliseconds, such as three for example, you must explicitly specify it: DATETIME2(3).

Metadata information in SQL Server 2008 reports meaningful precision and scale values for the new date and time data types. In the **sys.columns** view, the **precision** attribute describes the total number of characters in the default literal string representation of the value, and the scale describes the number of digits in the fractional part of the seconds. In INFORMATION_SCHEMA.COLUMNS, the DATETIME_PRECISION attribute describes the number of digits in the fractional part of the seconds.

The following code shows an example of using the new types:

```
DECLARE
  @d   AS DATE           = '2009-02-12',
  @t   AS TIME           = '12:30:15.1234567',
  @dt2 AS DATETIME2      = '2009-02-12 12:30:15.1234567',
  @dto AS DATETIMEOFFSET = '2009-02-12 12:30:15.1234567 +02:00';

SELECT @d AS [@d], @t AS [@t], @dt2 AS [@dt2], @dto AS [@dto];
```

The new data types are fully supported by the SQL Server Native Client OLE DB and ODBC providers as well as by ADO.NET in Microsoft Visual Studio® 2008. Table 2 shows the mappings between the new SQL Server 2008 data types and the corresponding client provider types.

SQL	ODBC	OLE DB	ADO.NET
DATE	SQL_TYPE_DATE/ SQL_DATE	DBTYPE_DBDATE	DateTime
TIME	SQL_TIME/ SQL_SS_TIME2	DBTYPE_DBTIME/ DBTYPE_DBTIME2	TimeSpan
DATETIMEOFFSET	SQL_SS_TIMESTAMPOFFSET	DBTYPE_DBTIMESTAMPOFFSET	DateTimeOffset
DATETIME2	SQL_TYPE_TIMESTAMP SQL_TIMESTAMP	DBTYPE_DBTIMESTAMP	DateTime

Table 2: Date and Time Data Type Mappings to Client Providers

New and Enhanced Functions

To support the new date and time data types, SQL Server 2008 introduces new functions and enhances existing functions. The new functions are SYSDATETIME, SYSUTCDATETIME, SYSDATETIMEOFFSET, SWITCHOFFSET, and TODATETIMEOFFSET.

SYSDATETIME, SYSUTCDATETIME, and SYSDATETIMEOFFSET return the current system date and time value. SYSDATETIME returns the current date and time as a DATETIME2 value, SYSUTCDATETIME returns the current date and time in UTC as a DATETIME2 value, and SYSDATETIMEOFFSET returns the current date and time along with the system time zone as a DATETIMEOFFSET value. To get only the current date or only the current time, you can cast the value returned from the SYSUTCDATETIME function to DATE or TIME, as the following example shows:

```
SELECT

  CAST(SYSDATETIME() AS DATE) AS [current_date],

  CAST(SYSDATETIME() AS TIME) AS [current_time];
```

Interestingly, when converting a column whose data type contains both a date and a time component to DATE, the SQL Server query optimizer can still rely on index ordering to process a query more efficiently. This is contrary to the usual behavior, where conversion of a column to a different type prevents the optimizer from relying on index order. The new behavior means that the optimizer might consider an index seek for a query filter that has a conversion to DATE. For example, the plan for the following query performs an index seek on the index on the CurrencyRateDate DATETIME column:

```
USE AdventureWorks;

SELECT FromCurrencyCode, ToCurrencyCode, EndOfDayRate

FROM Sales.CurrencyRate

WHERE CAST(CurrencyRateDate AS DATE) = '20040701';
```

The SWITCHOFFSET function adjusts an input DATETIMEOFFSET value to a specified time zone, while preserving the UTC value. The syntax is SWITCHOFFSET(*datetimeoffset_value*, *time_zone*). For example, the following code adjusts the current system datetimeoffset value to time zone GMT +05:00:

```
SELECT SWITCHOFFSET(SYSDATETIMEOFFSET(), '-05:00');
```

So if the current system datetimeoffset value is February 12, 2009 10:00:00.0000000 -08:00, this code returns the value February 12, 2009 13:00:00.0000000 -05:00.

The TODATETIMEOFFSET function sets the time zone offset of an input date and time value. Its syntax is TODATETIMEOFFSET(*date_and_time_value*, *time_zone*).

This function is different from SWITCHOFFSET in several ways. First, it is not restricted to a datetimeoffset value as input; rather it accepts any date and time data type. Second, it does not try to adjust the time based on the time zone difference between the source value and the specified time zone but instead simply returns the input date and time value with the specified time zone as a datetimeoffset value.

The main purpose of the TODATETIMEOFFSET function is to convert types that are not time zone aware to DATETIMEOFFSET by the given time zone offset. If the given date and time value is a DATETIMEOFFSET, the TODATETIMEOFFSET function changes the DATETIMEOFFSET value based on the same original local date and time value plus the new given time zone offset.

For example, the current system datetimeoffset value is February 12, 2009 10:00:00.0000000 -08:00, and you run the following code:

```
SELECT TODATETIMEOFFSET(SYSDATETIMEOFFSET(), '-05:00');
```

The value February 12, 2009 10:00:00.0000000 -05:00 is returned. Remember that the SWITCHOFFSET function returned February 12, 2009 13:00:00.0000000 -05:00 because it adjusted the time based on the time zone differences between the input (-08:00) and the specified time zone (-05:00).

As mentioned earlier, you can use the TODATETIMEOFFSET function with any date and time data type as input. For

example, the following code takes the current system date and time value and returns it as a datetimeoffset value with a time zone -00:05:

```
SELECT TODATETIMEOFFSET(SYSDATETIME(), '-05:00');
```

Functions that accept a date and time part as input now support new parts. The DATEADD, DATEDIFF, DATENAME, and DATEPART functions add support for microseconds and nanoseconds. DATENAME and DATEPART also support the TZoffset part (time zone), with DATEPART also supporting the ISO_WEEK part (ISO 8601 week number).

Many other functions are enhanced in SQL Server 2008 to support the new date and time types, among them the existing date and time functions, type conversion functions (CAST and CONVERT), set and aggregate functions (such as MAX, MIN), metadata functions (such as TYPEPROPERTY, COLUMNPROPERTY), and system functions (such as DATALENGTH, IS_DATE).

Large UDTs

In SQL Server 2005, user-defined types (UDTs) in the CLR were limited to 8,000 bytes. SQL Server 2008 lifts this limitation and now supports large UDTs. Similar to the built-in large object types that SQL Server supports, large UDTs can now reach up to 2 GB in size. If the UDT value does not exceed 8,000 bytes, the database system treats it as an inline value as in SQL Server 2005. If it exceeds 8,000 bytes, the system treats it as a large object and report its size as "unlimited."

One application where large UDTs are required is to support spatial data. For the most part, 8,000 bytes might be enough to represent a value of a spatial data type, but in some cases, the size requirement of such values might exceed 8,000 bytes. SQL Server 2008 introduces the built-in UDTs GEOMETRY and GEOGRAPHY, which are implemented as large CLR UDTs. These new types introduce support for spatial data. The GEOGRAPHY type supports round earth topology, and the GEOMETRY type supports flat earth topology. Spatial data is covered in detail in a separate paper.

A UDT's size is defined by the attribute **SqlUserDefinedTypeAttribute.MaxByteSize** as part of the type's definition. If this attribute is set to -1, the serialized UDT can reach the same size as other large object types (currently 2 GB); otherwise, the UDT cannot exceed the size specified in the **MaxByteSized** property.

