New T-SQL Features in SQL Server 2005 Part 2

Overview of the New features in Transact SQL 2005

I have been working on SQL Server for the last 6 years beginning with SQL Server 6.5 till SQL Server 2000 and now on the latest release SQL Server 2005. There have been a number of advancements with every release of SQL server over the last few years. For an Overview I have enlisted the few major changes that were introduced in each release of SQL Server.

The following were the advancements in version SQL 7.0 from that of SQL 6.5:

- New data types were added (nchar, nvarchar, ntext, uniqueidentifier)
- Increase in the max size of the data type from 255 bytes to 8000 bytes
- Full row level locking, autogrow features for disk and memory, Merge joins and Hash joins were introduced.
- Integrated replication, including multi-site update, for maintaining dependent data marts.

The following were the advancements in version SQL 2000 from that of SQL 7.0:

- New data types were added (bigint, sql_variant, table)
- Instead of and for Triggers were introduced as advancement to the DDL.
- Cascading referential integrity.
- XML support
- User defined functions and partition views.
- Indexed Views (Allowing index on views with computed columns).

The following are the advancements in version 2005 from that of SQL 2000:

- T-SQL Enhancements, a number of new features that includes Pivot/Unpivot etc.
- Common Language Runtime (Integration of .NET languages to build objects like stored procedures, triggers, functions etc.)
- Service Broker (Handling message between a sender and receiver in a loosely coupled manner)
- Data Encryption (Native capabilities to support encryption of data stored in user defined databases)
- SMTP mail
- HTTP endpoints (Creation of endpoints using simple T-SQL statement exposing an object to be accessed over the internet)
- Multiple Active Result Sets (MARS).This allows a persistent database connection from a single client to have more than one active request per connection.
- SQL Server Integration Services (Will be used as a primary ETL (Extraction, Transformation and Loading) Tool
- Surface Area configuration
- Enhancements in Analysis Services and Reporting Services.

It is evident from the comparison; there is a huge new list of features that is included in SQL 2005. In this article I will focus only on the major enhancements that have been done in T-SQL in SQL Server 20005.In the future articles I will cover the other major enhancements that are been done.

SQL Server 2005's T-SQL enhancements make writing certain types of queries significantly easier than in SQL Server 2000. While the new additions to 2005's T-SQL syntax could be replicated either in SQL Server 2000 or in the application layer, with 2005 the syntax is much more terse, readable, and sensible.

In my last article I loooked at some of the new T-SQL features and this one continues with a few more.

Cross APPLY and Outer Apply Operator:

With the APPLY operator, SQL Server 2005 allows you to refer to a table-valued function in a correlated sub query. The APPLY relational operator allows you to invoke a specified table-valued function once for each row of an outer table expression. You specify APPLY in the FROM clause of a query, similar to the way you use the JOIN relational operator.

APPLY comes in the following two forms:

CROSS APPLY: CROSS APPLY works like an INNER JOIN in that unmatched rows between the left table and the table-valued function dont appear in the result set.

OUTER APPLY: OUTER APPLY is like an OUTER JOIN, in that non-matched rows are still returned in the result set with NULL values in the function results. OUTER APPLY is very similar to CROSS APPLY, with the addition that it also returns rows from the outer table for which the table-valued function returned an empty set. Nulls are returned as the column values that correspond to the columns of the table-valued function.

The following script will demonstrate the usage of the Apply Operators:

```
/********* Script for Creation of Master and Detail Table *************************
Create table dbo.tblModule
(
 Moduleid int,
 Modulename varchar (20)
)
On [Primary]

Create table dbo.tblSection
(
 Sectionid int,
 Sectionname varchar (20),
 fk_Moduleid int
)
On [Primary]
/*************************************************************************************

/********* Inserting Data in the Tables ***********************************
Insert into dbo.tblModule (Moduleid, Modulename) values (1,'ARS')
Insert into dbo.tblModule (Moduleid, Modulename) values (2,'DEF')
Insert into dbo.tblModule (Moduleid, Modulename) values (3,'GMPT')
Insert into dbo.tblModule (Moduleid, Modulename) values (4,'TEST')
Insert into dbo.tblSection (Sectionid, Sectionname, fk_Moduleid) values (1,'Document
Insert into dbo.tblSection (Sectionid,Sectionname,fk_Moduleid) values (2,'Sampling',
```

```
Insert into dbo.tblSection (Sectionid,Sectionname,fk_Moduleid) values (3,'SalesFloor
Insert into dbo.tblSection (Sectionid, Sectionname, fk_Moduleid) values (4,'Document
Insert into dbo.tblSection (Sectionid,Sectionname,fk_Moduleid) values (5,'Sampling',
Insert into dbo.tblSection (Sectionid,Sectionname,fk_Moduleid) values (6,'SalesFloor
Insert into dbo.tblSection (Sectionid,Sectionname,fk_Moduleid) values (6,'TestSectio
/*************************************************************************
```

The above script creates table Module and Section and inserts data into both the tables. Next, we will create a table valued function on the section table and retrieve matching rows for the modules in section table.

