**T-SQL Definitions**

**TOP TEN MISTAKES IN TSQL:**

* NULLs and the NOT IN predicate
* Functions on indexed columns in predicates
* Incorrect subquery column
* Data type mismatch in predicates
* Predicate evaluation order
* Outer joins and placement of predicates
* Subqueries that return more than one value
* Use of SELECT \*
* Scalar user-defined functions
* Overuse of cursors

GOTO: http://www.simple-talk.com/sql/t-sql-programming/ten-common-sql-programming-mistakes/

**Trigger:** SQL Server Supports triggers either after or instead of an insert, update, or delete operation. DDL triggers can be set on tables and views but only an INSTEAD OF trigger can be set on a view. DDL Triggers can fire in reaction to a wide range of events, including: Drop Table, Create Table, Alter Table, Logins ect.

A trigger is typically used to apply specialized business rules to ensure data integrity.

**DML Triggers are commonly used to:**

audit changes (e.g. keep a log of the users and roles involved in changes)

enhance changes (e.g. ensure that every change to a record is time-stamped by the server's clock)

enforce business rules (e.g. require that every invoice have at least one line item)

execute business rules (e.g. notify a manager every time an employee's bank account number changes)

replicate data (e.g. store a record of every change, to be shipped to another database later)

enhance performance (e.g. update the account balance after every detail transaction, for faster queries)

**The four main types of triggers are:**

Row Level Trigger: This gets executed before or after any column value of a row changes

Column Level Trigger: This gets executed before or after the specified column changes

For Each Row Type: This trigger gets executed once for each row of the result set caused by insert/update/delete

For Each Statement Type: This trigger gets executed only once for the entire result set, but fires each time the statement is executed.

**Triggers can be placed inside a stored procedure:**

CREATE PROCEDURE ps\_Orders\_INSERT

@Ord\_Priority varchar(10)

AS

BEGIN TRANSACTION

INSERT Orders (Ord\_Priority) VALUES (@Ord\_Priority)

IF @@ERROR <> 0

GOTO ErrorCode

IF @Ord\_Priority = 'High'

PRINT 'Email Code Goes Here'

COMMIT TRANSACTION

ErrorCode:

IF @@TRANCOUNT <> 0

PRINT 'Error Code'

Go

/\*

This trigger checks for existing sales orders using the product being deleted. Prevents deletion if orders exist.

\*/

CREATE TRIGGER tr\_DelProduct

ON Production.Product

FOR DELETE

AS

IF (SELECT COUNT (\*)

FROM Sales.SalesOrderDetail

INNER JOIN Deleted

ON SalesOrderDetail.ProductID = Deleted.ProductID) >0

BEGIN

RAISERROR ( ‘ Cannot delete a product with sales orders’, 14, 1)

ROLLBACK TRANSACTION

RETURN

END

**STORED PROCEDURE:**

A stored procedure allows for more modular programming. You can create reusable, discrete pieces of functionality using stored procedures. Stored procedures have the parsing and execution plan built at the time they are created so they execute faster than dynamic SQL. If you have a stored procedure that manipulates large amounts of data, that data will remain on the server and not be transported across the network. You can also use stored procedures for security. You can grant users execute permissions on stored procedures without granting them permissions on objects manipulated by those stored procedures.

**In a stored procedure instead of using with(nolock) hints on each table call, you would set the Isolation level to ‘READ uncommited’**

**DEFAULT ISOLATION LEVEL in SQL is “READ COMMITTED” but can be changed**

SET TRANSACTION ISOLATION LEVEL

{ READ UNCOMMITTED

| READ COMMITTED

| REPEATABLE READ

| SNAPSHOT

| SERIALIZABLE

}

[ ; ]

1. **READ UNCOMMITTED**(**Best for historical data**) **UserA will see the change made by UserB.** This isolation level is called dirty reads, which means that read data is not consistent with other parts of the table or the query, and may not yet have been committed. This isolation level ensures the quickest performance, as data is read directly from the table’s blocks with no further processing, verifications or any other validation. The process is quick and the data is asdirty as it can get.
2. **READ COMMITTED(Best for financial data)UserA will not see the change made by UserB.** This is because in the READ COMMITTED isolation level, the rows returned by a query are the rows that were committed when the **query** was started. The change made by UserB was not present when the **query** started, and therefore will not be included in the query result.
3. **REPEATABLE READ(Most Restrictive)** **UserA will not see the change made by UserB.** This is because in the REPEATABLE READ isolation level, the rows returned by a query are the rows that were committed when the **transaction** was started. The change made by UserB was not present when the **transaction** was started, and therefore will not be included in the query result.  
   This means that “**All consistent reads within the same transaction read the snapshot established by the first read**” (from MySQL documentation. See <http://dev.mysql.com/doc/refman/5.1/en/innodb-consistent-read.html>).
4. **SERIALIZABLE:** This isolation level specifies that all transactions occur in a completely isolated fashion, meaning as if all transactions in the system were executed serially, one after the other. The DBMS can execute two or more transactions at the same time only if the illusion of serial execution can be maintained.  
   In practice,**SERIALIZABLE is similar to REPEATABLE READ**, but uses a different implementation for each database engine. In Oracle, the REPEATABLE READ level is not supported and SERIALIZABLE provides the highest isolation level. This level is similar to REPEATABLE READ, but InnoDB implicitly converts all plain SELECT statements to “SELECT … LOCK IN SHARE MODE.

**Differences between Stored procedures and User defined functions**

**Difference 1:**

**Stored procedure will be used to perform specific tasks**   
  
The stored procedure normally is used to perform a specific task. The bulk of sql statement that will be complied and it uses the cached execution plans. It can return more than one result set.

**Normally functions will be used for computing value**

The functions are used to do the calculations instead of doing in the query. It can be used for many places if we want the same operation.       
 **Difference 2:**   
  
**Stored procedures may or may not return values**

The stored procedure based on query type it will do the operation. If we write any select query then it will return the results. If we do only update, insert or delete then it won’t return any results. However if you want to check the confirmation of the transaction then we can return the result. It is not compulsory to return the result set.

**But function should return value**

The function must return the value. Based on the function type it will return the results.

If we have written scalar function then it returns single value. If we have written table valued function then it returns multiple rows. We cannot write the function without return any value to the calling program.

**Difference 3:**

**Stored procedure cannot be used in the select/where/having clause**

The stored procedure cannot be called like the following.   
  
 SELECT \* FROM Pr\_RetrieveEmployees -- It will throws an error

It will throw an error. Similarly the stored procedure cannot be part the sql query any where.

**But function can be called from select/where/having clause**   
  
The function can be called using the select query.

It can be called from the select/where/having clause.   
  
For instance SELECT [dbo].fn\_EmployeeSalary (5) Ãƒ  it is scalar UDF. It returns single value.

                         SELECT \* FROM fn\_EmployeeHistory (3) Ãƒ  its will return multi value.

**Difference 4:**

**Stored procedure can run** independently**. It can be executed using EXECUTE or EXEC command**

The stored procedure can run independently. Once the stored procedure is compiled then it can be executed. It can be executed using the sql command statement EXECUTE or EXEC.

EXECUTE proc\_RetrieveEmployeeDetails EXEC proc\_RetrieveEmployeeDetails proc\_RetrieveEmployeeDetails

**But function cannot run independently**

The function cannot run independently. It has to be the part of the SQL statement.

**Difference 5:**

**Temporary table (derived) cannot be created on function.**   
  
The temporary table cannot be created in the function. As you know if you create a temp table then it will be stored on the tempdb database. But the temp table won't allow us to create with inside the function  
  
There are three ways to create the temp table.

       1. Create temp table

       2. Derived table

3. CTE common table expression starts with ‘With’

SELECT \* INTO #tmpEmployee FROM Employees  
  
The above statement is derived table. It cannot be created in a function.

**But it can be created in stored procedures**   
  
The stored procedure allows us to create the temp tables in the stored procedure.

**Difference 6:**

**From sql server 2005 onwards, TRY CATCH statements can be used in the stored procedures.**

The TRY CATCH is one of the new features in the SQL server 2005 edition. It can be used inside the stored procedure. As you know it handles the error in the catch block, whatever the statements written in the try block.

**But it cannot be used in the function. But we can use raise error in the function.**

The TRY CATCH block cannot be used with inside the functions. But we can use the raiserror function to throw the exception.

**Difference 7:**

**Stored procedure can call the user defined functions**

The function can be called from the stored procedure. 

CREATE PROC Pr\_RetirveCustomers AS BEGIN SET NOCOUNT ON SET XACT\_ABORT ON SELECT \* FROM Customers

SELECT \*

FROM [dbo].fn\_GetOrderedCustomers (5) END

**But the function cannot call the stored procedures.**

The function cannot call the stored procedures like procedures. There are many types of stored procedures in sql server.

* System Stored procedure
* User defined Stored procedure
* .NET CLR stored procedure
* Extended stored procedure

**Difference 8:**

**Stored procedures can have input and output parameters.**

As you know, the input and output are the parameters which can return the results through that variable. The output parameter can be only used to return the results through the output variable. But the input parameter can do both input and output operations.

**But the function can have only input parameters.**   
  
This won't allow us to use the output parameters. But we can use input parameter.

**Difference 9:**

**Stored procedures can have select and all DML operations.**

The stored procedures can do all the DML operations like insert the new record, update the records and delete the existing records.

**But the function can do only select operation.**

The function won't allow us to do the DML operations such as Update, Insert, Delete in the database tables like in the stored procedure. It allows us to do only the **Select** operation.

It will not allow to do the DML on existing tables. But still we can do the DML operation only on the table variable inside the user defined functions.

**Difference 10:**

**Function cannot have the transaction statements.**

The transaction statement cannot be used in the function such as COMMIT or ROLLBACK.

Also, normally we won't do any DML operations in the function such as INSERT, UPDATE, DELETE only the SELECT.

**Stored procedure can use transaction statements.**

The transaction statement can be used inside the stored procedures such as COMMIT or ROLLBACK

**Difference 11:**

**Stored procedures can use all the data types available in sql server.**

The parameters for the stored procedures can be any data types which are available on the sql server.

**But the function cannot use the ntext, image and timestamp data types as return type.**

The function won't allow several data types of the sql server as a parameter.

**Difference 12:**

**Stored procedures can create table variable and cannot return the table variable.**

The table variable is one of the performances tuning mechanism. Because it takes minimum resources and it uses the memory location to store the data. (Recommended for minimum rows)

It can be created and do the operations. **But it cannot be the return type**.

**But the function can create, update and delete the table variable. It can return table variable.**

It can be created and can do all the DML operations and it can be the return type. That is called the **multi valued table function**.

**Difference 13:**

**Stored procedure can have the dynamic sql statement and which can be executed using sp\_executesql statement.**

The stored procedure can have the dynamic sql statement for the complex decision making operations which generated inside the stored procedures. It can be executed using the sp\_executesql statement.

**But the function cannot execute the sp\_executesql statement.**

The function can generate the dynamic sql statement. But it cannot get execute. It will not allow writing the sp\_executesql command to execute the dynamically created sql statement.

**Difference 14:**

**Stored procedure allows getdate () or other non-deterministic functions can be allowed.**   
  
The stored procedure will allow all the **sql server built-in functions like getdate(),DB\_ID(),**

DB\_NAME (), etc..,

**But the function won't allow the non-deterministic functions.**

The function will not allow using non-deterministic **built-in functions** functions like GETDATE ()

**DML** is abbreviation of Data Manipulation Language. It is used to retrieve, store, modify, delete, insert and update data in database.

* SELECT - retrieve data from the a database
* INSERT - insert data into a table
* UPDATE - updates existing data within a table
* DELETE - deletes all records from a table, the space for the records remain
* MERGE - UPSERT operation (insert or update)
* CALL - call a PL/SQL or Java subprogram
* EXPLAIN PLAN - explain access path to data
* LOCK TABLE - control concurrency

**DDL** is abbreviation of Data Definition Language. It is used to create and modify the structure of database objects in database.

* CREATE - to create objects in the database
* ALTER - alters the structure of the database
* DROP - delete objects from the database
* TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
* COMMENT - add comments to the data dictionary
* RENAME - rename an object

### DCL

**Data Control Language** (DCL) statements. Some examples:

* GRANT - gives user's access privileges to database
* REVOKE - withdraw access privileges given with the GRANT command

### TCL

**Transaction Control** (TCL) statements are used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.

* COMMIT - save work done
* SAVEPOINT - identify a point in a transaction to which you can later roll back
* ROLLBACK - restore database to original since the last COMMIT
* SET TRANSACTION - Change transaction options like isolation level and what rollback segment to use

[Choosing Between Transact-SQL and Managed Code](javascript:void(0))

When writing stored procedures, triggers, and user-defined functions, one decision you must make is whether to use traditional Transact-SQL, or a .NET Framework language such as Visual Basic .NET or Visual C#. Use Transact-SQL when the code will mostly perform data access with little or no procedural logic. Use managed code for CPU-intensive functions and procedures that feature complex logic, or when you want to make use of the BCL of the .NET Framework.

### [SQL String Functions Tutorial](http://sqlservercodebook.blogspot.com/2008/03/sql-string-functions-tutorial.html)

**1. CHARINDEX** string function takes 2 arguments. 1st argument specifies the character whose index is to be retrieved and 2nd argument takes as a string from which character index is carried out.

**Example:  
Select CHARINDEX ('S','MICROSOFT SQL SERVER 2000')  
Result: 6**

**2. LEFT** string function takes 2 arguments. 1st argument takes as a string value and 2nd argument as integer value as length parameter. It returns first characters of specified length starting from the left side of the string entered as 1st argument.

**Example:  
Select LEFT ('MICROSOFT SQL SERVER 2000',4)  
Result: MICR**

**3. RIGHT** string function takes 2 arguments. 1st argument takes as a string value and 2nd argument as integer value as length parameter. It returns last characters of specified length starting from the right side of the string entered as 1st argument.

**Example:  
Select RIGHT ('MICROSOFT SQL SERVER 2000',4)  
Result: 2000**

**4. LEN** string function takes 1 argument as string value and returns the length of entered string.

**Example:  
Select LEN ('MICROSOFT SQL SERVER 2000')  
Result: 25**

**5. REPLACE** string function takes 3 arguments.  
1st argument as string value.  
2nd argument is a part of string entered as 1st argument which is to be replaced.  
3rd argument as a new string value that is to be placed at the place of 2nd argument.

**Example:  
Select REPLACE ('MICROSOFT SQL SERVER 2000','MICROSOFT','MS')  
Result: MS SQL SERVER 2000**

**6. STUFF** string function takes 4 arguments. It is used to replace specified length of characters with provided pattern.  
1st argument as string value.  
2nd argument as integer value specifying the starting point of characters to be replaced.  
3rd arguments as integer value specifying the length of characters.  
4th argument as string value specifying the new pattern of characters.