Client APIs such as OLE DB and ODBC in the SQL Server Native Client and ADO.NET were enhanced to support large UDTs. Down-level clients (SQL Server 2005 and 2000 clients) convert a large UDT to VARBINARY(MAX) and IMAGE, respectively.

HIERARCHYID Data Type

The new HIERARCHYID data type in SQL Server 2008 is a system-supplied CLR UDT that can be useful for storing and manipulating hierarchies. This type is internally stored as a VARBINARY value that represents the position of the current node in the hierarchy (both in terms of parent-child position and position among siblings). You can perform manipulations on the type by using either Transact-SQL or client APIs to invoke methods exposed by the type. Let's look at indexing strategies for the HIERARCHYID type, how to use the type to insert new nodes into a hierarchy, and how to query hierarchies.

Indexing Strategies

The following code demonstrates how to use the HIERARCHYID type, creating a table named Employees (in the **tempdb** database for test purposes) with a column called hid that holds the HIERARCHYID value:

```
USE tempdb;

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

CREATE TABLE dbo.Employees

(

  empid   INT NOT NULL,

  hid     HIERARCHYID NOT NULL,

  lvl AS hid.GetLevel() PERSISTED,

  empname VARCHAR(25) NOT NULL,

  salary  MONEY       NOT NULL,
```

```
  CONSTRAINT PK_Emploees PRIMARY KEY NONCLUSTERED(empid)
);
```

Notice how the code uses the **GetLevel** method of the hid column to define the persisted computed column level. This method returns the level of the current node in the hierarchy.

The HIERARCHYID value provides topological sorting, meaning that a child's sort value is guaranteed to be higher than the parent's sort value. This definition is transitive; that is, a node's sort value is guaranteed to be higher than all of its ancestors. So if you create an index on the HIERARCHYID column, the index sorts the data in a depth-first manner—all members of the same subtree are close to each other in the leaf of the index. Such an index can efficiently satisfy requests such as returning all descendents of a node.

You can use another indexing strategy called *breadth-first*, in which you organize all nodes from the same level close to each other in the leaf of the index by having the column representing the level in the hierarchy first in the key list. Requests that can benefit from such an index include getting all nodes from the same level, getting direct subordinates of a node, and so on. The following code creates both types of indexes on the Employees table:

```
CREATE UNIQUE CLUSTERED INDEX idx_depth_first ON dbo.Employees(hid);

CREATE UNIQUE INDEX idx_breadth_first ON dbo.Employees(lvl, hid);
```

Inserting New Nodes

To insert a node into the hierarchy, you must first produce a new HIERARCHYID value that represents the correct position in the hierarchy. Use the **HIERARCHYID::GetRoot()** method to produce the value for the root node. You use the **GetDescendant** method to produce a value below a given parent. The **GetDescendant** method accepts two optional HIERARCHYID input values representing the two nodes between which you want to position the new node.

Note that the **GetDescendant** method does not guarantee that HIERARCHYID values are unique. To enforce uniqueness, you must define a primary key, a unique constraint, or a unique index on the column.

For example, the following code creates the usp_AddEmp stored procedure, which adds a new node to the hierarchy:

```
IF OBJECT_ID('dbo.usp_AddEmp', 'P') IS NOT NULL DROP PROC dbo.usp_AddEmp;
GO
CREATE PROC dbo.usp_AddEmp
  @empid   AS INT,
  @mgrid   AS INT = NULL,
  @empname AS VARCHAR(25),
  @salary  AS MONEY
AS

DECLARE
  @hid           AS HIERARCHYID,
  @mgr_hid       AS HIERARCHYID,
  @last_child_hid AS HIERARCHYID;

IF @mgrid IS NULL
  SET @hid = HIERARCHYID::GetRoot();
```

```
ELSE
BEGIN
  SET @mgr_hid = (SELECT hid FROM dbo.Employees WHERE empid = @mgrid);
  SET @last_child_hid =
    (SELECT MAX(hid) FROM dbo.Employees
     WHERE hid.GetAncestor(1) = @mgr_hid);
  SET @hid = @mgr_hid.GetDescendant(@last_child_hid, NULL);
END

INSERT INTO dbo.Employees(empid, hid, empname, salary)
  VALUES(@empid, @hid, @empname, @salary);
GO
```

If the input employee is the first node in the hierarchy, the procedure uses the **HIERARCHYID::GetRoot()** method to assign the hid value for the root. Otherwise, the procedure queries the last child hid value of the new employee's manager; it then invokes the **GetDescendant** method to produce a value that positions the new node after the last child of that manager. Run the following code to populate the table with a few employees:

```
EXEC dbo.usp_AddEmp @empid =  1, @mgrid = NULL, @empname = 'David'  , @salary = $10000.00;
EXEC dbo.usp_AddEmp @empid =  2, @mgrid =    1, @empname = 'Eitan'  , @salary = $7000.00;
EXEC dbo.usp_AddEmp @empid =  3, @mgrid =    1, @empname = 'Ina'    , @salary = $7500.00;
EXEC dbo.usp_AddEmp @empid =  4, @mgrid =    2, @empname = 'Seraph' , @salary = $5000.00;
EXEC dbo.usp_AddEmp @empid =  5, @mgrid =    2, @empname = 'Jiru'   , @salary = $5500.00;
EXEC dbo.usp_AddEmp @empid =  6, @mgrid =    2, @empname = 'Steve'  , @salary = $4500.00;
EXEC dbo.usp_AddEmp @empid =  7, @mgrid =    3, @empname = 'Aaron'  , @salary = $5000.00;
EXEC dbo.usp_AddEmp @empid =  8, @mgrid =    5, @empname = 'Lilach' , @salary = $3500.00;
EXEC dbo.usp_AddEmp @empid =  9, @mgrid =    7, @empname = 'Rita'   , @salary = $3000.00;
EXEC dbo.usp_AddEmp @empid = 10, @mgrid =    5, @empname = 'Sean'   , @salary = $3000.00;
EXEC dbo.usp_AddEmp @empid = 11, @mgrid =    7, @empname = 'Gabriel', @salary = $3000.00;
EXEC dbo.usp_AddEmp @empid = 12, @mgrid =    9, @empname = 'Emilia' , @salary = $2000.00;
EXEC dbo.usp_AddEmp @empid = 13, @mgrid =    9, @empname = 'Michael', @salary = $2000.00;
EXEC dbo.usp_AddEmp @empid = 14, @mgrid =    9, @empname = 'Didi'   , @salary = $1500.00;
```

Querying the Hierarchy

If you query the hid value, you get its binary representation, which is not very meaningful. You can use the **ToString** method to get a more logical string representation of the value, which shows the path with a slash sign used as a separator between the levels. For example, run the following query to get both the binary and logical representations of the hid value:

```
SELECT hid, hid.ToString() AS path, lvl, empid, empname, salary

FROM dbo.Employees

ORDER BY hid;
```

Recall that HIERARCHYID values provide topological sorting and that the **GetLevel** method produces the level in the hierarchy. Using these, you can easily produce a graphical depiction of the hierarchy—simply sort the rows by hid, and produce indentation based on the lvl column as follows:

```
SELECT

  REPLICATE(' | ', lvl) + empname AS emp,

  hid.ToString() AS path

FROM dbo.Employees

ORDER BY hid;
```