Below is the script for creation of the table valued function.

```
/*****************************Function Creation *********************************
Create function dbo.ModuleSection (@ModuleID Int) Returns Table
as
Return
select sectionid,sectionname from dbo.tblsection where fk_moduleid=@ModuleID
/*************************************************************************************
```

Now we will use the Apply operator on the Table and the Function to see how different values are returned upon application of the CROSS and OUTER APPLY operator.

```
/*****************************Using the Cross Apply Operator *********************
Select * from dbo.tblmodule cross apply dbo.ModuleSection (moduleid) order by 1
/*************************************************************************************
```

Table 1.11 displays the result set obtained after executing the query with Cross apply operator. As is evident, the result set only has matching rows returned.

ModuleID	ModuleName	SectionID	SectionName
1	ARS	1	Documentation
1	ARS	6	SalesFloor
2	DEF	2	Sampling
2	DEF	4	Documentation
3	GMPT	5	Sampling
3	GMPT	3	SalesFloor

Table 1.11: Results using the CROSS Apply Operator

Below is the script that uses the Outer Apply operator for the same set of data.

```
/*****************************Using the Cross Apply Operator *********************
Select * from dbo.tblmodule Outer apply dbo.ModuleSection(moduleid) order by 1
/*************************************************************************************
```

Table 1.12 displays the result set obtained after executing the query with the Outer apply operator.

ModuleID	ModuleName	SectionID	SectionName

1	ARS	1	Documentation
1	ARS	6	SalesFloor
2	DEF	2	Sampling
2	DEF	4	Documentation
3	GMPT	5	Sampling
3	GMPT	3	SalesFloor
4	**TEST**	**NULL**	**NULL**

Table 1.12: Results using the OUTER Apply Operator

The result set not only contains all the matching rows about so also includes all the rows from the left query with NULL value from the table valued function.

TOP Clause and Ranking Functions:

There are number of ranking functions that are introduced in SQL 2005.Also there has been an enhancement in the TOP clause thats been available from SQL 2000. The following is the list of new features for ranking introduced in SQL 2005.

TOP: The TOP keyword allows you to return the first n number of rows from a query based on the number of rows or percentage of rows that you define. The first rows returned are also impacted by how your query is ordered. Syntax for the TOP clause:

```
Select TOP @Noofrows from Table
```

@Noofrows: Is the no of records that are to be returned from the Table.

```
Select TOP @Noofrows PERCENT from Table
```

@Noofrows: Is the Percentage of records that are to be returned from the table.

Note: The TOP clause can be used with the insert, update and delete to limit the number of records acted upon during the data manipulation operation.

Ranking Functions:

Ranking functions return a ranking value for each row in a partition. Depending on the function that is used, some rows might receive the same value as other rows. Ranking functions are non-deterministic.

Table 1.13 summarizes the Ranking functions that are introduced in SQL Server 2005.

Function Name	Description
ROW_NUMBER	Returns an incrementing no for each row in the result set
RANK	Returns the rank of each row within the partition of a result set.
DENSE_RANK	Dense_Rank is identical to RANK functions, with the difference been that it doesnt return gap in the rank values

NTILE	NTILE divides the result set into a specified number of groups, based on the ordering and optional partition.