**Example:  
Select STUFF ('MICROSOFT SQL SERVER 2000', 11, 3,'S.Q.L.')  
Result: MICROSFT S.Q.L. SERVER 2000**

**7. SUBSTRING** string function returns the sub string of specified length starting from the entered start position. It takes 3 arguments.  
1st argument as string value.  
2nd argument as integer specifying the start position.  
3rd argument as integer specifying the length

**Example:  
Select SUBSTRING ('MICROSOFT SQL SERVER 2000', 11, 3)  
Result: SQL**

**8. LOWER** string function returns the lower case string whether the entered string has upper case letters. It takes 1 argument as string value.

**Example:  
select LOWER(‘MICROSOFT ASP .NET WEB HOSTING’)  
Result: microsoft asp .net web hosting**

**9. UPPER** string function returns the upper case string whether the entered string has lower case letters. It takes 1 argument as string value.

**Example:  
select LOWER(‘MICROSOFT ASP .NET WEB HOSTING with SQL Database’)  
Result: MICROSOFT ASP .NET WEB HOSTING WITH SQL DATABASE**

**10. REVERSE** string function returns the string in reverse order. It takes 1 argument as string value.

**Example:  
select REVERSE(‘ASP.NET’)  
Result: TEN.PSA**

**11. LTRIM** function returns the string by removing all the blank spaces at left side. It also takes 1 argument as string value.

**Example:  
select LTRIM (‘ ASP ’)  
Result: ASP-----  
blanks at the right side not removed.**

**12. RTRIM** function returns the string by removing all the blank spaces at left side. It also takes 1 argument as string value.

**Example:  
select RTRIM (‘ ASP ’)  
Result: -----ASP  
blanks at the left side not removed.**

**13. PATINDEX** function returns the position of first occurrence of specified pattern in the provided string. It takes 2 arguments.  
1st argument as string value specifying the pattern to match  
2nd argument as string value specifying the string to compare.

**Example:  
select PATINDEX('%RO%','MICROSOFT')  
Results: 4**

**14. STR** function returns character data converted from numeric data. It takes 3 arguments.  
1st argument as float data  
2nd argument as integer value specifying the length of the string including decimal that is to be retrieved.  
3rd argument as integer specifying the number of places to the right of the decimal point.

**Example:  
select STR(140.15, 6, 1)  
Result: 140.2**

**15. ASCII** function returns the ASCII code value from the leftmost character specified character expression. It takes 1 argument as string/character expression.

**Example:  
select ASCII('A')  
Result: 65**

**VARIABLES:**

The difference between a variable and a parameter is:

Parameter - Represents a constant value that cannot be changed throughout the session run.

Variable - Represents a value that can be changed during session run. There are some functions available to change the variable value like setvariable(), setmaxvariable(),..

**USER Defined Variables**

When a variable is first declared, its value is set to NULL. To assign a value to a variable, use the SET statement. This is the preferred method of assigning a value to a variable. A variable can also have a value assigned by being referenced in the select list of a SELECT statement.

To assign a variable a value by using the SET statement, include the variable name and the value to assign to the variable. This is the preferred method of assigning a value to a variable. The following batch, for example, declares two variables, assigns values to them, and then uses them in the WHERE clause of a SELECT statement:

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode_4177512c-54d7-4eab-b18f-b5880ac6c86b');)

USE AdventureWorks2008R2;

GO

-- Declare two variables.

DECLARE @FirstNameVariable nvarchar(50),

@PostalCodeVariable nvarchar(15);

-- Set their values.

SET @FirstNameVariable = N'Amy';

SET @PostalCodeVariable = N'BA5 3HX';

-- Use them in the WHERE clause of a SELECT statement.

SELECT LastName, FirstName, JobTitle, City, StateProvinceName, CountryRegionName

FROM HumanResources.vEmployee

WHERE FirstName = @FirstNameVariable

OR PostalCode = @PostalCodeVariable;

GO

A variable can also have a value assigned by being referenced in a select list. If a variable is referenced in a select list, it should be assigned a scalar value or the SELECT statement should only return one row. For example:

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode_deaed129-dfe4-434a-a99e-1efe2c8264ce');)

USE AdventureWorks2008R2;

GO

DECLARE @EmpIDVariable int;

SELECT @EmpIDVariable = MAX(EmployeeID)

FROM HumanResources.Employee;

GO

# Scalar user-defined functions

Reuse of code is one of the fundamental principles we learn when programming in any language, and the SQL language is no exception. It provides many means by which to logically group code and reuse it.

One such means in SQL Server is the scalar user-defined function. It seems so convenient to hide away all those complex calculations in a function, and then simply invoke it in our queries. However, the hidden "sting in the tail" is that it can bring a heavy toll **in terms of performance. When used in a query, scalar functions are evaluated for each row and, with large tables, this can result in very slow running queries. This is especially true when the scalar function needs to access another table to retrieve data.**

Here is one example. Given tables with products and sales for products, the request is to retrieve total sales per product. Since the total sales value can be reused in another place, you decide to use a scalar function to calculate the total sales for a product:

CREATE FUNCTION dbo.GetTotalSales(@sku INT)

RETURNS DECIMAL(15, 2)

AS

BEGIN

  RETURN(SELECT SUM(sale\_amount)

         FROM Sales

         WHERE sku = @sku);

END

Then the query to retrieve the total sales for each product will look like this;

SELECT sku, product\_description, dbo.GetTotalSales(sku) AS total\_sales

FROM Products;

Isn't this a very neat and good looking query? But just wait until you run it over a large data set. The total sales calculation will be repeated for each and every row, and the overhead will be proportional to the number of rows. **The correct way to handle this is, if possible, is to rewrite the function as a table-valued function, or simply perform the calculation in the main query.** In our example, performing the calculation in the query will look like this:

SELECT P.sku, P.product\_description, SUM(S.sale\_amount) As total\_sales

FROM Products AS P

JOIN Sales AS S

  ON P.sku = S.sku

GROUP BY P.sku, P.product\_description;

And here is a table-valued function that can be used to calculate total sales:

CREATE FUNCTION dbo.GetTotalSales(@sku INT)

**RETURNS TABLE**

AS

RETURN(SELECT SUM(sale\_amount) AS total\_sales

       FROM Sales

       WHERE sku = @sku);

Now the table-valued function can be invoked in the query using the **APPLY** operator:

SELECT sku, product\_description, total\_sales

FROM Products AS P

CROSS APPLY dbo.GetTotalSales(P.sku) AS S;

A scalar-valued function (SVF) returns a single value, such as a string, integer, or bit value. Beginning with SQL Server 2005, you can create scalar-valued user-defined functions in managed code using any .NET Framework programming language. These functions are accessible to Transact-SQL or other managed code. For information about the advantages of CLR integration and choosing between managed code and Transact-SQL,

This article covers all the basics of User Defined Functions. It discusses how (and why) to create them and when to use them. It talks about scalar, inline table-valued and multi-statement table-valued functions. (**This article has been updated through SQL Server 2005.**)

With SQL Server 2000, Microsoft has introduced the concept of User-Defined Functions that allow you to define your own T-SQL functions that can accept zero or more parameters and return a single scalar data value or a table data type.

**What Kind of User-Defined Functions can I Create?**

There are three types of User-Defined functions in SQL Server 2000 and they are **Scalar, Inline Table-Valued and Multi-statement Table-valued.**

**How do I create and use a Scalar User-Defined Function?**

A Scalar user-defined function returns one of the scalar data types. **Text, ntext, image and timestamp data types are not supported**. These are the type of user-defined functions that most developers are used to in other programming languages. You pass in 0 to many parameters and you get a return value. Below is an example that is based in the data found in the NorthWind Customers Table.

CREATE FUNCTION whichContinent

(@Country nvarchar(15))

RETURNS varchar(30)

AS

BEGIN

declare @Return varchar(30)

select @return = case @Country

when 'Argentina' then 'South America'

when 'Belgium' then 'Europe'

when 'Brazil' then 'South America'

when 'Canada' then 'North America'

when 'Denmark' then 'Europe'

when 'Finland' then 'Europe'

when 'France' then 'Europe'

else 'Unknown'

end

return @return

end

Because this function returns a scalar value of a varchar(30) this function could be used anywhere a varchar(30) expression is allowed such as a computed column in a table, view, a T-SQL select list item. Below are some of the examples that I was able to use after creating the above function definition. Note that I had to reference the dbo in the function name.

print dbo.WhichContinent('USA')

select dbo.WhichContinent(Customers.Country), customers.\*

from customers

create table test

(Country varchar(15),

Continent as (dbo.WhichContinent(Country)))

insert into test (country)

values ('USA')

select \* from test

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Country Continent

--------------- ------------------------------

USA North America

Stored procedures have long given us the ability to pass parameters and get a value back, but the ability to use it in such a variety of different places where you cannot use a stored procedure make this a very powerful database object. Also notice the logic of my function is not exactly brain surgery. But it does encapsulate the business rules for the different continents in one location in my application. If you were to build this logic into T-SQL statements scattered throughout your application and you suddenly noticed that you forgot a country (like I missed Austria!) you would have to make the change in every T-SQL statement where you had used that logic. Now, with the SQL Server User-Defined Function, you can quickly maintain this logic in just one place.

**How do I create and use an Inline Table-Value User-Defined Function?**

An Inline Table-Value user-defined function returns a table data type and is an exceptional alternative to a view as the user-defined function can pass parameters into a T-SQL select command and in essence provide us with a parameterized, non-updateable view of the underlying tables.

CREATE FUNCTION CustomersByContinent

(@Continent varchar(30))

**RETURNS TABLE**

AS

RETURN

SELECT dbo.WhichContinent(Customers.Country) as continent,

customers.\*

FROM customers

WHERE dbo.WhichContinent(Customers.Country) = @Continent

GO

SELECT \* from CustomersbyContinent('North America')

SELECT \* from CustomersByContinent('South America')

SELECT \* from customersbyContinent('Unknown')

Note that the example uses another function (WhichContinent) to select out the customers specified by the parameter of this function. After creating the user-defined function, I can use it in the FROM clause of a T-SQL command unlike the behavior found when using a stored procedure which can also return record sets. Also note that I do not have to reference the dbo in my reference to this function. However, when using SQL Server built-in functions that return a table, you must now add the prefix :: to the name of the function.

Example from Books Online: Select \* from ::fn\_helpcollations()

**How do I create and use a Multi-statement Table-Value User-Defined Function?**

A Multi-Statement Table-Value user-defined function returns a table and is also an exceptional alternative to a view as the function can support multiple T-SQL statements to build the final result where the view is limited to a single SELECT statement. Also, the ability to pass parameters into a T-SQL select command or a group of them gives us the capability to in essence create a parameterized, non-updateable view of the data in the underlying tables. Within the create function command you must define the table structure that is being returned. After creating this type of user-defined function, I can use it in the FROM clause of a T-SQL command unlike the behavior found when using a stored procedure which can also return record sets.

CREATE FUNCTION dbo.customersbycountry ( @Country varchar(15) )

RETURNS

@CustomersbyCountryTab table (

[CustomerID] [nchar] (5), [CompanyName] [nvarchar] (40),

[ContactName] [nvarchar] (30), [ContactTitle] [nvarchar] (30),

[Address] [nvarchar] (60), [City] [nvarchar] (15),

[PostalCode] [nvarchar] (10), [Country] [nvarchar] (15),

[Phone] [nvarchar] (24), [Fax] [nvarchar] (24)

)

AS

BEGIN

INSERT INTO @CustomersByCountryTab

SELECT [CustomerID],

[CompanyName],

[ContactName],

[ContactTitle],

[Address],

[City],

[PostalCode],

[Country],

[Phone],

[Fax]

FROM [Northwind].[dbo].[Customers]

WHERE country = @Country

DECLARE @cnt INT

SELECT @cnt = COUNT(\*) FROM @customersbyCountryTab

IF @cnt = 0

INSERT INTO @CustomersByCountryTab (

[CustomerID],

[CompanyName],

[ContactName],

[ContactTitle],

[Address],

[City],

[PostalCode],

[Country],

[Phone],

[Fax] )

VALUES ('','No Companies Found','','','','','','','','')

RETURN

END

GO

SELECT \* FROM dbo.customersbycountry('USA')

SELECT \* FROM dbo.customersbycountry('CANADA')

SELECT \* FROM dbo.customersbycountry('ADF')

**What are the benefits of User-Defined Functions?**

The benefits to SQL Server User-Defined functions are numerous. First, we can use these functions in so many different places when compared to the SQL Server stored procedure. The ability for a function to act like a table (for Inline table and Multi-statement table functions) gives developers the ability to break out complex logic into shorter and shorter code blocks. This will generally give the additional benefit of making the code less complex and easier to write and maintain. In the case of a Scalar User-Defined Function, the ability to use this function anywhere you can use a scalar of the same data type is also a very powerful thing. Combining these advantages with the ability to pass parameters into these database objects makes the SQL Server User-Defined function a very powerful tool.

**Summary**

So, if you have ever wanted to use the results of a stored procedure as part of a T-SQL command, use parameterized non-updateable views, or encapsulate complex logic into a single database object, the SQL Server 2000 User-Defined function is a new database object that you should examine to see if its right for your particular environment.

**LOOPING (WHILE):**

Sets a condition for the repeated execution of an SQL statement or statement block. The statements are executed repeatedly as long as the specified condition is true. The execution of statements in the WHILE loop can be controlled from inside the loop with the BREAK and CONTINUE keywords.

TSQL has WHILE() instead of FOR().

* TSql has While() instead of For.

DECLARE @counter int

SET @counter = 1

WHILE @counter < 10

BEGIN

--DO SOMETHING HERE

SET @counter = @counter +1

END

USE AdventureWorks2012;

GO

WHILE (SELECT AVG(ListPrice) FROM Production.Product) < $300

BEGIN

UPDATE Production.Product

SET ListPrice = ListPrice \* 2

SELECT MAX(ListPrice) FROM Production.Product

IF (SELECT MAX(ListPrice) FROM Production.Product) > $500

BREAK

ELSE

CONTINUE

END

PRINT 'Too much for the market to bear';

Looping is a repeated execution of the same commands until a certain condition evaluates to a different Boolean value (TRUE or FALSE). The only looping structure supported by SQL Server is WHILE loops.   
  
The while loops continue execution until the specified condition is no longer true OR until the loop is intentionally interrupted by a BREAK statement. If BREAK is used then processing passes to the next valid T-SQL statement outside the loop. The CONTINUE statement can be used to return execution to the top of the loop skipping any statements between the CONTINUE and the end of the loop.    
  