To get all subordinates of an employee (subtree), you can use a method called **IsDescendantOf**. This method accepts a node's HIERARCHYID value as input and returns 1 if the queried node is a descendant of the input node. For example, the following query returns all subordinates—direct and indirect—of employee 3:

```
SELECT C.empid, C.empname, C.lvl

FROM dbo.Employees AS P

  JOIN dbo.Employees AS C

    ON P.empid = 3

    AND C.hid.IsDescendantOf(P.hid) = 1;
```

You can also the **IsDescendantOf** method to return all managers of a given employee. For example, the following query returns all managers of employee 14:

```
SELECT P.empid, P.empname, P.lvl

FROM dbo.Employees AS P

  JOIN dbo.Employees AS C

    ON C.empid = 14

    AND C.hid.IsDescendantOf(P.hid) = 1;
```

To get a whole level of subordinates of a certain employee, use the **GetAncestor** method. This method accepts a number (call it *n*) as input and returns the HIERARCHYID value of the ancestor of the queried node, *n* levels above. For example, the following query returns direct subordinates (1 level below) of employee 9:

```
SELECT C.empid, C.empname

FROM dbo.Employees AS P

  JOIN dbo.Employees AS C
```

```
ON P.empid = 9

AND C.hid.GetAncestor(1) = P.hid;
```

Other Supported Methods

You can use several other methods to manipulate the HIERARCHYID data type, including **Parse**, **GetReparentedValue**, **Read**, and **Write**.

The HIERARCHYID::Parse method converts a canonical string representation of a hierarchical value to HIERARCHYID. This is the same as using CAST(<*string_val*> AS HIERARCHYID).

The **GetReparentedValue** method helps you reparent nodes. It accepts two arguments—*@old_root* and *@new_root*—and returns a value in which the @old_root portion of the path is replaced with *@new_root*. For example, the node you query currently has the path /1/1/2/3/2/ (logical representation), *@old_root* is /1/1/, and *@new_root* is /2/1/4/. The **GetReparentedValue** method returns /2/1/4/2/3/2/. As mentioned earlier in regard to the **GetDescendant** method, the **GetReparentedValue** method also does not guarantee unique HIERARCHYID values. To enforce uniqueness, you must define a primary key, a unique constraint, or a unique index on the column.

The **Read** and **Write** methods, available only in CLR code, are used to read from a BinaryReader and write to a BinaryWriter. In Transact-SQL, you simply use the CAST function to convert a binary value to a HIERARCHYID value and vice versa. Similarly, you can use the CAST function to convert a logical string representation of the path to HIERARCHYID and vice versa.

Table Types and Table-Valued Parameters

SQL Server 2008 introduces table types and table-valued parameters that help abbreviate your code and improve its performance. Table types allow easy reuse of table definition by table variables, and table-valued parameters enable you to pass a parameter of a table type to stored procedures and functions.

Table Types

Table types enable you to save a table definition in the database and use it later to define table variables and parameters to stored procedures and functions. Because table types let you reuse a table definition, they ensure consistency and reduce chances for errors.

You use the CREATE TYPE statement to create a new table type. For example, the following code defines a table type called OrderIDs in the AdventureWorks database:

```
USE AdventureWorks;

GO

CREATE TYPE dbo.OrderIDs AS TABLE

( pos INT NOT NULL PRIMARY KEY,

  orderid INT NOT NULL UNIQUE );
```

When declaring a table variable, simply specify the table type name as the data type of the variable. For example, the following code defines a table variable called *@T* of the OrderIDs type, inserts three rows into the table variable, and then queries it:

```
DECLARE @T AS dbo.OrderIDs;

INSERT INTO @T(pos, orderid) VALUES(1, 51480),(2, 51973),(3, 51819);

SELECT pos, orderid FROM @T ORDER BY pos;
```

To get metadata information about table types in the database, query the view **sys.table_types**.

Table-Valued Parameters

You can now use table types as the types for input parameters of stored procedures and functions. Currently, table-valued parameters are read only, and you must define them as such by using the READONLY keyword.

A common scenario where table-valued parameters are very useful is passing an "array" of keys to a stored procedure. Before SQL Server 2008, common ways to meet this need were based on dynamic SQL, a split function, XML, and other techniques. The approach using dynamic SQL involved the risk of SQL Injection and did not provide efficient reuse of execution plans. Using a split function was complicated, and using XML was complicated and nonrelational. (For details about this scenario, see "Arrays and Lists in SQL Server" by SQL Server MVP Erland Sommarskog at http://www.sommarskog.se/arrays-in-sql.html [http://www.sommarskog.se/arrays-in-sql.html] .)

In SQL Server 2008, you simply pass the stored procedure a table-valued parameter. There is no risk of SQL Injection, and there is opportunity for efficient reuse of execution plans. For example, the following procedure accepts a table-valued parameter of the OrderIDs type with a set of order IDs and returns all orders from the SalesOrderHeader table whose order IDs appear in the input table-valued parameter, sorted by the pos column:

```
CREATE PROC dbo.usp_getorders(@T AS dbo.OrderIDs READONLY)

AS

SELECT O.SalesOrderID, O.OrderDate, O.CustomerID, O.TotalDue

FROM Sales.SalesOrderHeader AS O

  JOIN @T AS T

    ON O.SalesOrderID = T.orderid

ORDER BY T.pos;

GO
```

The following code invokes the stored procedure:

```
DECLARE @MyOrderIDs AS dbo.OrderIDs;

INSERT INTO @MyOrderIDs(pos, orderid)

  VALUES(1, 51480),(2, 51973),(3, 51819);

EXEC dbo.usp_getorders @T = @MyOrderIDs;
```

Note that when you do not provide a parameter value, a table-valued parameter defaults to an empty table. This is important to stress because this case might be confused with an actual empty table passed to the procedure. Also note that you cannot set variables and parameters of a table type to NULL.

SQL Server 2008 also enhances client APIs to support defining and populating table-valued parameters. Table-valued parameters are treated internally like table variables. Their scope is the batch (procedure, function). They have several advantages in some cases over temporary tables and other alternative methods:

- They are strongly typed.
- SQL Server does not maintain distribution statistics (histograms) for them; therefore, they do not cause recompilations.
- They are not affected by a transaction rollback.
- They provide a simple programming model.

MERGE Statement

The new MERGE statement is a standard statement that combines INSERT, UPDATE, and DELETE actions as a single atomic operation based on conditional logic. Besides being performed as an atomic operation, the MERGE statement

is more efficient than applying those actions individually.

The statement refers to two tables: a target table specified in the MERGE INTO clause and a source table specified in the USING clause. The target table is the target for the modification, and the source table data can be used to modify the target.

The semantics (as well as optimization) of a MERGE statement are similar to those of an outer join. You specify a predicate in the ON clause that defines which rows in the source have matches in the target, which rows do not, and which rows in the target do not have a match in the source. You have a clause for each case that defines which action to take—WHEN MATCHED THEN, WHEN NOT MATCHED [BY TARGET] THEN, and WHEN NOT MATCHED BY SOURCE THEN. Note that you do not have to specify all three clauses, but only the ones you need.

As with other modification statements, the MERGE statement also supports the OUTPUT clause, which enables you to return attributes from the modified rows. As part of the OUTPUT clause, you can invoke the **$action** function, which returns the action that modified the row ('INSERT', 'UPDATE', 'DELETE').