Table 1.13: Ranking Functions in SQL Server 2005

The Example below will be used to demonstrate the different RANKING functions:

```
/******* Script for Creation & Inserting data**************************************
Create table dbo.tblStore
(
 StoreID varchar(4) ,
 StoreName varchar(50)
)
on [Primary]

Insert into dbo.tblStore (StoreID,StoreName) values ('0770','MarbleArch')
Insert into dbo.tblStore (StoreID,StoreName) values ('2655','Leeds')
Insert into dbo.tblStore (StoreID,StoreName) values ( '1740','Cambridge')
Insert into dbo.tblStore (StoreID,StoreName) values ('2587','Aberdeen')
Insert into dbo.tblStore (StoreID,StoreName) values ('2600','Oxford')
Insert into dbo.tblStore (StoreID,StoreName) values ('2600','Ballymena')
/*********************************************************************************/
```

ROW_NUMBER () : Returns the sequential number of a row within a partition of a result set, starting at 1 for the first row in each partition. Syntax for using the ROW_NUMBER ()

```
ROW_NUMBER ( )     OVER ([<partition_by_clause>] <order_by_clause>)
```

<Partition_by_clause>: (Optional) Divides the result set produced by the FROM clause into partitions to which the function is applied. This is only applicable for tables that have partitions associated to it.

<order_by_clause>: The order in which the row value is assigned to the rows in a partition. The order by clause is mandatory whilst using the Row_number function.

```
/*** Script to use the ROW NUMBER Function*************************************
Select ROW_NUMBER () OVER (ORDER BY StoreID) AS SrNo, StoreID ,StoreName
From tblStore;
/*********************************************************************************
```

Table 1.14 displays the results that is been obtained after executing the above script.

SrNo	StoreID	StoreName
1	0770	MarbleArch
2	1740	Cambridge
3	2587	Aberdeen
4	2600	Oxford
5	2600	Ballymena
6	2655	Leeds

Table 1.14: Results the ROW_NUMBER Function.

Note: The incremental row number sequence is based on the Order by Clause. Row Number function can be used with User defined function and table valued functions.

RANK (): The rank of a row is one plus the number of ranks that come before the row in question.

Rank function is similar to the ROW Number function, the key difference between them been that, rows with tied value will receive the same Rank Value. Syntax for using the RANK ()

```
RANK ( )    OVER ([< partition_by_clause >] < order_by_clause >)
```

The Script below applies Rank function to the store table created above.

```
/*** Script to use the RANK Function*********************************************/
Select RANK () OVER (ORDER BY StoreID) AS SrNo, StoreID ,StoreName
From tblStore;
/*******************************************************************************
```

The RANK function did not create a new incrementing value for the Oxford store as it has the same value for that of Leeds. Unlike ROW Number which provided incrementing values to both the stores.

Table 1.15 displays the result obtained after running the RANK script.

SrNo	StoreID	StoreName
1	0770	MarbleArch
2	1740	Cambridge
3	2587	Aberdeen
4	**2600**	**Ballymena**
4	**2600**	Oxford
6	2655	Leeds

Table 1.15: Results the RANK Function.

DENSE_RANK (): This is another Ranking Function which too is similar the aforementioned functions, the difference between RANK and DENSE_RANK been that it returns results without gaps in the rank values. Syntax for using the DENSE_RANK ()

```
DENSE_RANK ( )    OVER ([< partition_by_clause >] < order_by_clause >)
```

The Script below applies DENSE_Rank function to the store table created above.

```
/*** Script to use the DENSE_RANK Function***************************************
Select DENSE_RANK () OVER (ORDER BY StoreID) AS SrNo, StoreID ,StoreName
From tblStore;
/*******************************************************************************
```

Table 1.16 displays the result obtained after running the DENSE_RANK script.

SrNo	StoreID	StoreName

1	0770	MarbleArch
2	1740	Cambridge
3	2587	Aberdeen
4	2600	Ballymena
4	2600	Oxford
5	**2655**	Leeds

Table 1.16: Results the DENSE_RANK Function.

DENSE_RANK function did not create a new incrementing value for the Oxford store as it has the same value for that of Leeds, similar to the results of RANK Function. However the rank value for Leeds store is **5** above, thats without the gap which was there in the RANK function results.

NTILE: NTILE divides the result set into a specified number of groups based on the ordering and optional partition. The syntax is very similar to the other ranking functions, only it also includes an integer expression. Syntax for using the NTILE ()

```
NTILE (integer_expression)    OVER ( [ <partition_by_clause> ] < order_by_clause > )
```

integer_expression: It is used to determine the number of groups to divide the results into.

The Script below demonstrates usage of NTILE Function on Store table created and dividing the results into group of 3.

```
/*** Script to use the NTILE Function*********************************************
SELECT NTILE(3) OVER (ORDER BY StoreID) as SrNo,* FROM dbo.tblstore;
/*******************************************************************************
```

In the example the result set was divided into three Percentile groups. The total result set has six rows; each group has two rows each. Table 1.17 displays the result obtained after executing the NTILE script.

