New T-SQL Features in SQL Server 2005 Part 1

New features in Transact SQL 2005 Part 1

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.

I am starting with Transact SQL new features as this is the most widely used by developers for building WEB based applications that use SQL 2005 as the back end for data Storage. The following are the new features included in Transact SQL for SQL Server 2005.

Large data Types & XML data type : SQL Server 2005 changes include a lot of new data types to store values using the MAX indicator; it has also introduced a new data type termed XML for storing XML data.

varchar (max) ,nvarchar(max) and varbinary(max) : SQL server 7.0 raised the limit of varchar fields to 8000 bytes However as many large string based fields exceeded 8000 bytes; data was store in a binary field (like TEXT, NTEXT or IMAGE) in the database.

One of the major drawbacks of the old text and image data types is that they required you to use separate functions such as WRITETEXT and UPDATETEXT in order to manipulate the image/text data. Using the new large value data types, you can now use regular INSERT and Updates instead. The syntax for inserting a large value data type is no different from a regular insert. For updating large value data types, however, the UPDATE command now includes the .WRITE method.

SQL SERVER 2005 has the new MAX specifier which can be used with the aforementioned data types store vast quantities of data. A variable declared as VARCHAR (MAX) or VARBINARY (MAX) can store 231 (or about 2 billion) characters. A variable declared as NVARCHAR (MAX) can store 230or about 1 billion) characters. In SQL Server 2005,there is no limitation on the row size (In SQL 2000 row size was limited to 8060 bytes) rows can now span multiple pages, so the 8060 limit is no longer applicable, and you can define several columns on a single table as MAX types, if needed.

XML : The XML data type is used for storing XML documents. Introduction of this data type not only makes storage of XML documents and fragments easier and cleaner than the previous version but also provides mechanism to interact with the XML data type.

The new functions available with the XML data type are described in Table 1.1

Function Name	**Function Description**
Query	Gets a set of nodes from an XML document/fragment (returns XML)
Value	Gets a single value from an element or attribute of an XML document/fragment (returns scalar value)
Exist	Returns a Boolean value indicating if the XQuery expression returns values
Modify	Changes values in an XML document/fragment
Nodes	Gets a context to a node based upon the XQuery expression

Table 1.1: Functions associated with XML data Type

The new function makes it relatively easier to insert/delete and update a node in the XML.

Common Table Expressions : Common table expression (CTE) can be thought of as a temporary result set that is defined within the execution scope of a single SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement. Unlike a derived query that was commonly used in SQL 2000, you dont need to copy the query definition multiple times each time it is used. You can also use local variables within a CTE definitionsomething you cant do in a view definition.

In a nutshell CTE can be used to perform the following:

- Create a recursive query.
- Substitute for a view when the general use of a view is not required; that is, you do not have to store the definition in metadata.
- Enable grouping by a column that is derived from a scalar subselect, or a function that is either not deterministic or has external access.
- Reference the resulting table multiple times in the same statement.

The basic syntax structure for a CTE is:

```
WITH expression_name [(column_name [,...n ] )
]
AS (CTE_query_definition)
```

The arguments of a CTE are described in Table 1.2

Argument	Description
Expression_name	The name of the Common Table Expression
column_name [,...n]	The Unique column names of the expression
CTE_query_definition	The Query that defines the common table expression

Table 1.2: Arguments for the CTE

For further reference on CTE read the link: http://msdn2.microsoft.com/en-gb/library/ms190766.aspx

Hierarchical(Recursive) queries:

A recursive CTE is one in which an initial CTE (Common Table Expression) is repeatedly executed to return subsets of data until the complete result set is obtained. The structure of the recursive CTE in Transact-SQL is similar to recursive routines in other programming languages. Although a recursive routine in other languages returns a scalar value, a recursive CTE can return multiple rows. A recursive CTE consists of three elements:

1. Invocation of the routine.
   The first invocation of the recursive CTE consists of one or more *CTE_query_definitions* joined by UNION ALL, UNION, EXCEPT, or INTERSECT operators. Because these query definitions form the base result set of the CTE structure, they are referred to as anchor members.
2. Recursive invocation of the routine.

The recursive invocation includes one or more *CTE_query_definitions* joined by UNION ALL operators that reference the CTE itself. These query definitions are referred to as recursive members.