The statements inside the WHILE loop need to be enclosed in the BEGIN â€¦ END delimiters. The Boolean condition to be evaluated by the WHILE statement can be a variable value, comparison of variables or a SELECT statement. For instance, the following loop will continue executing while a variable is less than 5:

1.DECLARE @counter INT

2.SELECT @counter = 0

3.WHILE @counter < 5

4.BEGIN

5.  SELECT 'the variable @counter is ' + CAST(@counter AS VARCHAR(1))

6.  SELECT @counter = @counter + 1

7.END

In the next example two variable values are compared. The loop will continue until two of these variables have the same value:

1.DECLARE @counter INT, @counter1 INT

2.SELECT @counter = 0, @counter1 = 2

3.WHILE @counter <> @counter1

4.BEGIN

5.     SELECT 'the variable @counter is less than @counter1'

6.     SELECT @counter = @counter + 1

7.END

The next example shows how to use a SELECT statement as a loop condition:

1.CREATE TABLE #temp (my\_id INT NOT NULL identity(1,1),

2.             my\_name VARCHAR(20) NULL)

3.WHILE (SELECT count(\*) FROM #temp)<10

4.BEGIN

5.  INSERT #temp VALUES ('name\_variable')

6.END

Sometimes you have to write a loop that will execute an unknown number of times before it needs to stop executing. This can be accomplished by writing an infinite loop that contains a BREAK statement after the work has been accomplished, as in the following example:

1.<P>WHILE 10 = 10

2.BEGIN

3.INSERT #temp VALUES ('name\_variable')

4.BREAK

5.</P>

6.END

Although 10 will always equal 10 (an infinite loop), the BREAK statement will unconditionally stop execution of the program right after the first record has been inserted into the temporary table.    
  
The while loops can be nested. You need to exercise great care when working with nested loops since the innermost BREAK statement will only get out of the current loop and get into the next outermost loop. It will not completely exit the topmost loop. Any valid T-SQL statements following the innermost loop are executed before the next outermost loop restarts.    
  
If you wish to go to the beginning of the loop use the CONTINUE statement. This will cause the loop to reexamine the WHILE condition. If the condition still evaluates to true the loop will re-execute, otherwise the processing will continue at the statement immediately following the loop.    
  
Occasionally you might wish to terminate the loop before the WHILE condition evaluates to false. One way to do so would be using GOTO. You can use GOTO to jump to any labeled portion of code within or outside of the loop. If you wish to completely get out of the program execution, use the RETURN statement. RETURN will ultimately end the whole program, not just the loop. Finally, you can use multiple BREAK statements to get out of all levels of looping.    
  
The following example demonstrates the usage of nested loops. This code returns the list of the stores that have sold 50 or more items in each year of operations, incrementing the minimum number of items by 10. If the program finds that none of the stores sold more than 50 items during a particular year it will exit from the inner loop and check the following year:

01.<P>SET NOCOUNT ON

02./\* create a temporary table to hold values \*/

03.CREATE TABLE #temp (store INT NOT NULL, total\_items INT NOT NULL)

04.DECLARE @n INT, @m INT

05.SELECT @n = 50

06.SELECT @m = (SELECT DATEPART(YY, (MIN(ord\_date))) FROM sales)  -- outer loop

07.WHILE @m < = (SELECT DATEPART(YY, (MAX(ord\_date))) FROM sales)

08.BEGIN  -- inner loop

09.WHILE @n < (SELECT MAX(qty) FROM sales)

10.BEGIN

11.INSERT #temp

12.SELECT stor\_id, SUM(qty) FROM sales

13.WHERE DATEPART(YY, ord\_date) = @m

14.GROUP BY stor\_id

15.HAVING SUM(qty) > @n

16.IF (SELECT COUNT(\*) FROM #temp )<> 0

17.BEGIN

18.SELECT 'the following stores have sold more than ' +

19.CONVERT(VARCHAR(2), @n) + ' items in ' +

20.CONVERT(VARCHAR(4), @m)

21.SELECT \* FROM #temp

22.END

23.IF @@rowcount = 0

24.BREAK

25./\* empty out the temporary table and increment @n variable \*/

26.TRUNCATE TABLE #temp

27.SELECT @n = @n + 10

28.END

29.SELECT @m = @m + 1

30.SELECT @n = 50

31.END

32.</P>

33.DROP TABLE #temD

01.---------------------------------------------------------

02.<P>the following stores have sold more than 50 items in 1992

03.store       total\_items

04.----------- -----------

05.7067        80

06.---------------------------------------------------------

07.the following stores have sold more than 60 items in 1992

08.store       total\_items

09.----------- -----------

10.7067        80

11.---------------------------------------------------------

12.the following stores have sold more than 70 items in 1992

13.store       total\_items

14.----------- -----------

15.7067        80

16.---------------------------------------------------------

17.the following stores have sold more than 50 items in 1993

18.store       total\_items

19.----------- -----------

20.7131        85

21.7896        60

22.8042        55

23.---------------------------------------------------------

24.the following stores have sold more than 60 items in 1993

25.store       total\_items

26.----------- -----------

27.7131        85

28.---------------------------------------------------------

29.the following stores have sold more than 70 items in 1993

30.store       total\_items

31.----------- -----------

32.7131        85

33.---------------------------------------------------------

34.the following stores have sold more than 50 items in 1994

35.store       total\_items

36.----------- -----------

37.7066        75

38.---------------------------------------------------------

39.the following stores have sold more than 60 items in 1994

40.store       total\_items

41.----------- -----------

42.7066        75

43.---------------------------------------------------------

44.the following stores have sold more than 70 items in 1994

45.store       total\_items

46.----------- -----------

47.</P>

48.7066        75

**CASE WHEN (Replaces IF THEN ELSE) :**

The T-SQL CASE...WHEN statement is very similar to a switch, IF THEN, or case statement in other computer languages. The key that makes it especially useful is that you can use CASE...WHEN within a SELECT statement. Technically, CASE evaluates a list of conditions and returns one of multiple possible result expressions—so it's easy to see how someone new to the language might try to use a series of IF...THEN statements instead of looking for something a little more powerful.

Evaluates a list of conditions and returns one of multiple possible result expressions.

The CASE expression has two formats:

The simple CASE expression compares an expression to a set of simple expressions to determine the result.

The searched CASE expression evaluates a set of Boolean expressions to determine the result.

Both formats support an optional ELSE argument.

CASE can be used in any statement or clause that allows a valid expression. For example, you can use CASE in statements such as SELECT, UPDATE, DELETE and SET, and in clauses such as select\_list, IN, WHERE, ORDER BY, and HAVING.

**Within a SELECT statement,** a simple CASE expression allows for only an equality check; no other comparisons are made. The following example uses the CASE expression to change the display of product line categories to make them more understandable.

USE AdventureWorks2012;

GO

SELECT ProductNumber, Category =

CASE ProductLine

WHEN 'R' THEN 'Road'

WHEN 'M' THEN 'Mountain'

WHEN 'T' THEN 'Touring'

WHEN 'S' THEN 'Other sale items'

ELSE 'Not for sale'

END,

Name

FROM Production.Product

ORDER BY ProductNumber;

GO

Within a SELECT statement, the searched CASE expression allows for values to be replaced in the result set based on comparison values. The following example displays the list price as a text comment based on the price range for a product.

USE AdventureWorks2012;

GO

SELECT ProductNumber, Name, "Price Range" =

CASE

WHEN ListPrice = 0 THEN 'Mfg item - not for resale'

WHEN ListPrice < 50 THEN 'Under $50'

WHEN ListPrice >= 50 and ListPrice < 250 THEN 'Under $250'

WHEN ListPrice >= 250 and ListPrice < 1000 THEN 'Under $1000'

ELSE 'Over $1000'

END

FROM Production.Product

ORDER BY ProductNumber ;

GO

**Using CASE in a SET statement**

The following example uses the CASE expression in a SET statement in the table-valued function dbo.GetContactInfo. In the AdventureWorks2012 database, all data related to people is stored in the Person.Person table. For example, the person may be an employee, vendor representative, or a customer. The function returns the first and last name of a given BusinessEntityID and the contact type for that person.The CASE expression in the SET statement determines the value to display for the column ContactType based on the existence of the BusinessEntityID column in the Employee, Vendor, or Customer tables.

USE AdventureWorks2012;

GO

CREATE FUNCTION dbo.GetContactInformation(@BusinessEntityID int)

**RETURNS @retContactInformation TABLE**

(

BusinessEntityID int NOT NULL,

FirstName nvarchar(50) NULL,

LastName nvarchar(50) NULL,

ContactType nvarchar(50) NULL,

PRIMARY KEY CLUSTERED (BusinessEntityID ASC)

)

AS

-- Returns the first name, last name and contact type for the specified contact.

BEGIN

DECLARE

@FirstName nvarchar(50),

@LastName nvarchar(50),

@ContactType nvarchar(50);

-- Get common contact information

SELECT

@BusinessEntityID = BusinessEntityID,

@FirstName = FirstName,

@LastName = LastName

FROM Person.Person

WHERE BusinessEntityID = @BusinessEntityID;

SET @ContactType =

CASE

-- Check for employee

WHEN EXISTS(SELECT \* FROM HumanResources.Employee AS e

WHERE e.BusinessEntityID = @BusinessEntityID)

THEN 'Employee'

-- Check for vendor

WHEN EXISTS(SELECT \* FROM Person.BusinessEntityContact AS bec

WHERE bec.BusinessEntityID = @BusinessEntityID)

THEN 'Vendor'

-- Check for store

WHEN EXISTS(SELECT \* FROM Purchasing.Vendor AS v

WHERE v.BusinessEntityID = @BusinessEntityID)

THEN 'Store Contact'

-- Check for individual consumer

WHEN EXISTS(SELECT \* FROM Sales.Customer AS c

WHERE c.PersonID = @BusinessEntityID)

THEN 'Consumer'

END;

-- Return the information to the caller

IF @BusinessEntityID IS NOT NULL

BEGIN

INSERT @retContactInformation

SELECT @BusinessEntityID, @FirstName, @LastName, @ContactType;

END;

RETURN;

END;

GO

SELECT BusinessEntityID, FirstName, LastName, ContactType

FROM dbo.GetContactInformation(2200);

GO

SELECT BusinessEntityID, FirstName, LastName, ContactType

FROM dbo.GetContactInformation(5);

**Using CASE in a HAVING clause**

The following example uses the CASE expression in a HAVING clause to restrict the rows returned by the SELECT statement. The statement returns the the maximum hourly rate for each job title in the HumanResources. Employee table. The HAVING clause restricts the titles to those that are held by men with a maximum pay rate greater than 40 dollars or women with a maximum pay rate greater than 42 dollars.

USE AdventureWorks2012;

GO

SELECT JobTitle, MAX(ph1.Rate)AS MaximumRate

FROM HumanResources.Employee AS e

JOIN HumanResources.EmployeePayHistory AS ph1 ON e.BusinessEntityID = ph1.BusinessEntityID

GROUP BY JobTitle

HAVING (MAX(CASE WHEN Gender = 'M'

THEN ph1.Rate

ELSE NULL END) > 40.00

OR MAX(CASE WHEN Gender = 'F'

THEN ph1.Rate

ELSE NULL END) > 42.00)

ORDER BY MaximumRate DESC;

**MY EXAMPLE FOR LMS AR**

SELECT

CASE WHEN A.INS\_CO\_AR IN ('mc1','mc2','mc3','mc4','mca','mcb','mcc','mcd',

'mcf','sx4','sx5','8aa','8ab','8ac','8ad','8cc')

THEN 'MEDICARE'

WHEN A.INS\_CO\_AR IN ('15R','MAK','TG4','ST2','1A9','RB9','05Z','SO7','19X','EE1','0NQ','BC8',

'HE8','HI7','QS2','TC1')

THEN 'MEDICAID'

WHEN A.INS\_CO\_AR IN ('0','000')

THEN 'PRIVATE PAY'

ELSE 'OTHER'

END AS 'INSURANCE',

--The group by is summing these total amounts

SUM(CONVERT(money, A.TOTALAMOUNT)) AS 'TOTAL\_OPEN\_AR',

COUNT(DISTINCT A.APPLY\_TO\_AR) AS 'TOTAL\_INVOICE',

SUM(CASE WHEN DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate()) >=0

AND DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate() ) <=90

THEN CONVERT(money, A.TOTALAMOUNT) ELSE 0 END) AS [DAYS\_0\_to\_90],

COUNT(DISTINCT CASE WHEN DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate()) >=0

AND DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate() ) <=90

THEN A.APPLY\_TO\_AR ELSE 0 END) AS [INVOICE\_0\_to\_90],

SUM(CASE WHEN DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate()) >=91

AND DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate() ) <=180

THEN CONVERT(money, A.TOTALAMOUNT) ELSE 0 END) AS [DAYS\_91\_to\_180],

COUNT(DISTINCT CASE WHEN DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate()) >=91

AND DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate() ) <=180

THEN A.APPLY\_TO\_AR ELSE 0 END) AS [INVOICE\_91\_to\_180],

SUM(CASE WHEN DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate()) >=181

AND DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate() ) <=270

THEN CONVERT(money, A.TOTALAMOUNT) ELSE 0 END) AS [DAYS\_181\_to\_270],

COUNT(DISTINCT CASE WHEN DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate()) >=181

AND DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate() ) <=270

THEN A.APPLY\_TO\_AR ELSE 0 END) AS [INVOICE\_181\_to\_270],

SUM(CASE WHEN DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate()) >=271

AND DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate() ) <=360

THEN CONVERT(money, A.TOTALAMOUNT) ELSE 0 END) AS [DAYS\_271\_to\_360],

COUNT(DISTINCT CASE WHEN DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate()) >=271

AND DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate() ) <=360

THEN A.APPLY\_TO\_AR ELSE 0 END) AS [INVOICE\_271\_to\_360],

SUM(CASE WHEN DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate()) >=361

AND DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate() ) <=450

THEN CONVERT(money, A.TOTALAMOUNT) ELSE 0 END) AS [DAYS\_361\_to\_450],

COUNT(DISTINCT CASE WHEN DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate()) >=361

AND DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate() ) <=450

THEN A.APPLY\_TO\_AR ELSE 0 END) AS [INVOICE\_361\_to\_450],

SUM(CASE WHEN DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate()) >=451

AND DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate() ) <=540

THEN CONVERT(money, A.TOTALAMOUNT) ELSE 0 END) AS [DAYS\_451\_to\_540],

COUNT(DISTINCT CASE WHEN DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate()) >=451

AND DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate() ) <=540

THEN A.APPLY\_TO\_AR ELSE 0 END) AS [INVOICE\_451\_to\_540],

SUM(CASE WHEN DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate()) >=541

THEN CONVERT(money, A.TOTALAMOUNT) ELSE 0 END) AS [OVER 540 DAYS],

COUNT(DISTINCT CASE WHEN DATEDIFF(dd, C.SHIP\_DT\_CH, Getdate()) >=540

THEN A.APPLY\_TO\_AR ELSE 0 END) AS [OVER\_540 INVOICE]

FROM AR.dbo.tblAROPEN AS A WITH (NOLOCK)

INNER JOIN AR.dbo.tblCLMHDR AS C WITH (NOLOCK)

ON A.CUS\_NUM\_AR = C.CUSTNUM\_CH

AND A.APPLY\_TO\_AR = C.fINVNUM\_CH

--WHERE A.TOTALAMOUNT <>0

GROUP BY

CASE WHEN A.INS\_CO\_AR IN ('mc1','mc2','mc3','mc4','mca','mcb','mcc','mcd',

'mcf','sx4','sx5','8aa','8ab','8ac','8ad','8cc')

THEN 'MEDICARE'

WHEN A.INS\_CO\_AR IN ('15R','MAK','TG4','ST2','1A9','RB9','05Z','SO7','19X','EE1','0NQ','BC8',

'HE8','HI7','QS2','TC1')

THEN 'MEDICAID'

WHEN A.INS\_CO\_AR IN ('0','000')

THEN 'PRIVATE PAY'

ELSE 'OTHER'

END

**PARAMETERS:**

Parameters are generally used the same as regular variables in a T-SQL statement as they are preceded with an @ symbol and must be declared with a data type. **The difference between stored procedure parameters and standard variables is that parameters are passed in during the execution call by the application or website.**

[T-SQL](http://devguru.com/technologies/t-sql/home.asp) » [Stored Procedures](http://devguru.com/technologies/t-sql/7129.asp) » Using Parameters

Parameters can be passed to the stored procedures. This makes the procedure dynamic.