To demonstrate the MERGE statement, the following code creates the tables Customers and CustomersStage in **tempdb** for test purposes and populates them with sample data:

```
USE tempdb;

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

CREATE TABLE dbo.Customers

(

  custid      INT         NOT NULL,

  companyname VARCHAR(25) NOT NULL,

  phone       VARCHAR(20) NOT NULL,

  address     VARCHAR(50) NOT NULL,

  CONSTRAINT PK_Customers PRIMARY KEY(custid)

);

INSERT INTO dbo.Customers(custid, companyname, phone, address)

  VALUES

  (1, 'cust 1', '(111) 111-1111', 'address 1'),

  (2, 'cust 2', '(222) 222-2222', 'address 2'),

  (3, 'cust 3', '(333) 333-3333', 'address 3'),

  (4, 'cust 4', '(444) 444-4444', 'address 4'),

  (5, 'cust 5', '(555) 555-5555', 'address 5');


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

CREATE TABLE dbo.CustomersStage

(

  custid      INT         NOT NULL,

  companyname VARCHAR(25) NOT NULL,

  phone       VARCHAR(20) NOT NULL,
```

```
  address     VARCHAR(50) NOT NULL,
  CONSTRAINT PK_CustomersStage PRIMARY KEY(custid)
);
INSERT INTO dbo.CustomersStage(custid, companyname, phone, address)
  VALUES
  (2, 'AAAAA', '(222) 222-2222', 'address 2'),
  (3, 'cust 3', '(333) 333-3333', 'address 3'),
  (5, 'BBBBB', 'CCCCC', 'DDDDD'),
  (6, 'cust 6 (new)', '(666) 666-6666', 'address 6'),
  (7, 'cust 7 (new)', '(777) 777-7777', 'address 7');
```

The following MERGE statement defines the Customers table as the target for the modification and the CustomersState table as the source. The MERGE condition matches the custid attribute in the source with the custid attribute in the target. When a match is found in the target, the target customer's attributes are overwritten with the source customer attributes. When a match is not found in the target, a new row is inserted into the target, using the source customer attributes. When a match is not found in the source, the target customer row is deleted:

```
MERGE INTO dbo.Customers AS TGT
USING dbo.CustomersStage AS SRC
  ON TGT.custid = SRC.custid
WHEN MATCHED THEN
  UPDATE SET
    TGT.companyname = SRC.companyname,
    TGT.phone = SRC.phone,
    TGT.address = SRC.address
WHEN NOT MATCHED THEN
  INSERT (custid, companyname, phone, address)
  VALUES (SRC.custid, SRC.companyname, SRC.phone, SRC.address)
WHEN NOT MATCHED BY SOURCE THEN
  DELETE
OUTPUT
  $action, deleted.custid AS del_custid, inserted.custid AS ins_custid;
```

This MERGE statement updates three rows (customers 2, 3, and 5), inserts two rows (customers 6 and 7), and deletes two rows (customers 1 and 4).

The MERGE statement lets you specify an additional predicate in all WHEN clauses (add an AND operator followed by the predicate). For the action to take place, besides the original ON predicate, the additional predicate must also hold true.

Grouping Sets

SQL Server 2008 introduces several extensions to the GROUP BY clause that enable you to define multiple groupings in the same query. These extensions are: the GROUPING SETS, CUBE, and ROLLUP subclauses of the GROUP BY clause and the GROUPING_ID function. The new extensions are standard and should not be confused with the older, nonstandard CUBE and ROLLUP options.

GROUPING SETS, CUBE, and ROLLUP Subclauses

To demonstrate the new extensions, let's query the Orders table that the following code creates and populates:

```
USE tempdb;

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

CREATE TABLE dbo.Orders

(

  orderid   INT        NOT NULL,

  orderdate DATETIME   NOT NULL,

  empid     INT        NOT NULL,

  custid    VARCHAR(5) NOT NULL,

  qty       INT        NOT NULL,

  CONSTRAINT PK_Orders PRIMARY KEY(orderid)

);

INSERT INTO dbo.Orders (orderid, orderdate, empid, custid, qty)

VALUES

  (30001, '20060802', 3, 'A', 10), (10001, '20061224', 1, 'A', 12),

  (10005, '20061224', 1, 'B', 20), (40001, '20070109', 4, 'A', 40),

  (10006, '20070118', 1, 'C', 14), (20001, '20070212', 2, 'B', 12),

  (40005, '20080212', 4, 'A', 10), (20002, '20080216', 2, 'C', 20),

  (30003, '20080418', 3, 'B', 15), (30004, '20060418', 3, 'C', 22),

  (30007, '20060907', 3, 'D', 30);
```

Without the extensions, a single query normally defines one "grouping set" (a set of attributes to group by) in the GROUP BY clause. If you want to calculate aggregates for multiple grouping sets, you usually need multiple queries. If you want to unify the result sets of multiple GROUP BY queries, each with a different grouping set, you must use the UNION ALL set operation between the queries.

You might need to calculate and store aggregates for various grouping sets in a table. By preprocessing and materializing the aggregates, you can support applications that require fast response time for aggregate requests. However, the aforementioned approach, in which you have a separate query for each grouping set, is very inefficient. This approach requires a separate scan of the data for each grouping set and expensive calculation of aggregates.

With the new GROUPING SETS subclause, you simply list all grouping sets that you need. Logically, you get the same result set as you would by unifying the result sets of multiple queries. However, with the GROUPING SETS subclause, which requires much less code, SQL Server optimizes data access and the calculation of aggregates. SQL Server will not necessarily need to scan data once for each grouping set; plus, in some cases it calculates higher-level aggregates based on lower-level aggregates instead of re-aggregating base data.

For example, the following query uses the new GROUPING SETS subclause to return aggregates for four grouping sets:

```
SELECT custid, empid, YEAR(orderdate) AS orderyear, SUM(qty) AS qty

FROM dbo.Orders

GROUP BY GROUPING SETS (

  ( custid, empid, YEAR(orderdate) ),

  ( custid, YEAR(orderdate)       ),

  ( empid, YEAR(orderdate)        ),

  () );
```

The grouping sets defined in this query are (custid, empid, YEAR(orderdate)), (custid, YEAR(orderdate)), (empid, YEAR(orderdate)), and (). The last is an empty grouping set representing ALL. It is similar to an aggregate query with no GROUP BY clause, in which you treat the whole table as a single group.

The two new CUBE and ROLLUP subclauses should be considered as abbreviations to the GROUPING SETS subclause. The CUBE subclause produces the power set of the set of elements listed in its parentheses. In other words, it produces all possible grouping sets that can be formed out of the elements listed in parentheses, including the empty grouping set. For example, the following use of CUBE:

```
CUBE( a, b, c )
```

is logically equivalent to:

```
GROUPING SETS((a),(b),(c),(a, b),(a, c),(b, c),(a, b, c),())
```

For *n* elements. CUBE produces 2^*n* grouping sets.

Out of the elements listed in its parentheses, the ROLLUP subclause produces only the grouping sets that have business value, assuming a hierarchy between the elements. For example, the following use of ROLLUP:

```
ROLLUP( country, region, city )
```

is logically equivalent to:

```
GROUPING SETS((country, region, city),(country, region),(country),())
```

Notice that cases that have no business value, assuming a hierarchy between the elements—such as (city)—were not produced. There might be multiple cities with the same name in the world, and even within the same country, so there is no business value in aggregating them.