3. Termination check.
   The termination check is implicit; recursion stops when no rows are returned from the previous invocation

The basic syntax structure for a Recursive query is:

```
WITH cte_name (column_name [,...n] )
AS
(
CTE_query_definition – Invocation.
UNION ALL
CTE_query_definition – Recursive Invocation using the CTE Name
)
-- Statement using the CTE
SELECT *
FROM cte_name OPTION (Maxrecursion n)
```

Note: As a best practice, set the MAXRECURSION based on your understanding of the data. If you know that the hierarchy cannot go more than 10 levels deep, for example, then set **MAXRECURSION** to that value. This link below mentioned below has an illustrated example of the usage of Hierarchical queries: http://www.theserverside.net/tt/articles/showarticle.tss?id=HeirarchicalQueries

Intersect and Except Operands:

Transact SQL has a number of operations (Like Union, Joins) that are used for obtaining results from two or more result sets. SQL Server 2005 introduces the INTERSECT and EXCEPT operands.

INTERSECT operand is useful for retrieving only distinct rows that exist in both the left and right queries. EXCEPT operand is useful for retrieving only distinct rows that exist in the left query.

Example of using the Intersect and except Operand

In the example I will be creating two tables tblProductA and tblProductB for displaying the results that are obtained using the EXCEPT and INTERCEPT Operands.

```
/*** Script for creation of Table******************/
Create table DBO.tblProductA
(
 ProductID int identity,
 Description varchar (50)
)

Create table DBO.tblProductB
(
 ProductID int identity,
 Description varchar (50)
)
/*********************************************/

/*** Inserting Data in both the Tables******************/
Insert into DBO.tblProductA (Description) values ('Glass')
```

```
Insert into DBO.tblProductA (Description) values ('Bottles')
Insert into DBO.tblProductA (Description) values ('Plates')
Insert into tblProductB (Description) values ('Glass')
Insert into tblProductB (Description) values ('Bottles')
Insert into tblProductB (Description) values ('Jugs')
Insert into tblProductB (Description) values ('Scraps')

/************************************************/
```

The above script creates tables tblproductA and tblproductB and inserts different data in both the tables. Table 1.3 displays the data that is stored Table tblproductA.

ProductID	Description
1	Glass
2	Bottles
3	Plates

Table 1.3: Data available in Table tblProductA

Table 1.4 displays the data that is stored Table tblproductB.

ProductID	Description
1	Glass
2	Bottles
3	Jugs
4	Scraps

Table 1.4: Data available in Table tblProductB

The script below applies the intersect operand to the tables ProductA and ProductB.

```
/*****************Script to apply the INTERSECT Operand ************************/
Select ProductID,Description from tblProductA
Intersect
Select ProductID,Description from tblProductB
/************************************************** ************************/
```

Upon executing the above query, only distinct records from both the table will be returned as shown in Table 1.5.

INTRCEPT operand has only returned the distinct rows available in both the tables (i.e. Glass & Bottles)

ProductID	Description
1	Glass
2	Bottles

Table 1.5: Results using the INTERSECT operand

The script below applies the intersect operand to the tables ProductA and ProductB.

```
/*****************Script to apply the except Operand ************************/
Select ProductID,Description from tblProductA
Except
Select ProductID,Description from tblProductB
/*************************************************************************/
```

Upon execution of the above query using the EXCEPT operand only distinct rows that are not available in the left table will be returned.

Table 1.6 displays the records that will be returned after execution of the query.

ProductID	Description
3	Plates

Table 1.6: Results using the EXCEPT operand

Note: EXCEPT and INTERSECT cannot be used in distributed partitioned view definitions, query notifications or together with COMPUTE and COMPUTE by clauses. Usage of these operands in distributed queries on the local server may affect performance. If the operands are used in cursors (Dynamic & Keyset) they will be converted to static cursor.