The following [points](http://devguru.com/technologies/t-sql/7132.asp) are to be noted:

* One or more number of parameters can be passed in a procedure.
* The parameter name should proceed with an @ symbol.
* The parameter names will be local to the procedure in which they are defined.

The parameters are used to pass information into a procedure from the line that executes the parameter. The parameters are given just after the name of the procedure on a command line. Commas should separate the list of parameters.   
  
The values can be passed to stored procedures by:

1. By supplying the parameter values exactly in the same order as given in the CREATE PROCEDURE statement.
2. By explicitly naming the parameters and assigning the appropriate value.

## Examples

##### Code:

CREATE PROCEDURE spSelectStudent (@Course INTEGER, @Grade INTEGER)   
AS   
SELECT \* FROM Students   
WHERE Std\_Course=@Course AND Std\_Grade <= @Grade   
GO  
EXEC spSelectStudent 3, 2;

##### Output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | Name | Std\_Course | Phone | Std\_Grade |
|  |  |  |  |  |
| 3 | Harri Robins | 3 | 78788 | 1 |
| 4 | Joe Philip | 3 | 46456 | 2 |
|  |  |  |  |  |

(2 row(s) affected)

##### Explanation:

In the above example, the procedure is defined with two parameters. It should be noted that while executing the procedure the parameters should be passed in the same order of that in the CREATE statement. In this case, the first argument corresponds to Std\_Course and second argument to Std\_Grade.

**DEFAULT PARAMITERS**

You can create a stored procedure with optional parameters by specifying a default value for optional parameters. When the stored procedure is executed, the default value is used if no other value has been specified.

Specifying default values is necessary because a system error is returned if a parameter does not have a default value specified in the stored procedure and the calling program does not provide a value for the parameter when the stored procedure is executed.

If no value can be specified appropriately as a default for the parameter, you can specify NULL as the default for a parameter and have the stored procedure return a customized message if the stored procedure is executed without a value for the parameter.

The following example creates the usp\_GetSalesYTD procedure with one input parameter, @SalesPerson. NULL is assigned as the default value for the parameter and is used in error handling statements to return a custom error message for cases when the stored procedure is executed without a value for the @SalesPerson parameter.

USE AdventureWorks2008R2;

GO

IF OBJECT\_ID('Sales.uspGetSalesYTD', 'P') IS NOT NULL

DROP PROCEDURE Sales.uspGetSalesYTD;

GO

CREATE PROCEDURE Sales.uspGetSalesYTD

@SalesPerson nvarchar(50) **= NULL -- NULL default value**

AS

SET NOCOUNT ON;

-- Validate the @SalesPerson parameter.

IF @SalesPerson IS NULL

BEGIN

PRINT 'ERROR: You must specify the last name of the sales person.'

RETURN

END

-- Get the sales for the specified sales person and

-- assign it to the output parameter.

SELECT SalesYTD

FROM Sales.SalesPerson AS sp

JOIN HumanResources.vEmployee AS e ON e.BusinessEntityID = sp.BusinessEntityID

WHERE LastName = @SalesPerson;

RETURN

GO

The following example executes the stored procedure. The first statement executes the stored procedure without specifying an input value. This causes the error handling statements in the stored procedure to return the custom error message. The second statement supplies an input value and returns the expected result set.

-- Run the stored procedure without specifying an input value.

EXEC Sales.usp\_GetSalesYTD;

GO

-- Run the stored procedure with an input value.

EXEC Sales.usp\_GetSalesYTD N'Blythe';

GO

The following example shows the **my\_proc** procedure with default values for each of the three parameters **@first**, **@second**, and **@third**, and the values displayed when the stored procedure is executed with other parameter values:

**Transact-SQL**

IF OBJECT\_ID('dbo.my\_proc', 'P') IS NOT NULL

DROP PROCEDURE dbo.my\_proc;

GO

CREATE PROCEDURE dbo.my\_proc

@first int = NULL, -- NULL default value

@second int = 2, -- Default value of 2

@third int = 3 -- Default value of 3

AS

SET NOCOUNT ON;

SELECT @first, @second, @third;

GO

EXECUTE dbo.my\_proc; -- No parameters supplied

GO

Here is the result set.

NULL 2 3

EXECUTE dbo.my\_proc 10, 20, 30;-- All parameters supplied

GO

Here is the result set.

10 20 30

EXECUTE dbo.my\_proc @second = 500; -- Only second parameter supplied by name

GO

Here is the result set.

NULL 500 3

EXECUTE dbo.my\_proc 40, @third = 50 -- Only first and third parameters

-- are supplied.

Here is the result set.

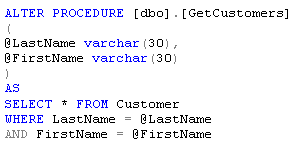
40 2 50

**OPTIONAL PARAMETERS**

You have a stored procedure GetCustomers with two parameters: LastName, FirstName. The stored procedure returns all the records matching the values of the parameters. You want the parameters be optional, which means skipping the parameter if you do not pass a value.

T-SQL does not provide optional parameters, but you can implement one.

1.       You have original stored procedure



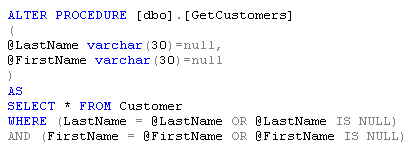
2.       Add =null at your parameter declaration of the stored procedure

http://weblogs.asp.net/blogs/stanleygu/image_20100208_Param2.gif

3.       Add IS NULL at your WHERE clause

http://weblogs.asp.net/blogs/stanleygu/image_20100208_Param3.gif

4.       Now you have optional parameters in the stored procedure



**SubQueries:**

# http://www.w3resource.com/sql/subqueries/understanding-sql-subqueries.php

# Plain Subqueries

**EXAMPLE**

SELECT dbo.Company\_Team.Last\_Update, dbo.Company\_Team.Updated\_By, dbo.Company\_Team.Description,

Company\_1.Company\_Name, dbo.Contact.First\_Name, dbo.Contact.Last\_Name,

(SELECT Company\_Name

FROM dbo.Company

WHERE (Company\_RecID = dbo.Contact.Company\_RecID)) AS ReferCo

FROM dbo.Contact RIGHT OUTER JOIN

dbo.Company\_Team ON dbo.Contact.Contact\_RecID = dbo.Company\_Team.Contact\_Recid LEFT OUTER JOIN

dbo.Company AS Company\_1 ON dbo.Company\_Team.Company\_RecID = Company\_1.Company\_RecID

WHERE (dbo.Company\_Team.Team\_Role\_Recid = '10')

**EXAMPLE**

set @timetofail = 23000;

select

                COUNT(distinct (C.LocationID))AS 'Total Locations',

                L.LocationID,

                (select max(TIMESTAMPDIFF(minute,  C1.LastContact, NOW() ))

                                from computers c1 where c1.locationid = c.locationid) AS MinutesToLastContact,

    IF((select max(TIMESTAMPDIFF(minute,  C1.LastContact, NOW() ))

                                from computers c1 where c1.locationid = c.locationid) >= @TimeToFail,'Fail','OK') AS 'Status',

                (select max(c1.lastcontact) from computers c1 where c1.locationid = c.locationid) as lastcontact

from computers C

    JOIN Locations AS L

                ON C.LocationID = L.LocationID

where

L.LocationID NOT IN (0 , 1)

AND IFNULL(TIMESTAMPDIFF(minute, C.LastContact, NOW()),0) >= @TimeToFail

AND C.LastContact > DATE\_ADD(NOW(), INTERVAL -1 year)

GROUP BY C.LocationID

ORDER BY MinutesToLastContact desc

SELECT     dbo.Company\_Team.Last\_Update, dbo.Company\_Team.Updated\_By, dbo.Company\_Team.Description, Company\_1.Company\_Name, dbo.Contact.First\_Name, dbo.Contact.Last\_Name,

                          (SELECT     Company\_Name

                            FROM          dbo.Company

                            WHERE      (Company\_RecID = dbo.Contact.Company\_RecID)) AS ReferCo

FROM         dbo.Contact RIGHT OUTER JOIN

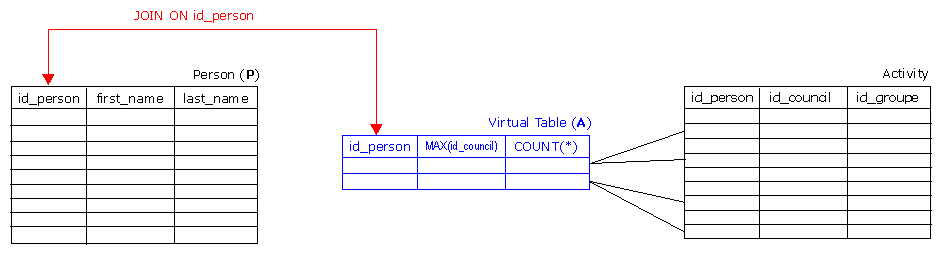
                      dbo.Company\_Team ON dbo.Contact.Contact\_RecID = dbo.Company\_Team.Contact\_Recid LEFT OUTER JOIN

                      dbo.Company AS Company\_1 ON dbo.Company\_Team.Company\_RecID = Company\_1.Company\_RecID

WHERE     (dbo.Company\_Team.Team\_Role\_Recid = '10')

Joining Virtual Tables

Joining virtual Tables is one of the most powerful solution you can build with subqueries. Virtual means in this context, that the result set you are joining is build on the fly. The following example shows, how to join a GROUP BY result set with another, real table (Person).



SELECT P.id\_person,  
P.first\_name,  
P.last\_name,  
CONVERT(varchar(30), P.birth, 104),  
A.id\_council,  
A.id\_groupe,  
A.numActivities  
FROM Person P **JOIN** (SELECT id\_person,  
MIN(id\_council) id\_council,  
MIN(id\_groupe) id\_groupe,  
COUNT(\*) numActivities  
FROM Activity  
GROUP BY id\_person) **A** ON (A.id\_person = P.id\_person)  
WHERE P.id\_person NOT IN (SELECT id\_person  
FROM Activity  
WHERE id\_council != 5)

The virtual Table is referenced in the outer query by the **alias A** and is joined with person\_id. You can use the virtual table columns in the outer query using the alias A. for example A.numActivities.

Joining more than one virtual Table (SQL Server)

The next example shows a very complex query using more than one virtual table.

--  
-- Declare Variables  
--  
DECLARE @LaufID BIGINT  
DECLARE @AbrDatum DATETIME  
DECLARE @CountLauf INT  
--  
-- Fill Variables  
--  
SELECT @LaufID = MAX(LaufID),  
@AbrDatum = MAX(AbrDatum),  
@CountLauf = COUNT(\*)  
FROM AbrLauf  
WHERE BuchDatum >= CONVERT(datetime, @DatumVon, 104)  
AND BuchDatum < DATEADD(day, 1, CONVERT(datetime, @DatumBis, 104))  
--  
-- Generate Report  
--  
SELECT P.Nr,  
P.Name,  
P.Vorname,  
CASE R.Rat WHEN 1 THEN 'NR' WHEN 2 THEN 'SR' ELSE NULL END Rat,  
ISNULL(Entschaedigung.Betrag, 0) EntschaedigungBetrag,  
ISNULL(Vorsorge.Betrag, 0) VorsorgeBetrag,  
ISNULL(Entschaedigung.Betrag, 0) + ISNULL(Vorsorge.Betrag, 0) Total,  
CONVERT(varchar(30), @DatumVon, 104) DatumVon,  
CONVERT(varchar(30), @DatumBis, 104) DatumBis,  
@LaufID LaufID,  
CONVERT(varchar(30), @AbrDatum, 104) AbrDatum,  
@CountLauf CountLauf  
FROM **Person P**  
--  
-- Now join the real Table P with the virtaul Table R ...  
--  
LEFT OUTER JOIN (SELECT M.PersonID,  
M.Rat  
FROM Ratsmitglied M  
WHERE M.Eintritt = (SELECT MAX(MI.Eintritt)  
FROM Ratsmitglied MI  
WHERE MI.PersonID = M.PersonID)) **R**  
ON (**P.PersonID** = **R.PersonID**)  
--  
-- ... then join Table P with the virtaul Table 'Entschaedigung'  
--  
LEFT OUTER JOIN (SELECT PersonID,  
SUM(Betrag) Betrag  
FROM ExportKreditor  
WHERE ExportKreditorID IN (SELECT EAEK.ExportKreditorID  
FROM EntAbrExportKreditor EAEK  
JOIN EntAbr EA ON (EA.EntAbrID = EAEK.EntAbrID)  
JOIN Abr A ON (A.AbrID = EA.AbrID)  
JOIN AbrArt AA ON (AA.AbrArtID = A.AbrArtID)  
WHERE AA.Abk = 'A')  
AND SollHabenBez = 'H'  
AND BuchDatum >= CONVERT(datetime, @DatumVon, 104)  
AND BuchDatum < DATEADD(day, 1, CONVERT(datetime, @DatumBis, 104))  
GROUP BY PersonID) **Entschaedigung**  
ON (**P.PersonID** = **Entschaedigung.PersonID**)  
--  
-- ... then join Table P with the virtaul Table 'Vorsorge'  
--  
LEFT OUTER JOIN (SELECT PersonID,  
SUM(Betrag) Betrag  
FROM ExportKreditor

WHERE ExportKreditorID IN (SELECT EAEK.ExportKreditorID  
FROM EntAbrExportKreditor EAEK  
JOIN EntAbr EA ON (EA.EntAbrID = EAEK.EntAbrID)  
JOIN Abr A ON (A.AbrID = EA.AbrID)  
JOIN AbrArt AA ON (AA.AbrArtID = A.AbrArtID)  
WHERE AA.Abk = 'V')  
AND SollHabenBez = 'H'  
AND BuchDatum >= CONVERT(datetime, @DatumVon, 104)  
AND BuchDatum < DATEADD(day, 1, CONVERT(datetime, @DatumBis, 104))  
GROUP BY PersonID) **Vorsorge**  
ON (**P.PersonID** = **Vorsorge.PersonID**)  
--  
-- ... then the final WHERE Clause, based on the virtual Tables  
--  
WHERE ISNULL(**Entschaedigung.Betrag**, 0) + ISNULL(**Vorsorge.Betrag**, 0) > 0  
ORDER BY P.Name, P.Vorname, R.Rat

Use of a Subquery in the Column List of a SELECT Statement

Suppose you would like to see the last OrderID and the OrderDate for the last order that was shipped to Paris. Along with that information, say you would also like to see the OrderDate for the last order shipped regardless of the ShipCity. In addition to this, you would also like to calculate the difference in days between the two different OrderDates. Here is my T-SQL SELECT statement to accomplish this:

**SELECT TOP 1 OrderId,  
CONVERT(CHAR(10), OrderDate,121) Last\_Paris\_Order,  
(SELECT CONVERT(CHAR(10),MAX(OrderDate),121)  
FROM Northwind.dbo.Orders) Last\_OrderDate,  
DATEDIFF(dd,OrderDate,(SELECT MAX(OrderDate)  
FROM Northwind.dbo.Orders)) Day\_Diff  
FROM Northwind.dbo.Orders  
WHERE ShipCity = 'Paris'  
ORDER BY OrderDate DESC**

The above code contains two subqueries. The first subquery gets the OrderDate for the last order shipped regardless of ShipCity, and the second subquery calculates the number of days between the two different OrderDates. Here we used the first subquery to return a column value in the final result set. The second subquery was used as a parameter in a function call. This subquery passed the "max(OrderDate)" date to the DATEDIFF function.