Grouping Sets Algebra

You are not restricted to only one subclause in the GROUP BY clause; you can specify multiple subclauses separated by commas. The comma serves as a product operator, meaning that you get a Cartesian product of the grouping sets represented by each subclause. For example, the following code represents a Cartesian product between two GROUPING SETS subclauses:

```
GROUPING SETS ( (a, b), (c, d) ), GROUPING SETS ( (w, x), (y, z) )
```

This code is logically equivalent to:

```
GROUPING SETS ( (a, b, w, x), (a, b, y, z), (c, d, w, x), (c, d, y, z) )
```

Bearing in mind that the CUBE and ROLLUP options are simply abbreviations of the GROUPING SETS subclause, you can also use CUBE and ROLLUP subclauses as part of a Cartesian product.

GROUPING_ID Function

The GROUPING_ID function lets you identify the grouping set that each result row belongs to. As input, you provide all attributes that participate in any grouping set. The function produces an integer result that is a bitmap, in which each bit represents a different attribute. This way the function produces a unique integer for each grouping set.

The following query shows an example of using the GROUPING_ID function to identify the grouping set:

```
SELECT
  GROUPING_ID(
    custid, empid,
    YEAR(orderdate), MONTH(orderdate), DAY(orderdate) ) AS grp_id,
  custid, empid,
  YEAR(orderdate) AS orderyear,
  MONTH(orderdate) AS ordermonth,
  DAY(orderdate) AS orderday,
  SUM(qty) AS qty
FROM dbo.Orders
GROUP BY
  CUBE(custid, empid),
  ROLLUP(YEAR(orderdate), MONTH(orderdate), DAY(orderdate));
```

Notice that in the output, shown here in abbreviated form, each grouping set is represented by a unique integer:

```
grp_id  custid empid      orderyear   ordermonth  orderday    qty
------- ------ ----------- ----------- ----------- ----------- -----------
0       C      3           2006        4           18          22
16      NULL   3           2006        4           18          22
24      NULL   NULL        2006        4           18          22
25      NULL   NULL        2006        4           NULL        22
0       A      3           2006        8           2           10
16      NULL   3           2006        8           2           10
24      NULL   NULL        2006        8           2           10
25      NULL   NULL        2006        8           NULL        10
0       D      3           2006        9           7           30
16      NULL   3           2006        9           7           30
...
```

For example, the integer 25 represents the grouping set (orderyear, ordermonth). The bits representing the elements that are part of the grouping set are turned off (ordermonth – 2 and orderyear – 4), and the bits representing the elements that are not part of the grouping set are turned on (orderday – 1, empid – 8, and custid – 16). The integer 25 is achieved by adding the values represented by the bits that are turned on: 1 + 8 + 16 = 25. This feature is especially useful when you need to materialize the aggregates and then query only specific grouping sets. You can cluster the table by the grp_id attribute, which would allow SQL Server to efficiently satisfy a request for a specific grouping set.

DDL Trigger Enhancements

In SQL Server 2008, the type of events on which you can now create DDL triggers is enhanced to include stored procedures that perform DDL-like operations. This gives you more complete coverage of DDL events that you can capture with triggers. The XML schema for events is installed as part of the database engine installation in the directory C:\Program Files\Microsoft SQL Server\100\Tools\Binn\schemas\sqlserver\2006\11\events\events.xsd and can also be found at http://schemas.microsoft.com/sqlserver [http://schemas.microsoft.com/sqlserver/default.aspx] .

Many stored procedures perform DDL-like operations. Before SQL Server 2008, you could not capture their invocation with a trigger. Now you can capture many new events that fire as a result of calls to such procedures. You can find the full list of trappable events in SQL Server Books Online. For example, the stored procedure **sp_rename** now fires a trigger created on the new RENAME event. To demonstrate this type of trigger, the following code creates a database called testdb and, within it, a trigger on the RENAME event that prints the source and target entity details for test purposes:

```
USE master;
GO
IF DB_ID('testdb') IS NOT NULL DROP DATABASE testdb;
CREATE DATABASE testdb;
GO
USE testdb;
GO
CREATE TRIGGER trg_testdb_rename ON DATABASE FOR RENAME
AS

DECLARE
  @SchemaName       AS SYSNAME =
    EVENTDATA().value('(/EVENT_INSTANCE/SchemaName)[1]', 'sysname'),
  @TargetObjectName AS SYSNAME =
    EVENTDATA().value('(/EVENT_INSTANCE/TargetObjectName)[1]', 'sysname'),
  @ObjectName       AS SYSNAME =
    EVENTDATA().value('(/EVENT_INSTANCE/ObjectName)[1]', 'sysname'),
  @NewObjectName    AS SYSNAME =
    EVENTDATA().value('(/EVENT_INSTANCE/NewObjectName)[1]', 'sysname');

DECLARE
  @msg AS NVARCHAR(1000) =
N'RENAME event occurred.
SchemaName: ' + @SchemaName + N'
TargetObjectName: ' + @TargetObjectName + N'
ObjectName: ' + @ObjectName + N'
```

```
NewObjectName: ' + @NewObjectName;


PRINT @msg;

GO
```

To test the trigger, the following code creates a table called dbo.T1 with a column called col1 and runs the **sp_rename** procedure to rename the column to col2:

```
CREATE TABLE dbo.T1(col1 INT);

EXEC sp_rename 'dbo.T1.col1', 'col2', 'COLUMN';
```

The trigger on the RENAME event fires and prints the following message:

```
RENAME event occurred.

SchemaName: dbo

TargetObjectName: T1

ObjectName: col1

NewObjectName: col2
```

Sparse Columns

Sparse columns are columns that are optimized for the storage of NULLs. To define a column as sparse, specify the SPARSE attribute as part of the column definition. Sparse columns consume no storage for NULLs, even with fixed-length types; however, when a column is marked as sparse, storage of non-NULL values becomes more expensive than usual. Therefore, you should define a column as sparse only when it will store a large percentage of NULLs. SQL Server Books Online provides recommendations for the percentage of NULLs that justify making a column sparse for each data type.

Querying and manipulation of sparse columns is the same as for regular columns, with one exception described later in this paper. For example, the following code creates a table named T1 in **tempdb** (for test purposes), marks three of its columns with the SPARSE attribute, inserts a couple of rows, and queries the table:

```
USE tempdb;

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

CREATE TABLE dbo.T1

(

  keycol INT                NOT NULL PRIMARY KEY,

  col1   VARCHAR(20)          NOT NULL,

  col2   INT          SPARSE     NULL,

  col3   CHAR(10)       SPARSE     NULL,

  col4   NUMERIC(12, 2) SPARSE     NULL

);
```

```
INSERT INTO dbo.T1(keycol, col1, col2) VALUES(1, 'a', 10);

INSERT INTO dbo.T1(keycol, col1, col4) VALUES(2, 'b', 20.00);


SELECT keycol, col1, col2, col3, col4

FROM dbo.T1;
```

There are several restrictions on using sparse columns not covered in this paper; see SQL Server Books Online for complete information.

SQL Server 2008 lets you define a column set that combines all sparse columns of a table into a single XML column. You might want to consider this option when you have a large number of sparse columns in a table (more than 1,024) and operating on them individually might be cumbersome.