SrNo	StoreID	StoreName
1	0770	MarbleArch
1	1740	Cambridge
2	2587	Aberdeen
2	2600	Oxford
3	2600	Ballymena
3	2655	Leeds

Table 1.17: Results the NTILE Function.

Output Clause:

The Output Clause returns information from, or expressions based on, each row affected by an INSERT, UPDATE, or DELETE statement. The results can be used for audit purpose by inserting the data into a table. The Syntax for using the OUTPUT Clause ()

```
<OUTPUT_CLAUSE>:= {     [OUTPUT < {DELETED | INSERTED | from_table_name}. {* | colum
```

The arguments of the Output clause are described in Table 1.18

Argument	Description
@table_variable	Specifies a **table** variable that the returned rows are inserted into instead of being returned to the caller. *@table_variable* must be declared before the INSERT, UPDATE, or DELETE statement. If *column_list* is not specified, the **table** variable must have the same number of columns as the OUTPUT result set.
output_table	Specifies a table that the returned rows are inserted into instead of being returned to the caller. *output_table* may be a temporary table. If *column_list* is not specified, the **table** variable must have the same number of columns as the OUTPUT result set.
Column_list	Is an optional list of column names on the target table of the INTO clause
DELETED	Is a column prefix that specifies the value deleted by the update or delete operation. Columns prefixed with DELETED reflect the value before the UPDATE or DELETE statement is completed. DELETED cannot be used with the OUTPUT clause in the INSERT statement.
INSERTED	Is a column prefix that specifies the value added by the insert or update operation. Columns prefixed with INSERTED reflect the value after the UPDATE or INSERT statement is completed but before triggers are executed. INSERTED cannot be used with the OUTPUT clause in the DELETE statement.
from_table_name	Is a column prefix that specifies a table included in the FROM clause of a DELETE or UPDATE statement that is used to specify the rows to update or delete

Table 1.18: Arguments for the Output Clause

The following example demonstrates the use of OUTPUT clause in INSERT/UPDATE and DELETES scenarios. For the Output clause examples we will create the Stores Table, insert few rows in the table and perform Data manipulation operations on them.

```
/******* Script for Creation & Inserting data*************************************
Create table dbo.tblStore
(
 StoreID varchar(4) ,
 StoreName varchar(50)
)
on [Primary]

Insert into dbo.tblStore (StoreID,StoreName) values ('0770','MarbleArch')
Insert into dbo.tblStore (StoreID,StoreName) values ('2655','Leeds')
Insert into dbo.tblStore (StoreID,StoreName) values ('1740','Cambridge')
/*********************************************************************************/
```

First and foremost we will update a row in the stores table and use OUTPUT to return information on

the original and the updated column names.

```
/*************Script for Creating Audit Table ***************/
Create Table StoreAudit
(
DeletedStore nvarchar(50), InsertedStore nvarchar(50)
)

UPDATE tblstore
SET StoreName = 'Grantham'
OUTPUT DELETED.StoreName,
 INSERTED.StoreName
INTO StoreAudit
WHERE storeName = 'Oxford'

Select * from StoreAudit
/************************************/
```

After executing Select on the Audit table following results is retrieved. Table 1.19 displays the data that is available in the Audit table after using the OUTPUT clause.

DeletedStore	InsertedStore
Grantham	Oxford

Table 1.19: Data in the Audit table

The results display both the Deleted and the Inserted store for the update operation.

The following example demonstrates the usage of Output for deletions done to the Store table.

```
/*************Script for Creating Audit Table *************************/
DECLARE @StoresDeleted TABLE(
Storeid varchar(4),
StoreName varchar(50),
ModifiedDate datetime NOT NULL default getdate() )

DELETE tblstore
OUTPUT DELETED.Storeid,DELETED.StoreName
INTO @StoresDeleted(Storeid,StoreName)

Select * from @StoresDeleted
/*****************************************************************/
```

Table 1.20 displays the data that is deleted from the store table and is stored in the audit table using the OUTPUT clause.