Use of a Subquery in the WHERE clause

A subquery can be used to control the records returned from a SELECT by controlling which records pass the conditions of a WHERE clause. In this case the results of the subquery would be used on one side of a WHERE clause condition. Here is an example:

**SELECT DISTINCT country  
FROM Northwind.dbo.Customers  
WHERE country NOT IN (SELECT DISTINCT country  
FROM Northwind.dbo.Suppliers)**

Here we have returned a list of countries where customers live, but there is no supplier located in that country. We suppose if you where trying to provide better delivery time to customers, then you might target these countries to look for additional suppliers.

Suppose a company would like to do some targeted marketing. This targeted marketing would contact customers in the country with the fewest number of orders. It is hoped that this targeted marketing will increase the overall sales in the targeted country. Here is an example that uses a subquery to return the customer contact information for the country with the fewest number of orders:

**SELECT Country,  
CompanyName,  
ContactName,  
ContactTitle,  
Phone  
FROM Northwind.dbo.Customers  
WHERE country = (SELECT TOP 1 country  
FROM Northwind.dbo.Customers C  
JOIN Northwind.dbo.Orders O  
ON C.CustomerId = O.CustomerID  
GROUP BY country  
ORDER BY count(\*))**

Here we have written a subquery that joins the Customer and Orders Tables to determine the total number of orders for each country. The subquery uses the "TOP 1" clause to return the country with the fewest number of orders. The country with the fewest number of orders is then used in the WHERE clause to determine which Customer Information will be displayed.

Use of a Subquery in the FROM clause

The FROM clause normally identifies the tables used in the T-SQL statement. You can think of each of the tables identified in the FROM clause as a set of records. Well, a subquery is just a set of records, and therefore can be used in the FROM clause just like a table. Here is an example where a subquery is used in the FROM clause of a SELECT statement:

**SELECT au\_lname,**

**au\_fname,**

**title FROM (SELECT au\_lname, au\_fname, au\_id**

**FROM pubs.dbo.authors**

**WHERE state = 'CA') as A**

**JOIN pubs.dbo.titleauthor ta ON A.au\_id = ta.au\_id**

**JOIN pubs.dbo.titles t ON ta.title\_id = t.title\_id**

Here we have used a subquery to select only the author record information, if the author's record has a state column equal to "CA." We have named the set returned from this subquery with a table alias of "A". WeI can then use this alias elsewhere in the T-SQL statement to refer to the columns from the subquery by prefixing them with an "A", as we did in the "ON" clause of the "JOIN" criteria. Sometimes using a subquery in the FROM clause reduces the size of the set that needs to be joined. Reducing the number of records that have to be joined enhances the performance of joining rows, and therefore speeds up the overall execution of a query.

Subquery in the FROM clause of an UPDATE statement:

SET NOCOUNT ON

CREATE TABLE x(

i INT IDENTITY,

a CHAR(1))

INSERT INTO x VALUES ('A')

INSERT INTO x VALUES ('B')

INSERT INTO x VALUES ('C')

INSERT INTO x VALUES ('D')

SELECT \* FROM x

**UPDATE x**

**SET a = b.a**

**FROM (SELECT MAX(a) AS a FROM x) b**

**WHERE I > 2**

SELECT \* FROM x

DROP TABLE x

Here we created a table named "x" that has four rows. Then we proceeded to update the rows where "i" was greater than 2 with the max value in column "a". We used a subquery in the FROM clause of the UPDATE statement to identity the max value of column "a."

Use of a Subquery in the HAVING clause

In the following example, we used a subquery to find the number of books a publisher has published where the publisher is not located in the state of California. To accomplish this we used a subquery in a HAVING clause. Here is the code:

**SELECT pub\_name,  
COUNT(\*) bookcnt  
FROM pubs.dbo.titles t  
JOIN pubs.dbo.publishers p on t.pub\_id = p.pub\_id  
GROUP BY pub\_name  
HAVING p.pub\_name IN (SELECT pub\_name  
FROM pubs.dbo.publishers  
WHERE state <> 'CA')**

Here the subquery returns the pub\_name values for all publishers that have a state value not equal to "CA." The HAVING condition then checks to see if the pub\_name is in the set returned by my subquery.

# Correlated Subqueries

A correlated subquery is a SELECT statement nested inside another T-SQL statement, which contains a reference to one or more columns in the outer query. Therefore, **the correlated subquery can be said to be dependent on the outer query. This is the main difference between a correlated subquery and just a plain subquery. A plain subquery is not dependent on the outer query, can be run independently of the outer query, and will return a result set**. A correlated subquery, since it is dependent on the outer query will return a syntax errors if it is run by itself.

A correlated subquery will be executed many times while processing the T-SQL statement that contains the correlated subquery. The correlated subquery will be run once for each candidate row selected by the outer query. The outer query columns, referenced in the correlated subquery, are replaced with values from the candidate row prior to each execution. Depending on the results of the execution of the correlated subquery, it will determine if the row of the outer query is returned in the final result set.

Using a Correlated Subquery in a WHERE Clause

Suppose you want a report of all "OrderID's" where the customer did not purchase more than 10% of the average quantity sold for a given product. This way you could review these orders, and possibly contact the customers, to help determine if there was a reason for the low quantity order. A correlated subquery in a WHERE clause can help you produce this report. Here is a SELECT statement that produces the desired list of "OrderID's":

**SELECT DISTINCT OrderId  
FROM Northwind.dbo.[Order Details] OD  
WHERE Quantity > (SELECT AVG(Quantity) \* .1  
FROM Northwind.dbo.[Order Details]  
WHERE OD.ProductID = ProductID)**

The correlated subquery in the above command is contained within the parenthesis following the greater than sign in the WHERE clause above. Here you can see this correlated subquery contains a reference to "OD.ProductID". This reference compares the outer query's "ProductID" with the inner query's "ProductID". When this query is executed, the SQL engine will execute the inner query, the correlated subquery, for each "[Order Details]" record. This inner query will calculate the average "Quantity" for the particular "ProductID" for the candidate row being processed in the outer query. This correlated subquery determines if the inner query returns a value that meets the condition of the WHERE clause. If it does, the row identified by the outer query is placed in the record set that will be returned from the complete T-SQL SELECT statement.

The code below is another example that uses a correlated subquery in the WHERE clause to display the top two customers, based on the dollar amount associated with their orders, per region. You might want to perform a query like this so you can reward these customers, since they buy the most per region.

**SELECT C1.CompanyName,  
C1.ContactName,  
C1.Address,  
C1.City,  
C1.Country,  
C1.PostalCode  
FROM Northwind.dbo.Customers C1  
WHERE C1.CustomerID IN** **(SELECT TOP 2 C2.CustomerId  
FROM Northwind.dbo.[Order Details] OD  
JOIN Northwind.dbo.Orders O on OD.OrderId = O.OrderID  
JOIN Northwind.dbo.Customers C2 on O.CustomerID = C2.CustomerId  
WHERE C2.Region = C1.Region  
GROUP BY C2.Region, C2.CustomerId  
ORDER BY SUM(OD.UnitPrice \* OD.Quantity \* (1 - OD.Discount)) DESC)**  
**ORDER BY C1.Region**

Here you can see the inner query is a correlated subquery because it references **"C1"**, which is the table alias for the "Northwind.DBO.Customers" table in the outer query. This inner query uses the "Region" value to calculate the top two customers for the region associated with the row being processed from the outer query. If the "CustomerID" of the outer query is one of the top two customers, then the record is placed in the record set to be returned.

Correlated Subquery in the HAVING Clause

Say your organizations wants to run a yearlong incentive program to increase revenue. Therefore, they advertise to your customers that if each order they place, during the year, is over $750 you will provide them a rebate at the end of the year at the rate of $75 per order they place. Below is an example of how to calculate the rebate amount. This example uses a correlated subquery in the HAVING clause to identify the customers that qualify to receive the rebate.

**SELECT C.CustomerID,  
COUNT(\*) \* 75 Rebate  
FROM Northwind.DBO.Customers C  
JOIN Northwind.DBO.Orders O ON C.CustomerID = O.CustomerID  
WHERE DATEPART(yy,OrderDate) = '1998'  
GROUP BY C.CustomerId  
HAVING 750 < ALL**(**SELECT SUM(UnitPrice \* Quantity \* (1 - Discount))  
FROM Northwind.DBO.Orders O  
JOIN Northwind.DBO.[Order Details] OD ON O.OrderID = OD.OrderID  
WHERE O.CustomerID = C.CustomerId  
AND DATEPART(yy,O.OrderDate) = '1998'  
GROUP BY O.OrderId)**

By reviewing this query, you can see the correlated query in the HAVING clause to calculate the total order amount for each customer order. We use the "CustomerID" from the outer query and the year of the order "Datepart(yy,OrderDate)", to help identify the Order records associated with each customer, that were placed the year '1998'. For these associated records I am calculating the total order amount, for each order, by summing up all the "[Order Details]" records, using the following formula: sum(UnitPrice \* Quantity \* (1-Discount)). If each and every order for a customer, for year 1998 has a total dollar amount greater than 750, I then calculate the Rebate amount in the outer query using this formula "Count(\*) \* 75 ".

SQL Server's query engine will only execute the inner correlated subquery in the HAVING clause for those customer records identified in the outer query, or basically only those customer that placed orders in "1998".

Performing an Update Statement Using a Correlated Subquery

A correlated subquery can even be used in an update statement. Here is an example:

create table A(A int, S int)  
create table B(A int, B int)  
  
set nocount on  
insert into A(A) values(1)  
insert into A(A) values(2)  
insert into A(A) values(3)  
insert into B values(1,1)  
insert into B values(2,1)  
insert into B values(2,1)  
insert into B values(3,1)  
insert into B values(3,1)  
insert into B values(3,1)  
 **update A  
set S = (select sum(B)  
from B  
where A.A = A group by A)**select \* from A  
drop table A,B

A S   
----------- -----------  
1 1  
2 2  
3 3

In the query above, I used the correlated subquery to update column A in table A with the sum of column B in table B for rows that have the same value in column A as the row being updated.

Conclusion

A subquery and a correlated subquery are SELECT queries coded inside another query, known as the outer query. The correlated subquery and the subquery help determine the outcome of the result set returned by the complete query. A subquery, when executed independent of the outer query, will return a result set, and is therefore not dependent on the outer query. Where as, a correlated subquery cannot be executed independentl

A subquery is a SELECT query that returns a single value and is nested inside a SELECT, INSERT, UPDATE, or DELETE statement, or inside another subquery. Many Transact-SQL statements that include subqueries can be alternatively formulated as joins. Other questions can be posed only with subqueries.

/\* SELECT statement built using a subquery. \*/

SELECT ProductName

FROM Northwind.dbo.Products

WHERE UnitPrice =

(SELECT UnitPrice

FROM Northwind.dbo.Products

WHERE ProductName = 'Sir Rodney''s Scones')

/\* SELECT statement built using a join that returns

the same result set. \*/

SELECT Prd1.ProductName

FROM Northwind.dbo.Products AS Prd1

JOIN Northwind.dbo.Products AS Prd2

ON (Prd1.UnitPrice = Prd2.UnitPrice)

WHERE Prd2.ProductName = 'Sir Rodney''s Scones'

A subquery nested in the outer SELECT statement has the following components:

A regular SELECT query including the regular select list components.

A regular FROM clause including one or more table or view names.

An optional WHERE clause.

An optional GROUP BY clause.

An optional HAVING clause.

There are three basic types of subqueries.

Those that:

Operate on lists introduced with IN, or those that a comparison operator modified by ANY or ALL.

Are introduced with an unmodified comparison operator and must return a single value.

Are existence tests introduced with EXISTS.

When to Use Joins and Subqueries:

Use a join or a subquery any time that you reference information from multiple tables. Joins and subqueries are often used together in the same query. In many cases, you can solve a data retrieval problem by using a join, a subquery, or both. Here are some guidelines for using joins and queries.

If your report needs data that is from more than one table, then you must perform a join. Whenever multiple tables (or views) are listed in the FROM clause, those tables become joined.

**If you need to combine related information from different rows within a table, then you can join the table with itself.**

**Use subqueries when the result that you want requires more than one query and each subquery provides a subset of the table involved in the query.**

**If a membership question is asked, then a subquery is usually used. If the query requires a NOT EXISTS condition, then you must use a subquery because NOT EXISTS operates only in a subquery; the same principle holds true for the EXISTS condition.**

Many queries can be formulated as joins or subqueries. Although the PROC SQL query optimizer changes some subqueries to joins, a join is generally more efficient to process.

let us run following four queries, all of them are giving exactly same resultset.

USE AdventureWorks  
GO  
-- use of =  
SELECT \*  
FROM HumanResources.Employee E  
WHERE E.EmployeeID = ( SELECT EA.EmployeeID  
FROM HumanResources.EmployeeAddress EA  
WHERE EA.EmployeeID = E.EmployeeID)  
GO

-- use of in  
SELECT \*  
FROM HumanResources.Employee E  
WHERE E.EmployeeID IN ( SELECT EA.EmployeeID  
FROM HumanResources.EmployeeAddress EA  
WHERE EA.EmployeeID = E.EmployeeID)  
GO

-- use of exists  
SELECT \*  
FROM HumanResources.Employee E  
WHERE EXISTS ( SELECT EA.EmployeeID  
FROM HumanResources.EmployeeAddress EA  
WHERE EA.EmployeeID = E.EmployeeID)  
GO

-- Use of Join  
SELECT \*  
FROM HumanResources.Employee E  
INNER JOIN HumanResources.EmployeeAddress EA ON E.EmployeeID = EA.EmployeeID  
GO

**IN:**

Returns true if a specified value matches any value in a subquery or a list.