Synonym:

Microsoft has introduced Synonym with the release of SQL server 2005. SYNONYM is a single-part name that can replace a two, three or four part name in many SQL statements. SYNONYMS can be created on the following objects:

- Table
- View
- Assembly Stored Procedures, Table Valued Functions, Aggregations
- SQL Scalar Function
- SQL Stored Procedure
- SQL Table Valued Functions
- SQL Inline-Table-Valued Function
- Local and Global Temporary Tables
- Replication-filter-procedure

The following example first creates a synonym for the base object, Product in the AdventureWorks database, and then queries the synonym.

```
/*****************Script to create a Synonym ************************/
Use AdventureWorks
Create SYNONYM Product
For AdventureWorks.Production.Product;
/************************************************** ************************/
```

Once a Synonym is created, data can be queried from the Synonym similar to as it is done from the base tables.

```
/*****************Script to query a Synonym ************************/
Use AdventureWorks
Select * from Product where productID<100
/************************************************** ************************/
```

Note: To create a synonym in a given schema, a user must have CREATE SYNONYM permission and either owns the schema or have ALTER SCHEMA permission. Synonyms can be used with dynamic SQL.

PIVOT and UNPIVOT Function:

This is one of the most important enhancements to TSQL in SQL 2005 and is extremely useful for generating reports based on cross tab values. The PIVOT and UNPIVOT relational operators are used in the FROM clause of the query. The PIVOT operator rotates rows into columns, optionally performing aggregations or other mathematical calculations along the way.

PIVOT FUNCTION:

It widens the input table expression based on a given pivot column and generates an output table with a column for each unique value in the pivot column. The PIVOT operator is useful for handling open-schema scenarios and for generating cross tab reports.

The syntax for the PIVOT relational operator is:

```
FROMtable_source
PIVOT ( aggregate_function ( value_column )
 FOR pivot_column
IN ( <column_list>)
 ) table_alias
```

The arguments of a pivot are described in Table 1.2

Argument	Description
table_source	The table where the data will be pivoted.
aggregate_function (value_column)	The aggregate function that will be used against the specified column.
pivot_column	The column that will be used to create the column headers.
column_list	The values to pivot from the pivot column.

Table 1.7: Arguments for the pivot Function

To demonstrate the example of PIVOT .We will first construct a Department and Employee table.

```
/************Script for constructing the PIVOT base Tables************************
Create table dbo.Employeedetails
(
 Employeeid int,
```

```
 EmployeeName Varchar(50),
 Deptid int
)
On [Primary]
Create table dbo.Department
(
 Deptid int,
 DepartmentName Varchar(50),
)
On [Primary]
/*********************************************************************************
```

We will be inserting data into the Tables so that we can apply the PIVOT function and yield the results.

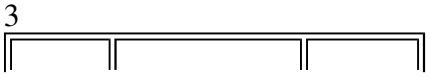
```
Insert into Employeedetails (Employeeid, EmployeeName, Deptid)
Values (1,'Joe', 1)
Insert into Employeedetails (Employeeid, EmployeeName, Deptid)
Values (2,'Mark', 1)
Insert into Employeedetails (Employeeid, EmployeeName, Deptid)
Values (3,'Andy', 1)
Insert into Employeedetails (Employeeid, EmployeeName, Deptid)
Values (4,'Claire', 2)
Insert into Employeedetails (Employeeid, EmployeeName, Deptid)
Values (5,'Chris', 2)
Insert into Employeedetails (Employeeid, EmployeeName, Deptid)
Values (6,'James', 2)
Insert into Employeedetails (Employeeid, EmployeeName, Deptid)
Values (7,'Jane', 3)
Insert into Department (Deptid, DepartmentName)
Values (1,'Design')
Insert into Department (Deptid, DepartmentName)
Values (2,'Management')
Insert into Department (Deptid, DepartmentName)
Values (3,'Analyst')
/*************************************************************************/
```

The scenario represents Employees in an Organisation, with the details of the department to which they belong. Suppose you have a requirement for reporting wherein you want to know how many people work for a particular stream of business you will use the pivot function.

```
/**********Script for Using the PIVOT Function *********************************/
Select Design, Management, Analyst from
(Select Employeeid, DepartmentName from Department inner join
Employeedetails on Department.deptid=Employeedetails.deptid) as A
PIVOT
(
COUNT(EmployeeID)
FOR DepartmentName IN ([Design], [Management],[Analyst]))
AS B
/*************************************************************************/
```

The Following result set is obtained after executing the above query, which PIVOTS data based on the department thats been selected. Table 1.8 displays the results obtained after executing the PIVOT function.

3



Design	Management	Analyst
3	1	

Table 1.8: Result after executing the PIVOT Function.

The results took the three columns fixed in the FOR part of the PIVOT operation and aggregated counts of employees by DepartmentName. The important point to note here is that list of column names cannot already exist in the query results being pivoted.