StoreID	StoreName	ModifiedDate
0770	MarbleArch	2006-11-09 18:11:16.753
2655	Leeds	2006-11-09 18:11:16.753
1740	Cambridge	2006-11-09 18:11:16.753
2587	Aberdeen	2006-11-09 18:11:16.753

2600	Oxford	2006-11-09 18:11:16.753
2600	Ballymena	2006-11-09 18:11:16.753

Table 1.20: Data deleted from stores table

The following example demonstrates the usage of Output for Insertions done to the Store table.

```
/*************Script for Creating Audit Table *************************/
DECLARE @StoresInserted TABLE(
        Storeid varchar(4),
        StoreName varchar(50)
)

INSERTtblstore (StoreID,StoreName)
OUTPUT INSERTED.*
INTO @StoresInserted
VALUES ('0343','MeadowHall')

Select * from @StoresInserted
/*********************************************************************/
```

Table 1.21 displays the new row inserted in the Store table is captured by the Output and in turn stored in a Memory Table.

StoreID	StoreName
0343	MeadowHall

Table 1.21: Row inserted into stores table

Note: If the OUTPUT clause is specified without also specifying the INTO keyword, the target of the DML operation cannot have any enabled trigger defined on it for the given DML action. For example if the Output clause is defined in an output statement, then the Output table cannot have any enabled Update Triggers.

Error Handling (TRY/CATCH):

In SQL Server 2005, the TRY...CATCH command can be used to capture execution errors within your Transact-SQL code. TRY...CATCH can catch any execution error with a severity level greater than 10 (so long as the raised error doesnt forcefully terminate the Transact-SQL user session). TRY...CATCH can also handle severity level errors (greater than 10) invoked using RAISERROR. The syntax for TRY...CATCH is as follows:

```
BEGIN TRY
 {sql_statement | statement_block}
END TRY
BEGIN CATCH
 {sql_statement | statement_block}
END CATCH
```

The arguments, used in both the TRY and CATCH sections are sql_statement and statement_block. In a nutshell, statements within the TRY block are those you wish to execute. If errors are raised within the

TRY block, then the CATCH block of code is executed. The CATCH block is then used to handle the error. Handling just means that you wish to take some action in response to the error: whether its to report the errors information, log information in an error table, or roll back an open transaction.

The benefit of TRY...CATCH is in the ability to nest error handling inside code blocks, allowing you to handle errors more gracefully and with less code than non-TRY...CATCH methods. TRY...CATCH also allows you to use new SQL Server 2005 error logging and transaction state functions which capture granular error information about an error event. Table 1.22 details the use of each of the function

Function	Description
ERROR_LINE	The error line number in the SQL statement or block where the error was raised.
ERROR_MESSAGE	The error message raised in the SQL statement or block.
ERROR_NUMBER	The error number raised in the SQL statement or block.
ERROR_PROCEDURE	Name of the trigger or stored procedure where the error was raised (assuming TRY...CATCH was used in a procedure or trigger).
ERROR_SEVERITY	The severity level of the error raised in the SQL statement or block.
ERROR_STATE	The state of the error raised in the SQL statement or block.
XACT_STATE	In the CATCH block, XACT_STATE reports on the state of open transactions from the TRY block. If 0 is returned, there are no open transactions from the TRY block. If 1 is returned, it means that no errors were raised in the TRY block. If -1 is returned, an error occurred in the TRY block, and the transaction must be rolled back. XACT_STATE can also be used outside of a TRYCATCH command.

Table 1.22: Error and transaction state functions of SQL Server 2005

If an error is encountered in a TRY batch, SQL Server will exit at the point of the error and move to the CATCH block, without processing any of the other statements in the TRY batch (the exception to the rule is if youre using nested TRY...CATCH blocks, which Ill demonstrate later on in the chapter).

TRY...CATCH can be used within a trigger or stored procedure, or used to encapsulate the actual execution of a stored procedure (capturing any errors that bubble up from the procedure execution and then handling them accordingly).

Warnings and most informational attention messages (severity level less than 10 or lower), are *not* caught by TRY...CATCH, and neither are syntax and object name resolution errors. Nonetheless, this new construct is now an ideal choice for capturing many other common error messages that in previous versions required bloated and inelegant Transact-SQL code.

Conclusion:

Microsoft SQL Server 2005 provides the tools that developers need to build new classes of database applications. By removing the barriers to code execution and storage location, and by integrating standards such as XML, SQL Server 2005 opens up a world of possibilities to the database developer.

The enhancement in TSQL increases your expressive powers in query writing, allows you to improve

the performance of your code, and extend your error management capabilities.

The article only covered few of the enhancements done in Transact SQL for SQL server 2005. In the next article I will cover the functions Partitioned table, function and Scheme, Event Notification & DDL triggers, Indexes with Included Columns, MARS-Multiple Active Result sets, .Net Assembly Support, Snapshot Isolation.