**Exists:**

Returns true if a subquery contains any rows.

**Join:**

Joins 2 resultsets on the joining column.

**WHAT IS THE DIFFERENCE BETWEEN A SUBQUERY AND A CORRELATED QUERY?**

A correlated subquery is a SELECT statement nested inside another T-SQL statement, which contains a reference to one or more columns in the outer query. Therefore, **the correlated subquery can be said to be dependent on the outer query.** This is the main difference between a correlated subquery and just a plain subquery. **A plain subquery is not dependent on the outer query, can be run independently of the outer query, and will return a result set**. A correlated subquery, since it is dependent on the outer query will return a syntax errors if it is run by itself.   
  
A correlated subquery will be executed many times while processing the SQL statement that contains the correlated subquery. The correlated subquery will be run once for each candidate row selected by the outer query. The outer query columns, referenced in the correlated subquery, are replaced with values from the candidate row prior to each execution. Depending on the results of the execution of the correlated subquery, it will determine if the row of the outer query is returned in the final result set.

Many queries can be evaluated by executing the subquery once and substituting the resulting value or values into the WHERE clause of the outer query. In queries that include a correlated subquery (also known as a repeating subquery), the subquery depends on the outer query for its values. This means that the subquery is executed repeatedly, once for each row that might be selected by the outer query.  
  
This query finds the names of all authors who earn 100 percent of the shared royalty (royaltyper) on a book.  
  
USE pubs  
SELECT au\_lname, au\_fname  
FROM authors  
WHERE 100 IN  
   (SELECT royaltyper  
   FROM titleauthor  
   WHERE titleauthor.au\_ID = authors.au\_id)  
  
Here is the result set:  
  
au\_lname au\_fname   
---------------------------------------- --------------------   
White Johnson   
Green Marjorie   
Carson Cheryl   
Straight Dean   
Locksley Charlene   
Blotchet-Halls Reginald   
del Castillo Innes   
Panteley Sylvia   
Ringer Albert

A ***correlated subquery*** is an inner subquery which is referenced by the main outer query such that the inner query is considered as being executed repeatedly.

Example:  
----Example of Correlated Subqueries  
USE AdventureWorks;  
GO  
SELECT e.EmployeeID  
FROM HumanResources.Employee e  
WHERE e.ContactID IN  
(  
SELECT c.ContactID  
FROM Person.Contact c  
WHERE MONTH(c.ModifiedDate) = MONTH(e.ModifiedDate)  
)  
GO  
  
A ***noncorrelated subquery*** is subquery that is independent of the outer query and it can executed on its own without relying on main outer query.

Example:  
----Example of Noncorrelated Subqueries  
USE AdventureWorks;  
GO  
SELECT e.EmployeeID  
FROM HumanResources.Employee e  
WHERE e.ContactID IN  
(  
SELECT c.ContactID  
FROM Person.Contact c  
WHERE c.Title = 'Mr.'  
)  
GO  
  
Both of above subqueries can be written using Joins, Exists, In clauses.

A subquery is subject to the following restrictions:

* The select list of a subquery introduced with a comparison operator can include only one expression or column name (except that EXISTS and IN operate on SELECT \* or a list, respectively).
* If the WHERE clause of an outer query includes a column name, it must be join-compatible with the column in the subquery select list.
* The ntext, text, and image data types cannot be used in the select list of subqueries.
* Because they must return a single value, subqueries introduced by an unmodified comparison operator (one not followed by the keyword ANY or ALL) cannot include GROUP BY and HAVING clauses.
* The DISTINCT keyword cannot be used with subqueries that include GROUP BY.
* The COMPUTE and INTO clauses cannot be specified.
* ORDER BY can only be specified when TOP is also specified.
* A view created by using a subquery cannot be updated.
* The select list of a subquery introduced with EXISTS, by convention, has an asterisk (\*) instead of a single column name. The rules for a subquery introduced with EXISTS are the same as those for a standard select list, because a subquery introduced with EXISTS creates an existence test and returns TRUE or FALSE, instead of data.

**JOINS (LEFT, RIGHT, INNER, SELF, CROSS & MULTIPLE JOINS):**

**Difference between FULL OUTER JOIN and UNION**

The UNION Statement will retrieve all the lines from both tables. If in table A you have 10 lines and in table B you have 30 lines, it will return 40 lines. **But in full outer join, you will only get 30 lines, and also in full outer join, you need a join on the 2 tables.**

**UNION:**

SELECT A.\* FROM

(SELECT HCT.creation\_date,HCT.username,HCT.cusno,HCT.payuser,

HCT.caplno,HCT.ctype,SUM(HCT.amt\_recvd) AS 'SumOf\_Amt\_Received',

COUNT(HCT.linenuma) AS 'CountOf\_LineNum',HCT.batch

FROM CashTrans.dbo.HCashTrans\_LMS HCT WITH (NOLOCK)

WHERE creation\_date ='20120530'

GROUP BY HCT.creation\_date,HCT.cusno,HCT.username,HCT.payuser,

HCT.caplno,HCT.ctype,HCT.batch

UNION ALL select CT.creation\_date,CT.username,CT.cusno,CT.payuser,

CT.caplno,CT.ctype,SUM(CT.amt\_recvd) AS 'SumOf\_Amt\_Received',

COUNT(CT.linenuma) AS 'CountOf\_LineNum',CT.batch

FROM CashTrans.dbo.CashTrans\_LMS CT WITH (NOLOCK)

WHERE creation\_date ='20120530'

GROUP BY CT.creation\_date,CT.cusno,CT.username,CT.payuser,

CT.caplno,CT.ctype,CT.batch) A

ORDER BY A.username,A.cusno

Assume we have the following two tables. **Table A** is on the left, and **Table B** is on the right. We'll populate them with four records each.

id name id name

-- ---- -- ----

1 Pirate 1 Rutabaga

2 Monkey 2 Pirate

3 Ninja 3 Darth Vader

4 Spaghetti 4 Ninja

Let's join these tables by the name field in a few different ways and see if we can get a conceptual match to those nifty Venn diagrams.

|  |  |
| --- | --- |
| SELECT \* FROM TableA  **INNER JOIN** TableB  ON TableA.name = TableB.name  id name id name  -- ---- -- ----  1 Pirate 2 Pirate  3 Ninja 4 Ninja  **Inner join** produces only the set of records that match in both Table A and Table B. | Venn diagram of SQL inner join |
| SELECT \* FROM TableA  **FULL OUTER JOIN** TableB  ON TableA.name = TableB.name  id name id name  -- ---- -- ----  1 Pirate 2 Pirate  2 Monkey null null  3 Ninja 4 Ninja  4 Spaghetti null null  null null 1 Rutabaga  null null 3 Darth Vader  **Full outer join** produces the set of all records in Table A and Table B, with matching records from both sides where available**. If there is no match, the missing side will contain null.** | Venn diagram of SQL cartesian join |
| SELECT \* FROM TableA  **LEFT OUTER JOIN** TableB  ON TableA.name = TableB.name  id name id name  -- ---- -- ----  1 Pirate 2 Pirate  2 Monkey null null  3 Ninja 4 Ninja  4 Spaghetti null null  **Left outer join** produces a complete set of records from Table A, with the matching records (where available) in Table B. If there is no match, the right side will contain null. | Venn diagram of SQL left join |
| SELECT \* FROM TableA  LEFT OUTER JOIN TableB  ON TableA.name = TableB.name  **WHERE TableB.id IS null**  id name id name  -- ---- -- ----  2 Monkey null null  4 Spaghetti null null  To produce the set of records only in Table A, but not in Table B, we perform the same left outer join, then **exclude the records we don't want from the right side via a where clause**. | join-left-outer.png |
| SELECT \* FROM TableA  FULL OUTER JOIN TableB  ON TableA.name = TableB.name  **WHERE TableA.id IS null**  **OR TableB.id IS null**  id name id name  -- ---- -- ----  2 Monkey null null  4 Spaghetti null null  null null 1 Rutabaga  null null 3 Darth Vader  To produce the set of records unique to Table A and Table B, we perform the same full outer join, then **exclude the records we don't want from both sides via a where clause**. | join-outer.png |

There's also a cartesian product or **cross join**, which as far as I can tell, can't be expressed as a Venn diagram:

SELECT \* FROM TableA

**CROSS JOIN** TableB

This joins "everything to everything", resulting in 4 x 4 = 16 rows, far more than we had in the original sets. If you do the math, you can see why this is a *very* dangerous join to run against large tables.

**LEFT/RIGHT Outer Join**

OUTER JOIN is used to join two tables even if there is not a match. An OUTER JOIN can be used to return a list of all the customers and the orders even if no orders have been placed for some of the customers. A keyword, RIGHT or LEFT, is used to specify which side of the join returns all possible rows. I like using LEFT because it makes sense to me to list the most important table first. Except for one example demonstrating RIGHT OUTER JOIN, this article will use left joins. Just a note: the keywords INNER and OUTER are optional.

The next example returns a list of all the customers and the SalesOrderID for the orders that have been placed, if any.

SELECT c.CustomerID, s.SalesOrderID

FROM Sales.Customer c

LEFT OUTER JOIN Sales.SalesOrderHeader s ON c.CustomerID = s.CustomerID

It uses the LEFT keyword because the Sales.Customer table is located on the left side and we want all rows returned from that table even if there is no match in the Sales.SalesOrderHeader table. This is an important point. Notice also that the CustomerID column is the primary key of the Sales.Customer table and a foreign key in the Sales.SalesOrderHeader table. This means that there must be a valid customer for every order placed. Writing a query that returns all orders and the customers if they match doesn’t make sense. The LEFT table should always be the primary key table when performing a LEFT OUTER JOIN.

Occasionally, you will need to be more specific. How can you find all the customers who have not placed an order in 2002? There are several ways to solve this problem.

My favorite technique to solve this problem is much simpler. Additional criteria, in this case filtering on the OrderDate, can be added to the join condition. The query joins all customers to the orders placed in 2002. Then the results are restricted to those where there is no match. This query will return exactly the same results as the previous, more complicated query:

SELECT c.CustomerID, s.SalesOrderID

FROM Sales.Customer c

LEFT OUTER JOIN Sales.SalesOrderHeader s ON c.CustomerID = s.CustomerID

and s.OrderDate between '1/1/2002' and '12/31/2002'

WHERE s.SalesOrderID IS NULL

**CROSS JOIN**

There's also a cartesian product or **cross join**, which as far as I can tell, can't be expressed as a Venn diagram:

SELECT \* FROM TableA

**CROSS JOIN** TableB

This joins "everything to everything", resulting in 4 x 4 = 16 rows, far more than we had in the original sets. If you do the math, you can see why this is a *very* dangerous join to run against large tables.

“Cartesian” or “Cross” Join. This join is very simple to create (especially by accident) although it's rarely used.

**SELF JOIN**

Self Join has always been a note-worthy case. It is interesting to ask questions on self join in a room full of developers. I often ask – if there are three kind of joins, i.e.- Inner Join, Outer Join and Cross Join; what type of join is Self Join? The usual answer is that it is an Inner Join. **In fact, it can be classified under any type of join.**

Self Join in SQL Server 2000/2005 helps in retrieving the records having some relation or similarity with other records in the same database table. A common example of employees table can do more clearly about the self join in sql. **Self join in sql means joining the single table to itself.** It creates the partial view of the single table and retrieves the related records. You can use aliases for the same table to set a self join between the single table and retrieve the records satisfying the condition in where clause.

For self join in sql you can try the following example:

**Create table employees:**

|  |  |  |
| --- | --- | --- |
| **emp\_id** | **emp\_name** | **emp\_manager\_id** |
| 1 | John | Null |
| 2 | Tom | 1 |
| 3 | Smith | 1 |
| 4 | Albert | 2 |
| 5 | David | 2 |
| 6 | Murphy | 5 |
| 7 | Petra | 5 |

Now to get the names of managers from the above single table you can use sub queries or simply the self join.

**Self Join SQL Query to get the names of manager and employees:**

select e1.emp\_name 'manager',e2.emp\_name 'employee'  
from employees e1 join employees e2  
on e1.emp\_id=e2.emp\_manager\_id

**Result:**

|  |  |
| --- | --- |
| **manager** | **employee** |
| John | Tom |
| John | Smith |
| Tom | Albert |
| Tom | David |
| David | Murphy |
| David | Petra |

**Understanding the Self Join Example**

In the above self join query, **employees** table is joined with itself using **table aliases e1 and e2**. This creates the two views of a single table.

from employees e1 join employees e2  
on e1.emp\_id=e2.emp\_manager\_id

Here e.emp\_manager\_id passes the manager id from the 2nd view to the first aliased e1 table to get the names of managers.

[**Using GROUP BY to avoid self-joins**](http://weblogs.sqlteam.com/jeffs/archive/2007/06/12/using-group-by-to-avoid-self-joins.aspx)

Sometimes, it appears that a necessary solution to common SQL problems is to join a table to itself.   While self-joins do indeed have their place, and can be very powerful and useful, often times there is a much easier and more efficient way to get the results you need when querying a single table.  
  
For example, suppose we have "budget" and "actual" transactions together in a single table like this:

create table GLTrans   
(  
    TranID int primary key,  
    Company varchar(10) not null,  
    Type varchar(10) not null,  
    TimePeriod int not null,  
    Amount money not null  
)  
  
insert into GLTrans  
select 1,'Company A', 'Actual', 1,20 union all  
select 2,'Company A', 'Budget', 1,15 union all  
select 3,'Company A', 'Actual', 2,30 union all  
select 4,'Company A', 'Budget', 2,50 union all  
select 5,'Company B', 'Actual', 1,25 union all  
select 6,'Company B', 'Budget', 1,30 union all  
select 8,'Company B', 'Budget', 2,25

If we'd like to see a variance (Actual-Budget) calculation for all companies, it is often thought that a **self-join** is necessary, and the SELECT might be written like this:

select a.Company, a.Actual, b.Budget, a.Actual-b.Budget as Variance   
from  
    (select company, sum(Amount) as Actual  
    from GLTrans   
    where type='Actual'  
    group by company  
    ) a  
inner join  
    (select company, sum(Amount) as Budget  
    from GLTrans   
    where type='Budget'  
    group by company  
    ) b  
    on a.company = b.company  
  
Company    Actual                Budget                Variance  
---------- --------------------- --------------------- ---------------------  
Company A  50.00                 65.00                 -15.00  
Company B  25.00                 55.00                 -30.00  
  
(2 row(s) affected)

Basically, we have one derived table that retrieves the actual per company, another that retrieves the budget per company, and we join them together.  However, in this case, there is no need for a join at all!  All of the data we need is already in one table, and a simple GROUP BY combined with CASE does the trick perfectly, and also much more efficiently:

select x.\*, Actual-Budget as Variance  
from  
(  
    select   
        company,   
        sum(case when type='Actual' then amount else 0 end) as Actual,  
        sum(case when type='Budget' then amount else 0 end) as Budget  
    from  
        GLTrans  
    group by   
        company  
) x  
  
company    Actual                Budget                Variance  
---------- --------------------- --------------------- ---------  
Company A  50.00                 65.00                 -15.00  
Company B  25.00                 55.00                 -30.00

The following example uses a self-join to find the products that are supplied by more than one vendor.