UNPIVOT FUNCTION:

The UNPIVOT operator performs an operation opposite to the one PIVOT performs; it rotates columns into rows. The UNPIVOT operator narrows the input table expression based on a pivot column. The syntax for Unpivot function is identical to the PIVOT syntax, the only difference been that its designated with UNPIVOT instead of PIVOT

For demonstrating the UNPIVOT function we will first construct a d-normalised table named EmployeeContact which will hold the information of the employee and this telephone details.

```
/*************Script for creation of EmployeeContact Table and inserting data in it
CREATE TABLE dbo.EmployeeContact
(   EmployeeID int NOT NULL,
 EmployeeName varchar(50),
 TelephoneNo bigint,
 Mobileno bigint,
 FaxNo bigint)
GO
INSERT dbo.EmployeeContact (EmployeeID,EmployeeName, TelephoneNo, Mobileno, FaxNo)
VALUES(1,'Joe', 2718353881, 3385531980, 5324571342)
INSERT dbo.EmployeeContact (EmployeeID,EmployeeName, TelephoneNo, Mobileno, FaxNo)
VALUES(2,'Mark', 6007163571, 6875099415, 7756620787)
INSERT dbo.EmployeeContact (EmployeeID,EmployeeName, TelephoneNo, Mobileno, FaxNo)
VALUES(3,'Bill', 9439250939, NULL, NULL)

Select * from dbo.employeecontact
/*************************************************************************************
```

The above script creates a table and inserts data into it. The table below shows the data stored in employeecontact table. Table 1.9 displays the data that is available in table EmployeeContact

EmployeeID	EmployeeName	TelephoneNo	MobileNo	FaxNo
1	Joe	2718353881	3385531980	5324571342
2	Mark	6007163571	6875099415	7756620787
3	Bill	9439250939	NULL	NULL

Table 1.9: Data stored in table EmployeeContact

Now using the UNPIVOT, we will convert the phone numbers into a normalized form. We will be using a single Phoneno field instead of repeating the Phone column multiple times.

```
/***************Script for excepting the UNPIVOT Function *************************
SELECT EmployeeName,
 PhoneCategory,
 PhoneNo
FROM
(SELECT EmployeeName, TelephoneNo, Mobileno, FaxNo
FROM dbo.EmployeeContact) EmpContact
UNPIVOT
 (PhoneNo FOR PhoneCategory IN ([TelephoneNo], [Mobileno], [FaxNo])
) AS Phone
/*******************************************************************************
```

Table 1.10 displays the results obtained after executing the above query using the UNPIVOT function.

EmployeeName	PhoneCategory	PhoneNo
Joe	TelephoneNo	2718353881
Joe	Mobileno	3385531980
Joe	FaxNo	5324571342
Mark	TelephoneNo	6007163571
Mark	Mobileno	6875099415
Mark	FaxNo	7756620787
Bill	TelephoneNo	9439250939

Table 1.10: Data stored in table EmployeeContact

This query returned the phone data merged into two columns, one to describe the PhoneCategory and another to hold the actual PhoneNo. Also notice that there are seven rows, instead of nine. This is because for EmployeeName Bill, only non-NULL values were returned. UNPIVOT does not process NULL values from the pivoted result set.

TABLESAMPLE Clause:

Introduced in SQL Server 2005, tablesample clause limits the number of rows returned from a table in the FROM clause to a Sample number or percent of rows. TABLESAMPLE clause can be beneficial in situations where only a sampling of rows is necessary for the application instead of full result set.

Syntax for using TABLESAMPLE Clause

```
TABLESAMPLE [SYSTEM] (sample_number [PERCENT | ROWS])
```

sample_number: The sample percentage of records that are to be returned.

Percent: Is the percentage of the tables data pages. Once the sample pages are selected, all rows for the selected pages are returned.

Example of using the TABLESAMPLE Clause:

```
Use AdventureWorks
```

```
Go
SELECT FirstName,LastName
FROM Person. Contact TABLESAMPLE SYSTEM (50 PERCENT)
```

Note: TABLESAMPLE cannot be specified in the definition of a view or an inline table-valued function.

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 a few of the enhancements in Transact SQL for SQL server 2005. In the next article I will cover more and then move on to more complicated features in future parts of this series.