To define a column set, specify the following as part of the CREATE TABLE statement:

```
<column_set_name> XML column_set FOR ALL_SPARSE_COLUMNS
```

For example, the following code recreates the table T1 with a column set named cs:

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

CREATE TABLE dbo.T1

(

  keycol INT                 NOT NULL PRIMARY KEY,

  col1   VARCHAR(20)         NOT NULL,

  col2   INT          SPARSE     NULL,

  col3   CHAR(10)      SPARSE     NULL,

  col4   NUMERIC(12, 2) SPARSE     NULL,

  cs     XML column_set FOR ALL_SPARSE_COLUMNS

);
```

You can use the same code that was used earlier to insert rows into the table, and then query it:

```
INSERT INTO dbo.T1(keycol, col1, col2) VALUES(1, 'a', 10);

INSERT INTO dbo.T1(keycol, col1, col4) VALUES(2, 'b', 20.00);


SELECT keycol, col1, col2, col3, col4

FROM dbo.T1;
```

But you can also operate on the column set by using XML operations instead of relational operations. For example,

the following code inserts a row into the table by using the column set:

```
INSERT dbo.T1(keycol, col1, cs)
  VALUES(3, 'c', '<col3>CCCCCCCCCC</col3><col4>30.00</col4>');
```

NULL is assumed for a column that is omitted from the XML value, such as col2 in this case.

Note that if you have a column set defined in the table, SELECT * does not return the same result as a SELECT statement with an explicit list of all columns. SELECT * returns all sparse columns as a single XML value in the column set. To demonstrate this, run the following code:

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

This code returns the following output:

```
keycol      col1       cs
----------- ---------- -----------------------------------------
1           a          <col2>10</col2>
2           b          <col4>20.00</col4>
3           c          <col3>CCCCCCCCCC</col3><col4>30.00</col4>
```

If you explicitly list the columns in the SELECT clause, all result columns are returned as relational ones.

Another new feature that can be used in conjunction with sparse columns is filtered indexes. This feature is explained in the next section.

Filtered Indexes and Statistics

SQL Server 2008 introduces filtered indexes and statistics. You can now create a nonclustered index based on a predicate, and only the subset of rows for which the predicate holds true are stored in the index B-Tree. Similarly, you can manually create statistics based on a predicate. The optimizer has the logic to figure out when such filtered indexes and statistics are useful.

Well-designed filtered indexes can improve query performance and plan quality because they are smaller than nonfiltered indexes. Also, filtered statistics—whether created automatically for a filtered index or manually—are more accurate than nonfiltered statistics because they need to cover only a subset of rows from the table.

You can also reduce index maintenance cost by using filtered indexes because there is less data to maintain. This includes modifications against the index, index rebuilds, and the cost of updating statistics. Filtered indexes also obviously reduce storage costs.

Let’s look at a few examples that demonstrate filtered indexes. The following code creates an index on the CurrencyRateID column in the Sales.SalesOrderHeader table, with a filter that excludes NULLs:

```
USE AdventureWorks;
GO
CREATE NONCLUSTERED INDEX idx_currate_notnull
  ON Sales.SalesOrderHeader(CurrencyRateID)
```

```
WHERE CurrencyRateID IS NOT NULL;
```

Considering query filters, besides the IS NULL predicate that explicitly looks for NULLs, all other predicates exclude NULLs, so the optimizer knows that there is the potential to use the index. For example, the plan for the following query shows that the index is used:

```
SELECT *

FROM Sales.SalesOrderHeader

WHERE CurrencyRateID = 4;
```

The CurrencyRateID column has a large percentage of NULLs; therefore, this index consumes substantially less storage than a nonfiltered one on the same column. You can also create similar indexes on sparse columns.

The following code creates a nonclustered index on the Freight column, filtering rows where the Freight is greater than or equal to 5000.00:

```
CREATE NONCLUSTERED INDEX idx_freight_5000_or_more

  ON Sales.SalesOrderHeader(Freight)

  WHERE Freight >= 5000.00;
```

The optimizer considers using an index when a subinterval of the index filtered interval is requested in the query filter:

```
SELECT *

FROM Sales.SalesOrderHeader

WHERE Freight BETWEEN 5500.00 AND 6000.00;
```

Filtered indexes can also be defined as UNIQUE and have an INCLUDE clause as with regular nonclustered indexes.

SQL/CLR Enhancements

Common language runtime (CLR) support is enhanced in several ways in SQL Server 2008. Enhancements to CLR UDTs were described earlier. This section describes enhancements to user-defined aggregates (UDAs) and table-valued functions (TVFs).

Enhancements to User-Defined Aggregates

The new support for large UDTs was covered earlier in this paper. Similarly, SQL Server 2008 introduces support for large UDAs—the maximum size in bytes of a persisted value can now exceed 8,000 bytes and reach up to 2 GB. To allow a UDA to exceed 8,000 bytes, specify -1 in the **MaxByteSize** attribute; otherwise, you must specify a value that is smaller than or equal to 8,000, in which case the size in bytes cannot exceed the specified value.

UDAs are enhanced in another way as well—they now support multiple inputs. An example of a UDA that can benefit from both enhancements is one that performs string concatenation. The UDA can accept two input parameters: the column holding the string to concatenate and the character to use as a separator. The UDA can return an output larger than 8,000 bytes.

Enhancements to Table-Valued Functions

CLR TVFs now support a new ORDER clause as part of the CREATE FUNCTION DDL statement. You can use this clause

to specify column names in the output table when you know that rows will always be returned in that order. This can help the optimizer when you query the table function and rely on those columns for ordering purposes (such as when they are used in ORDER BY, GROUP BY, DISTINCT, and INSERT when the (?) target has an (?) index).

For example, the following C#® code defines a function called fn_split that accepts a separated list of values and a separator, and then splits the input string to the individual elements. The function returns a table result with two columns—pos and element—with a row for each element. The pos column represents the position of the element in the list, and the element column returns the element itself:

```
using System;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.Collections;
using System.Collections.Generic;


public partial class TVF
{
    // Struct used in string split functions
    struct row_item
    {
        public string item;
        public int pos;
    }

    // Split array of strings and return a table
    // FillRowMethodName = "ArrSplitFillRow"
    [SqlFunction(FillRowMethodName = "ArrSplitFillRow",
     DataAccess = DataAccessKind.None,
     TableDefinition = "pos INT, element NVARCHAR(MAX)")]
    public static IEnumerable fn_split(SqlString inpStr,
        SqlString charSeparator)
    {
        string locStr;
        string[] splitStr;
        char[] locSeparator = new char[1];
        locSeparator[0] = (char)charSeparator.Value[0];
        if (inpStr.IsNull)
            locStr = "";
        else
```

```
            locStr = inpStr.Value;
        splitStr = locStr.Split(locSeparator,
              StringSplitOptions.RemoveEmptyEntries);
        //locStr.Split(charSeparator.ToString()[0]);
        List<row_item> SplitString = new List<row_item>();
        int i = 1;
        foreach (string s in splitStr)
        {
            row_item r = new row_item();
            r.item = s;
            r.pos = i;
            SplitString.Add(r);
            ++i;
        }
        return SplitString;
    }

    public static void ArrSplitFillRow(
      Object obj, out int pos, out string item)
    {
        pos = ((row_item)obj).pos;
        item = ((row_item)obj).item;
    }
}
```

The function always returns the rows in pos order; however, you cannot rely on this order when querying the function unless you specify an ORDER BY clause in the outer query.

Assuming that the path for the assembly's .dll file is C:\TVF\TVF\bin\Debug\TVF.dll, the following code creates the assembly in **tempdb** for test purposes:

```
USE tempdb;
CREATE ASSEMBLY TVF FROM 'C:\TVF\TVF\bin\Debug\TVF.dll';
```

The following code registers two functions based on the CLR fn_split function: fn_split_no_order does not have the ORDER clause, and fn_split_order_by_pos specifies the ORDER clause with pos as the ordering column:

```
CREATE FUNCTION dbo.fn_split_no_order
```

```
  (@string AS NVARCHAR(MAX), @separator AS NCHAR(1))
RETURNS TABLE(pos INT, element NVARCHAR(4000))
EXTERNAL NAME TVF.TVF.fn_split;
GO
CREATE FUNCTION dbo.fn_split_order_by_pos
  (@string AS NVARCHAR(MAX), @separator AS NCHAR(1))
RETURNS TABLE(pos INT, element NVARCHAR(4000))
ORDER (pos)
EXTERNAL NAME TVF.TVF.fn_split;
GO
```

Now consider the following queries:

```
SELECT *
FROM dbo.fn_split_no_order(
  N'a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z',
  N',') AS T
ORDER BY pos;

SELECT *
FROM dbo.fn_split_order_by_pos(
  N'a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z',
  N',') AS T
ORDER BY pos;
```

The first queries the table function that was registered without the ORDER clause, and the second queries the table function that was registered with ORDER(pos). Both request the data sorted by pos. If you examine the execution plans for both queries, you see that the plan for the first query involves sorting, while the plan for the second query does not. Also, the cost estimate for the first plan is about 10 times higher than the cost estimate for the second plan.

Object Dependencies

SQL Server 2008 delivers several objects that provide reliable discovery of object dependencies, replacing the unreliable older **sys.sql_dependencies** view and the **sp_depends** stored procedure. The new objects provide information about dependencies that appear in static code, including both schema-bound and non-schema-bound objects as well as cross-database and even cross-server dependencies. The new objects do not cover dependencies that appear in dynamic SQL code or in CLR code.

To see how to query the new object dependency information, first run the following code to create several objects in the **tempdb** database for test purposes:

```
USE tempdb;
```

```
IF OBJECT_ID('dbo.Proc1', 'P') IS NOT NULL DROP PROC dbo.Proc1;

IF OBJECT_ID('dbo.V1', 'V') IS NOT NULL DROP VIEW dbo.V1;

IF OBJECT_ID('dbo.V2', 'V') IS NOT NULL DROP VIEW dbo.V2;

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

IF OBJECT_ID('dbo.T2', 'U') IS NOT NULL DROP TABLE dbo.T2;

GO


CREATE PROC dbo.Proc1

AS

SELECT * FROM dbo.T1;

EXEC('SELECT * FROM dbo.T2');

GO

CREATE TABLE dbo.T1(col1 INT);

CREATE TABLE dbo.T2(col2 INT);

GO

CREATE VIEW dbo.V1

AS

SELECT col1 FROM dbo.T1;

GO

CREATE VIEW dbo.V2

AS

SELECT col1 FROM dbo.T1;

GO
```

SQL Server 2008 introduces three new objects that provide object dependency information: the **sys.sql_expression_dependencies** catalog view, and the **sys.dm_sql_referenced_entities** and **sys.dm_sql_referencing_entities** dynamic management functions (DMFs).

The **sys.sql_expression_dependencies** view provides object dependencies by name. It replaces the older **sys.sql_dependencies** view. The following query against **sys.sql_expression_dependencies** shows all dependencies in the current database:

```
SELECT

  OBJECT_SCHEMA_NAME(referencing_id) AS srcschema,

  OBJECT_NAME(referencing_id) AS srcname,

  referencing_minor_id AS srcminorid,

  referenced_schema_name AS tgtschema,

  referenced_entity_name AS tgtname,
```

```
  referenced_minor_id AS tgtminorid
FROM sys.sql_expression_dependencies;
```

This query produces the following output:

```
srcschema srcname srcminorid tgtschema tgtname tgtminorid
--------- ------- ---------- --------- ------- ----------
dbo       Proc1   0          dbo       T1      0
dbo       V1      0          dbo       T1      0
dbo       V2      0          dbo       T1      0
```

Notice that the query identified all dependencies in static code but not dependencies in dynamic code (the reference to dbo.T2 from dbo.Proc1).

The **sys.dm_sql_referenced_entities** DMF provides all entities that the input entity references—in other words, all entities that the input entity depends on. For example, the following code returns all entities that dbo.Proc1 depends on:

```
SELECT
  referenced_schema_name AS objschema,
  referenced_entity_name AS objname,
  referenced_minor_name  AS minorname,
  referenced_class_desc  AS class
FROM sys.dm_sql_referenced_entities('dbo.Proc1', 'OBJECT');
```

This code returns the following output:

```
objschema objname   minorname class
--------- --------- --------- ----------------
dbo       T1        NULL      OBJECT_OR_COLUMN
dbo       T1        col1      OBJECT_OR_COLUMN
```

The output shows that dbo.Proc1 depends on the table dbo.T1 and the column dbo.T1.col1. Again, dependencies that appear in dynamic code are not identified.

The **sys.dm_sql_referencing_entities** DMF provides all entities that reference the input entity—in other words, all entities that depend on the input entity. For example, the following code returns all entities that depend on dbo.T1:

```
SELECT
  referencing_schema_name AS objschema,
```

```
  referencing_entity_name AS objname,
  referencing_class_desc  AS class
FROM sys.dm_sql_referencing_entities('dbo.T1', 'OBJECT');
```

This code returns the following output:

```
objschema objname   class
--------- --------- ----------------
dbo       Proc1     OBJECT_OR_COLUMN
dbo       V1        OBJECT_OR_COLUMN
dbo       V2        OBJECT_OR_COLUMN
```

The output shows that dbo.Proc1, dbo.V1, and dbo.V2 depend on dbo.T1.

Change Data Capture

Change data capture is a new mechanism in SQL Server 2008 that enables you to easily track data changes in a table. The changes are read by a capture process from the transaction log and recorded in change tables. Those change tables mirror the columns of the source table and also contain metadata information that can be used to deduce the changes that took place. Those changes can be consumed in a convenient relational format through TVFs.

An extract, transform, and load (ETL) process in SQL Server Integration Services that applies incremental updates to a data warehouse is just one example of an application that can benefit from change data capture.

Here, I'll demonstrate a simple process of capturing changes against an Employees table in a database called testdb.

Before you can enable tables for change data capture, you must first enable the database for change data capture by using the stored procedure **sys.sp_cdc_enable_db**. This stored procedure creates in the database several system objects related to change data capture, including the cdc schema, the cdc user, tables, jobs, stored procedures, and functions. To check whether the database is enabled for change data capture, query the **is_cdc_enabled** column in **sys.databases**. The following code creates a database called testdb and enables change data capture in the database:

```
USE master;
IF DB_ID('testdb') IS NOT NULL DROP DATABASE testdb;
CREATE DATABASE testdb;
GO
USE testdb;
EXECUTE sys.sp_cdc_enable_db;
```

To disable change data capture on the database, use the stored procedure **sys.sp_cdc_disable_db**.

The capture process starts reading changes from the log and recording them in the change tables as soon as the first table in the database is enabled for change data capture. To enable change data capture for a table, you use the **sys.sp_cdc_enable_table** stored procedure. Note that SQL Server Agent must be running for the capture processes to work. The following code creates a table NAMED Employees in the testdb database and inserts one employee row into the table:

```
CREATE TABLE dbo.Employees
(
  empid  INT         NOT NULL,
  name   VARCHAR(30) NOT NULL,
  salary MONEY       NOT NULL
);
INSERT INTO dbo.Employees(empid, name, salary) VALUES(1, 'Emp1', 1000.00);
```

The following code enables the table for change data capture:

```
EXECUTE sys.sp_cdc_enable_table
    @source_schema = N'dbo'
  , @source_name = N'Employees'
  , @role_name = N'cdc_Admin';
```

Because this is the first table in the database that is enabled for change data capture, this code also causes the capture process to start (two jobs will start: cdc.testdb_capture and cdc.testdb_cleanup). The *@role_name* argument enables you to assign a database role that will be granted access to the change data. If the specified role does not exist, SQL Server creates it. By default, changes are captured for all table columns. To capture changes only in a subset of columns, you can specify the list of columns in the *@captured_column_list* argument. To get metadata information about the columns included in the capture instance, you can use the **sys.sp_cdc_get_captured_columns** procedure.

If you later want to disable change data capture for a table, use the **sys.sp_cdc_disable_table stored** procedure. And to check whether a table is enabled for change data capture, query the **is_tracked_by_cdc** column in the **sys.tables** view.

To get information about the change data capture configuration for each enabled table, use the stored procedure **sys.sp_cdc_help_change_data_capture** as follows:

```
EXECUTE sys.sp_cdc_help_change_data_capture;
```

The changes are not consumed directly from the change tables but rather through table-valued functions. You can easily identify and consume only the delta of changes that was not yet consumed. Data is requested for changes that lie within a specified range of log serial numbers (LSNs). SQL Server 2008 also provides the mapping functions **sys.fn_cdc_map_time_to_lsn** and **sys.fn_cdc_map_lsn_to_time** that help you convert a date-time range to a range of LSNs and vice versa.

Separate functions provide all changes within an interval (*cdc.fn_cdc_get_all_changes_<capture_instance>*) and net changes that took place against distinct rows (*cdc.fn_cdc_get_net_changes_<capture_instance>*), if that option was enabled by the **sys.sp_cdc_enable_table** procedure.

SQL Server 2008 also has a stored procedure named **sys.sp_cdc_get_ddl_history** that gives you the DDL change history associated with a specified capture instance.

Collation Alignment with Windows

SQL Server 2008 aligns support for collations with Microsoft Windows Server® 2008, Windows Vista®, Windows Server 2003, and Windows® XP Home Edition. The new SQL Server version adds new collations and revises existing collation versions. You can recognize the new and revised collations by finding the number 100 (the internal version number of SQL Server 2008) in their names. You can use the following query to get the list of new and revised collations:

```
SELECT name, description

FROM sys.fn_helpcollations() AS C

WHERE name LIKE '%100%';
```

This query returns more than 1,400 rows representing different variations of 84 new collations.

The new collations introduce significant changes. Many of them address East Asian languages that support supplementary characters. Chinese_Taiwan_Stroke_100 and Chinese_Taiwan_Bopomofo_100 now assign culture-correct weight for each character, specifically the Ext. A + B characters.

The new collations also provide string comparisons between supplementary characters based on linguistic sorting. A binary flag is added for true code point comparisons: binary-code point.

Note that some collations supported in SQL Server 2005 are deprecated in SQL Server 2008, including the Korean, Hindi, Macedonian, and Lithuanian_Classic Windows collations and the SQL_ALTDiction_CP1253_CS_AS SQL collation. These collations are still supported in SQL Server 2008 for backward compatibility, but they do not show up in the Setup collation list and are not retuned when querying the **fn_helpcollations** function.

Deprecated Features

SQL Server 2008 handles deprecation policy more seriously than previous versions. SQL Server 2008 discontinues support for the following features (partial list):

- 60, 65, and 70 compatibility levels—at a minimum, the database must be at compatibility level 80 (2000)
- DUMP and LOAD commands—use the BACKUP and RESTORE commands instead
- BACKUP LOG WITH TRUNCATE_ONLY, BACKUP_LOG WITH NO_LOG, and BACKUP TRANSACTION statements
- The **sp_addalias** stored procedure—replace aliases with user accounts and database roles
- **sp_addgroup**, **sp_changegroup**, **sp_dropgroup**, and **sp_helpgroup** stored procedures—use roles instead
- The Surface Area Configuration Tool
- Window calculations (using the OVER clause) are not allowed in the recursive member of recursive common table expressions (CTEs)

SQL Server provides performance counters and trace events to keep track of deprecated feature usage. The **SQLServer:Deprecated Features** object provides the Usage counter and an instance for each deprecated feature. As for trace events, the **Deprecation Final Support** event class occurs when you use a feature that will be removed from the next version of SQL Server. The **Deprecation Announcement** event class occurs when you use a feature that will be removed from a future version of SQL Server.

For the full list of discontinued features as well as a list of features planned for deprecation in the next and future versions of SQL Server, see SQL Server Books Online.

Conclusion

This paper introduces Transact-SQL enhancements and SQL/CLR improvements in SQL Server 2008. It covers Transact-SQL delighters, data type enhancements, Data Manipulation Language (DML) enhancements, Data Definition Language (DDL) enhancements, SQL/common language runtime (CLR) enhancements, object dependencies, change data capture, collation alignment with Windows, and deprecated features.

The new and enhanced features in SQL Server 2008 help you improve the performance of your database, enhance database support beyond the relational realm, and gain more functionality—all of which make you better equipped to support your enterprise data platform needs.

About the Author

Itzik Ben-Gan is a Mentor and Co-Founder of Solid Quality Mentors. A SQL Server Microsoft Most Valuable Professional (MVP) since 1999, Itzik has delivered numerous training events around the world focused on Transact-SQL querying, query tuning, and programming. Itzik is the author of *Inside Microsoft SQL Server 2005: T-SQL Querying* (MSPress) and *Inside Microsoft SQL Server 2005: T-SQL Programming* (MSPress) and is a co-author of *Advanced Transact-SQL for SQL Server 2000* (APress). He has written many articles for *SQL Server Magazine* as well as articles and

whitepapers for MSDN. Itzik's speaking activities include Tech Ed, DevWeek, PASS, and SQL Server Magazine Connections as well as various user groups and other engagements around the world.

For more information:

Microsoft SQL Server Web site [http://www.microsoft.com/sqlserver/default.mspx]

Microsoft SQL Server TechCenter [http://technet.microsoft.com/en-us/sqlserver/default.aspx]

Microsoft SQL Server DevCenter [http://msdn2.microsoft.com/en-us/sqlserver/default.aspx]

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

- Are you rating it high due to having good examples, excellent screenshots, clear writing, or another reason?
- Are you rating it low due to poor examples, fuzzy screenshots, unclear writing?

This feedback will help us improve the quality of white papers we release. Send feedback.

Tags:

Community Content