Because this query involves a join of the ProductVendor table with itself, the ProductVendor table appears in two roles. To distinguish these roles, you must give the ProductVendor table two different aliases (pv1 and pv2) in the FROM clause. These aliases are used to qualify the column names in the rest of the query. This is an example of the self-join Transact-SQL statement:

USE AdventureWorks2008R2;

GO

SELECT DISTINCT pv1.ProductID, pv1.VendorID

FROM Purchasing.ProductVendor pv1

    INNER JOIN Purchasing.ProductVendor pv2

    ON pv1.ProductID = pv2.ProductID

        AND pv1.VendorID <> pv2.VendorID

ORDER BY pv1.ProductID

## Background

I'm a pretty visual person. Things seem to make more sense as a picture. I looked all over the Internet for a good graphical representation of SQL JOINs, but I couldn't find any to my liking. Some had good diagrams but lacked completeness (they didn't have all the possible JOINs), and some were just plain terrible. So, I decided to create my own and write an article about it.

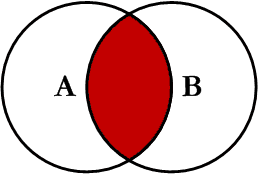
## Using the code

I am going to discuss seven different ways you can return data from two relational tables. I will be excluding cross Joins and self referencing Joins. The seven Joins I will discuss are shown below:

1. **INNER JOIN**
2. **LEFT JOIN**
3. **RIGHT JOIN**
4. **OUTER JOIN**
5. **LEFT JOIN EXCLUDING INNER JOIN**
6. **RIGHT JOIN EXCLUDING INNER JOIN**
7. **OUTER JOIN EXCLUDING INNER JOIN**

For the sake of this article, I'll refer to 5, 6, and 7 as LEFT EXCLUDING JOIN, RIGHT EXCLUDING JOIN, and OUTER EXCLUDING JOIN, respectively. Some may argue that 5, 6, and 7 are not really joining the two tables, but for simplicity, I will still refer to these as Joins because you use a SQL Join in each of these queries (but exclude some records with a WHERE clause).

#### Inner JOIN



This is the simplest, most understood Join and is the most common. This query will return all of the records in the left table (table A) that have a matching record in the right table (table B). This Join is written as follows:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins)

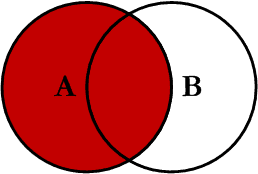
SELECT <select\_list>

FROM Table\_A A

INNER JOIN Table\_B B

ON A.Key = B.Key

#### Left JOIN



This query will return all of the records in the left table (table A) regardless if any of those records have a match in the right table (table B). It will also return any matching records from the right table. This Join is written as follows:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins)

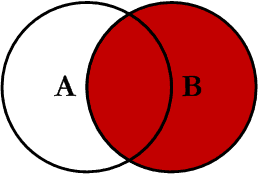
SELECT <select\_list>

FROM Table\_A A

LEFT JOIN Table\_B B

ON A.Key = B.Key

#### Right JOIN



This query will return all of the records in the right table (table B) regardless if any of those records have a match in the left table (table A). It will also return any matching records from the left table. This Join is written as follows:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins)

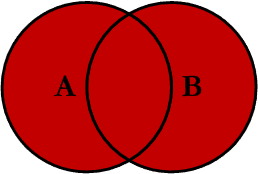
SELECT <select\_list>

FROM Table\_A A

RIGHT JOIN Table\_B B

ON A.Key = B.Key

#### Outer JOIN



This Join can also be referred to as a FULL OUTER JOIN or a FULL JOIN. This query will return all of the records from both tables, joining records from the left table (table A) that match records from the right table (table B). This Join is written as follows:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins)

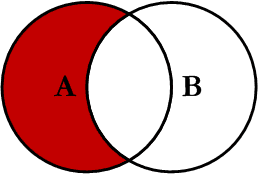
SELECT <select\_list>

FROM Table\_A A

FULL OUTER JOIN Table\_B B

ON A.Key = B.Key

#### Left Excluding JOIN



This query will return all of the records in the left table (table A) that do not match any records in the right table (table B). This Join is written as follows:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins)

SELECT <select\_list>

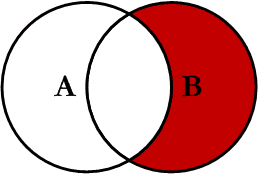
FROM Table\_A A

LEFT JOIN Table\_B B

ON A.Key = B.Key

WHERE B.Key IS NULL

#### Right Excluding JOIN



This query will return all of the records in the right table (table B) that do not match any records in the left table (table A). This Join is written as follows:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins)

SELECT <select\_list>

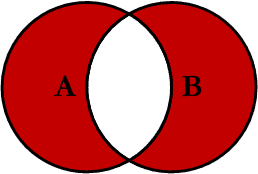
FROM Table\_A A

RIGHT JOIN Table\_B B

ON A.Key = B.Key

WHERE A.Key IS NULL

#### Outer Excluding JOIN



This query will return all of the records in the left table (table A) and all of the records in the right table (table B) that do not match. I have yet to have a need for using this type of Join, but all of the others, I use quite frequently. This Join is written as follows:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins)

SELECT <select\_list>

FROM Table\_A A

FULL OUTER JOIN Table\_B B

ON A.Key = B.Key

WHERE A.Key IS NULL OR B.Key IS NULL

#### Examples

Suppose we have two tables, Table\_A and Table\_B. The data in these tables are shown below:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins)

**TABLE\_A**

PK Value

---- ----------

1 FOX

2 COP

3 TAXI

6 WASHINGTON

7 DELL

5 ARIZONA

4 LINCOLN

10 LUCENT

**TABLE\_B**

PK Value

---- ----------

1 TROT

2 CAR

3 CAB

6 MONUMENT

7 PC

8 MICROSOFT

9 APPLE

11 SCOTCH

The results of the seven Joins are shown below:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins)

**-- INNER JOIN**

SELECT A.PK AS A\_PK, A.Value AS A\_Value,

B.Value AS B\_Value, B.PK AS B\_PK

FROM Table\_A A

INNER JOIN Table\_B B

ON A.PK = B.PK

A\_PK A\_Value B\_Value B\_PK

---- ---------- ---------- ----

1 FOX TROT 1

2 COP CAR 2

3 TAXI CAB 3

6 WASHINGTON MONUMENT 6

7 DELL PC 7

(5 row(s) affected)

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins)

**-- LEFT JOIN**

SELECT A.PK AS A\_PK, A.Value AS A\_Value,

B.Value AS B\_Value, B.PK AS B\_PK

FROM Table\_A A

LEFT JOIN Table\_B B

ON A.PK = B.PK

A\_PK A\_Value B\_Value B\_PK

---- ---------- ---------- ----

1 FOX TROT 1

2 COP CAR 2

3 TAXI CAB 3

4 LINCOLN NULL NULL

5 ARIZONA NULL NULL

6 WASHINGTON MONUMENT 6

7 DELL PC 7

10 LUCENT NULL NULL

(8 row(s) affected)

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins)

**-- RIGHT JOIN**

SELECT A.PK AS A\_PK, A.Value AS A\_Value,

B.Value AS B\_Value, B.PK AS B\_PK

FROM Table\_A A

RIGHT JOIN Table\_B B

ON A.PK = B.PK

A\_PK A\_Value B\_Value B\_PK

---- ---------- ---------- ----

1 FOX TROT 1

2 COP CAR 2

3 TAXI CAB 3

6 WASHINGTON MONUMENT 6

7 DELL PC 7

NULL NULL MICROSOFT 8

NULL NULL APPLE 9

NULL NULL SCOTCH 11

(8 row(s) affected)

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins)

**-- OUTER JOIN**

SELECT A.PK AS A\_PK, A.Value AS A\_Value,

B.Value AS B\_Value, B.PK AS B\_PK

FROM Table\_A A

FULL OUTER JOIN Table\_B B

ON A.PK = B.PK

A\_PK A\_Value B\_Value B\_PK

---- ---------- ---------- ----

1 FOX TROT 1

2 COP CAR 2

3 TAXI CAB 3

6 WASHINGTON MONUMENT 6

7 DELL PC 7

NULL NULL MICROSOFT 8

NULL NULL APPLE 9

NULL NULL SCOTCH 11

5 ARIZONA NULL NULL

4 LINCOLN NULL NULL

10 LUCENT NULL NULL

(11 row(s) affected)

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins)

**-- LEFT EXCLUDING JOIN**

SELECT A.PK AS A\_PK, A.Value AS A\_Value,

B.Value AS B\_Value, B.PK AS B\_PK

FROM Table\_A A

LEFT JOIN Table\_B B

ON A.PK = B.PK

WHERE B.PK IS NULL

A\_PK A\_Value B\_Value B\_PK

---- ---------- ---------- ----

4 LINCOLN NULL NULL

5 ARIZONA NULL NULL

10 LUCENT NULL NULL

(3 row(s) affected)

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins)

**-- RIGHT EXCLUDING JOIN**

SELECT A.PK AS A\_PK, A.Value AS A\_Value,

B.Value AS B\_Value, B.PK AS B\_PK

FROM Table\_A A

RIGHT JOIN Table\_B B

ON A.PK = B.PK

WHERE A.PK IS NULL

A\_PK A\_Value B\_Value B\_PK

---- ---------- ---------- ----

NULL NULL MICROSOFT 8

NULL NULL APPLE 9

NULL NULL SCOTCH 11

(3 row(s) affected)

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins)

**-- OUTER EXCLUDING JOIN**

SELECT A.PK AS A\_PK, A.Value AS A\_Value,

B.Value AS B\_Value, B.PK AS B\_PK

FROM Table\_A A

FULL OUTER JOIN Table\_B B

ON A.PK = B.PK

WHERE A.PK IS NULL

OR B.PK IS NULL

A\_PK A\_Value B\_Value B\_PK

---- ---------- ---------- ----

NULL NULL MICROSOFT 8

NULL NULL APPLE 9

NULL NULL SCOTCH 11

5 ARIZONA NULL NULL

4 LINCOLN NULL NULL

10 LUCENT NULL NULL

(6 row(s) affected)

Note on the OUTER JOIN that the inner joined records are returned first, followed by the right joined records, and then finally the left joined records (at least, that's how my Microsoft SQL Server did it; this, of course, is without using any ORDER BY statement).

You can visit the Wikipedia article for more info [here](http://en.wikipedia.org/wiki/Sql_join) (however, the entry is not graphical).

I've also created a cheat sheet that you can print out if needed. If you right click on the image below and select "Save Target As...", you will download the full size image.

[](http://www.codeproject.com/KB/database/Visual_SQL_Joins/Visual_SQL_JOINS_orig.jpg)

**MULTIPLE JOINS**

Here's a simple, three-step formula for creating effective joins:

1. Form a question that defines the data set you want to see
2. Find the relevant tables and their joining fields
3. Join one table at a time to the next

**USE AdventureWorks;**

**DECLARE @Year int, @Month int**

**SET @Year = 2004; SET @Month=1;**

**SELECT   SOH.SalesOrderNumber               AS SON,**

**SOH.PurchaseOrderNumber            AS PO,**

**S.Name                             AS Store,**

**Convert(VARCHAR,SOH.OrderDate,110) AS OrderDate,**

**Convert(VARCHAR,SOH.ShipDate,110)  AS ShipDate,**

**'$' + Convert(VARCHAR,TotalDue,1)  AS [Total Due],**

**C.FirstName + ' ' + C.LastName     AS SalesStaff,**

**SM.Name                            AS ShpngMethod,**

**BA.AddressLine1                    AS BlngAddress1,**

**Isnull(BA.AddressLine2,'')         AS BlngAddress2,**

**BA.City                            AS BlngCity,**

**BSP.Name                           AS BlngStateProvince,**

**BA.PostalCode                      AS BlngPostalCode,**

**BCR.Name                           AS BlngCountryRegion,**

**SA.AddressLine1                    AS ShpngAddress1,**

**Isnull(SA.AddressLine2,'')         AS ShpngAddress2,**

**SA.City                            AS ShpngCity,**

**SSP.Name                           AS ShpngStateProvince,**

**SA.PostalCode                      AS ShpngPostalCode,**

**SCR.Name                           AS ShpngCountryRegion,**

**CC.FirstName + ' ' + CC.LastName   AS CustomerContact,**

**CC.Phone                           AS CustomerPhone,**

**SOH.AccountNumber**

**FROM Person.Address SA**

**INNER JOIN Person.StateProvince SSP**

**ON SA.StateProvinceID = SSP.StateProvinceID**

**INNER JOIN Person.CountryRegion SCR**

**ON SSP.CountryRegionCode = SCR.CountryRegionCode**

**INNER JOIN Sales.SalesOrderHeader SOH**

**INNER JOIN Person.Contact CC**

**ON SOH.ContactID = CC.ContactID**

**INNER JOIN Person.Address BA**

**INNER JOIN Person.StateProvince BSP**

**ON BA.StateProvinceID = BSP.StateProvinceID**

**INNER JOIN Person.CountryRegion BCR**

**ON BSP.CountryRegionCode = BCR.CountryRegionCode**

**ON SOH.BillToAddressID = BA.AddressID**

**ON SA.AddressID = SOH.ShipToAddressID**

**INNER JOIN Person.Contact C**

**INNER JOIN HumanResources.Employee E**

**ON C.ContactID = E.ContactID**

**ON SOH.SalesPersonID = E.EmployeeID**

**INNER JOIN Purchasing.ShipMethod SM**

**ON SOH.ShipMethodID = SM.ShipMethodID**

**INNER JOIN Sales.Store S**

**ON SOH.CustomerID = S.CustomerID**

**WHERE    Year(SOH.OrderDate) = @Year**

**AND Month(SOH.OrderDate) = @Month**

**ORDER BY Store,**

**OrderDate DESC**

**GO**

**/\***

**SON         PO                Store                         OrderDate**

**SO61257     PO3741176337      Activity Center               01-01-2004**

**SO61256     PO1421187796      All Cycle Shop                01-01-2004**

**SO61251     PO6380165323      All Seasons Sports Supply     01-01-2004**

**SO61263     PO5452121402      Amalgamated Parts Shop        01-01-2004**

**....**

**\*/**

**INDEX (Clustered, Non-Clustered):**

### Comparing the two ways to create indexes

You can create indexes on tables either by using the **create index** statement (described in this chapter) or by using the **unique** or **primary key** integrity constraints of the **create table** command. However, **integrity constraints are limited in the following ways**:

* You cannot create non unique indexes.
* You cannot use the options provided by the **create index** command to tailor how indexes work.
* You can only drop these indexes as a constraint using the **alter table** statement.

If your application requires these features, you should create your indexes using **create index**. Otherwise, the **unique** or **primary key** integrity constraints offer a simpler way to define an index for a table

When to index

Use the following general guidelines:

* If you plan to do manual insertions into the IDENTITY column, create a unique index to ensure that the inserts do not assign a value that has already been used.
* A column that is often accessed in **sorted order**, that is, specified in the **order by** clause, probably should be indexed so that Adaptive Server can take advantage of the indexed order.
* Columns that are regularly used in joins should always be indexed, since the system can perform the join faster if the columns are in sorted order.
* The column that stores the primary key of the table often has a clustered index, especially if it is frequently joined to columns in other tables. Remember, there can be only one clustered index per table.
* A column that is often searched for ranges of values might be a good choice for a clustered index. Once the row with the first value in the range is found, rows with subsequent values are guaranteed to be physically adjacent. A clustered index does not offer as much of an advantage for searches on single values.

When not to index

In some cases, indexes are not useful:

* **Columns that are seldom or never referenced in queries do not benefit from indexes, since the system seldom has to search for rows on the basis of values in these columns.**
* **Columns that can have only two or three values, for example, "male" and "female" or "yes" and "no", get no real advantage from indexes.**

Indexes help Adaptive Server locate data. They speed up data retrieval by pointing to the location of a table column's data on disk. For example, suppose you need to run frequent queries using the identification numbers of stores in the *stores* table. To prevent Adaptive Server from having to search through each row in the table--which can be time-consuming if the *stores* table contains millions of rows--you could create the following index, entitled *stor\_id\_ind*:

**create index stor\_id\_ind**

**on stores (stor\_id)**

The *stor\_id\_ind* index goes into effect automatically the next time you query the *stor\_id* column in *stores*. In other words, indexes are transparent to users. SQL includes no syntax for referring to an index in a query. You can only create or drop indexes from a table; Adaptive Server decides whether to use the indexes for each query submitted for that table. As the data in a table changes over time, Adaptive Server may change the table's indexes to reflect those changes. Again, these changes are transparent to users; Adaptive Server handles this task on its own.

Adaptive Server supports the following types of indexes:

* Composite indexes - these indexes involve more than one column. Use this type of index when two or more columns are best searched as a unit because of their logical relationship.
* Unique indexes - these indexes do not permit any two rows in the specified columns to have the same value. Adaptive Server checks for duplicate values when the index is created (if data already exists) and each time data is added.
* Clustered or nonclustered indexes - clustered indexes force Adaptive Server to continually sort and re-sort the rows of a table so that their physical order is always the same as their logical (or indexed) order. You can have only one clustered index per table. Nonclustered indexes do not require the physical order of rows to be the same as their indexed order. Each nonclustered index can provide access to the data in a different sort order.

1)      Script out the CREATE statements for the table including the CREATE statements for the indexes. (Always have a backup plan, if it helps take a backup of the database.

2)      Drop all the non clustered indexes.

            DROP INDEX [AK\_Department\_Name] ON [HumanResources].[Department]

WITH ( ONLINE = OFF )

3)      Drop the clustered index.

ALTER TABLE [HumanResources].[Department] DROP CONSTRAINT [PK\_Department\_DepartmentID]

4)      Create the new clustered index

ALTER TABLE [HumanResources].[Department] ADD  CONSTRAINT [PK\_Department\_DepartmentID] PRIMARY KEY CLUSTERED

(

            [DepartmentID] ASC

)WITH (PAD\_INDEX  = OFF, STATISTICS\_NORECOMPUTE  = OFF, SORT\_IN\_TEMPDB = OFF, IGNORE\_DUP\_KEY = OFF, ONLINE = OFF, ALLOW\_ROW\_LOCKS  = ON, ALLOW\_PAGE\_LOCKS  = ON) ON [PRIMARY]

Do not forget to mention the new filegroup for the primary key index.

5)      Create all the remaining non clustered indexes and specify the index filegroup.

CREATE NONCLUSTERED INDEX [AK\_Department\_Name] ON [HumanResources].[Department]

(

            [Name] ASC

)WITH (PAD\_INDEX  = OFF, STATISTICS\_NORECOMPUTE  = OFF, SORT\_IN\_TEMPDB = OFF, IGNORE\_DUP\_KEY = OFF, DROP\_EXISTING = OFF, ONLINE = OFF, ALLOW\_ROW\_LOCKS  = ON,

      ALLOW\_PAGE\_LOCKS  = ON) ON [INDEXES]

**UNIQUE CONSTRAINT**

EXEC sp\_helpconstraint [HumanResources.Employee]

## SQL SERVER – Difference Between Unique Index vs Unique Constraint

Unique Index and Unique Constraint are the same. They achieve same goal. SQL Performance is same for both.

**Add Unique Constraint**  
ALTER TABLEdbo.<tablename> ADD CONSTRAINT  
<namingconventionconstraint> UNIQUE NONCLUSTERED  
(  
< columnname>  
) ON [PRIMARY]

**Add Unique Index**  
CREATE UNIQUE NONCLUSTERED INDEX  
<namingconventionconstraint> ON dbo.<tablename>  
(  
< columnname>  
) ON [PRIMARY]

There is no difference between Unique Index and Unique Constraint. Even though syntax are different the effect is the same. Unique Constraint creates Unique Index to maintain the constraint to prevent duplicate keys. Unique Index or Primary Key Index are physical structure that maintain uniqueness over some combination of columns across all rows of a table. It is a convenient way to enforce a Unique Constraint for SQL Server.

# Placing Indexes on Filegroups

**SQL Server 2008 R2**

As you develop your index design strategy, you should consider the placement of the indexes on the filegroups associated with the database. Careful selection of the filegroup or partition scheme can improve query performance.

By default, indexes are stored in the same filegroup as the base table on which the index is created. A nonpartitioned clustered index and the base table always reside in the same filegroup. However, you can do the following:

* Create nonclustered indexes on a filegroup other than the filegroup of the base table.
* Partition clustered and nonclustered indexes to span multiple filegroups.
* Move a table from one filegroup to another by dropping the clustered index and specifying a new filegroup or partition scheme in the MOVE TO clause of the DROP INDEX statement or by using the CREATE INDEX statement with the DROP\_EXISTING clause.

By creating the nonclustered index on a different filegroup, you can achieve performance gains if the filegroups are using different physical drives with their own controllers. Data and index information can then be read in parallel by the multiple disk heads. For example, if **Table\_A** on filegroup **f1** and **Index\_A** on filegroup **f2** are both being used by the same query, performance gains can be achieved because both filegroups are being fully used without contention. However, if **Table\_A** is scanned by the query but **Index\_A** is not referenced, only filegroup **f1** is used. This creates no performance gain.

Because you cannot predict what type of access will occur and when it will occur, it could be a better decision to spread your tables and indexes across all filegroups. This would guarantee that all disks are being accessed because all data and indexes are spread evenly across all disks, regardless of which way the data is accessed. This is also a simpler approach for system administrators.

[Partitions Across Multiple Filegroups](javascript:void(0))

You can also consider partitioning clustered and nonclustered indexes across multiple filegroups. Partitioned indexes are partitioned horizontally, or by row, based on a partition function. The partition function defines how each row is mapped to a set of partitions based on the values of certain columns, called partitioning columns. A partition scheme specifies the mapping of the partitions to a set of filegroups.

Partitioning an index can provide the following benefits:

* Provide scalable systems that make large indexes more manageable. OLTP systems, for example, can implement partition-aware applications that deal with large indexes.
* Make queries run faster and more efficiently. When queries access several partitions of an index, the query optimizer can process individual partitions at the same time and exclude partitions that are not affected by the query.

# Partitioned Table and Index Concepts

**SQL Server 2008 R2**

Partitioning makes large tables or indexes more manageable, because partitioning enables you to manage and access subsets of data quickly and efficiently, while maintaining the integrity of a data collection. By using partitioning, an operation such as loading data from an OLTP to an OLAP system takes only seconds, instead of the minutes and hours the operation takes in earlier versions of SQL Server. Maintenance operations that are performed on subsets of data are also performed more efficiently because these operations target only the data that is required, instead of the whole table.

|  |
| --- |
| **NoteNote** |
| Partitioned tables and indexes are available only on the Enterprise, Developer, and Evaluation editions of SQL Server. |

The data of partitioned tables and indexes is divided into units that can be spread across more than one filegroup in a database. The data is partitioned horizontally, so that groups of rows are mapped into individual partitions. The table or index is treated as a single logical entity when queries or updates are performed on the data. All partitions of a single index or table must reside in the same database.

Partitioned tables and indexes support all the properties and features associated with designing and querying standard tables and indexes, including constraints, defaults, identity and timestamp values, and triggers. Therefore, if you want to implement a partitioned view that is local to one server, you might want to implement a partitioned table instead.

Deciding whether to implement partitioning depends primarily on how large your table is or how large it will become, how it is being used, and how well it is performing against user queries and maintenance operations.

Generally, a large table might be appropriate for partitioning if both of the following are true:

* The table contains, or is expected to contain, lots of data that are used in different ways.
* Queries or updates against the table are not performing as intended, or maintenance costs exceed predefined maintenance periods.

For example, if a current month of data is primarily used for INSERT, UPDATE, DELETE, and MERGE operations while previous months are used primarily for SELECT queries, managing this table may be easier if it is partitioned by month. This benefit can be especially true if regular maintenance operations on the table only have to target a subset of the data. If the table is not partitioned, these operations can consume lots of resources on an entire data set. With partitioning, maintenance operations, such as index rebuilds and defragmentations, can be performed on a single month of write-only data, for example, while the read-only data is still available for online access.

To expand on this example, suppose you want to move one month of read-only data from this table to a data warehouse table for analysis. With partitioning, subsets of data can be separated quickly into staging areas for offline maintenance and then added as partitions to existing partitioned tables, assuming these tables are all in the same database instance. Operations such as these typically take seconds, instead of the minutes or hours they took in previous releases.

Partitioning a table or index might improve query performance if the partitions are designed correctly, based on the types of queries you frequently run and on your hardware configuration. For more information, see [Designing Partitions to Improve Query Performance](http://technet.microsoft.com/en-us/library/ms177411(v=sql.105).aspx).

Partitioning is often used in conjunction with SQL Server Replication. Using partitions may enable you to optimize the performance of transactional replication and merge replication by effectively reducing the amount of data and metadata that has to be managed by the replication system. Replication supports a maximum of 1024 partitions per table. For more information, see [Replicating Partitioned Tables and Indexes](http://technet.microsoft.com/en-us/library/cc280940(v=sql.105).aspx).

To provide an example of how a partitioning solution can be applied in a real-world database, a partitioning scenario that you can implement is available in the AdventureWorks2008R2 sample database. This scenario is explained in [Partitioning in the AdventureWorks2008R2 Sample Database](http://technet.microsoft.com/en-us/library/ms191133(v=sql.105).aspx).

[Partitioning Architecture](javascript:void(0))

In SQL Server, all tables and indexes in a database are considered partitioned, even if they are made up of only one partition. Essentially, partitions form the basic unit of organization in the physical architecture of tables and indexes. This means that the logical and physical architecture of tables and indexes comprised of multiple partitions mirrors that of single-partition tables and indexes. For more information, see [Table and Index Organization](http://technet.microsoft.com/en-us/library/ms189051(v=sql.105).aspx).

# Unique Constraints and Check Constraints

**SQL Server 2012**

UNIQUE constraints and CHECK constraints are two types of constraints that can be used to enforce data integrity in SQL Server tables. These are important database objects.

This topic contains the following sections.

[UNIQUE Constraints](http://technet.microsoft.com/en-us/library/ms187550.aspx#Unique)

[CHECK Constraints](http://technet.microsoft.com/en-us/library/ms187550.aspx#Check)

**Constraints are rules that the SQL Server Database Engine enforces for you. For example, you can use UNIQUE constraints to make sure that no duplicate values are entered in specific columns that do not participate in a primary key. Although both a UNIQUE constraint and a PRIMARY KEY constraint enforce uniqueness, use a UNIQUE constraint instead of a PRIMARY KEY constraint when you want to enforce the uniqueness of a column, or combination of columns, that is not the primary key.**

**Unlike PRIMARY KEY constraints, UNIQUE constraints allow for the value NULL. However, as with any value participating in a UNIQUE constraint, only one null value is allowed per column. A UNIQUE constraint can be referenced by a FOREIGN KEY constraint.**

When a UNIQUE constraint is added to an existing column or columns in the table, by default, the Database Engine examines the existing data in the columns to make sure all values are unique. If a UNIQUE constraint is added to a column that has duplicated values, the Database Engine returns an error and does not add the constraint.

The Database Engine automatically creates a UNIQUE index to enforce the uniqueness requirement of the UNIQUE constraint. Therefore, if an attempt to insert a duplicate row is made, the Database Engine returns an error message that states the UNIQUE constraint has been violated and does not add the row to the table. Unless a clustered index is explicitly specified, a unique, nonclustered index is created by default to enforce the UNIQUE constraint.

Arrow icon used with Back to Top link[[Top]](http://technet.microsoft.com/en-us/library/ms187550.aspx#Intro)

[CHECK Constraints](javascript:void(0))

**CHECK constraints enforce domain integrity by limiting the values that are accepted by one or more columns. You can create a CHECK constraint with any logical (Boolean) expression that returns TRUE or FALSE based on the logical operators. For example, the range of values for a salary column can be limited by creating a CHECK constraint that allows for only data that ranges from $15,000 through $100,000. This prevents salaries from being entered beyond the regular salary range. The logical expression would be the following: salary >= 15000 AND salary <= 100000.**

You can apply multiple CHECK constraints to a single column. You can also apply a single CHECK constraint to multiple columns by creating it at the table level. For example, a multiple-column CHECK constraint could be used to confirm that any row with a **country\_region** column value of **USA** also has a two-character value in the **state** column. This allows for multiple conditions to be checked in one location.

CHECK constraints are similar to FOREIGN KEY constraints in that they control the values that are put in a column. The difference is in how they determine which values are valid: FOREIGN KEY constraints obtain the list of valid values from another table, while CHECK constraints determine the valid values from a logical expression.

|  |
| --- |
| **Caution noteCaution** |
| Constraints that include implicit or explicit data type conversion may cause certain operations to fail. For example, such constraints defined on tables that are sources of partition switching may cause an ALTER TABLE...SWITCH operation to fail. **Avoid data type conversion in constraint definitions.** |

### Limitations of CHECK Constraints

CHECK constraints reject values that evaluate to FALSE. Because null values evaluate to UNKNOWN, their presence in expressions may override a constraint. For example, suppose you place a constraint on an int column **MyColumn** specifying that **MyColumn** can contain only the value 10 (**MyColumn** **=** **10**). If you insert the value NULL into **MyColumn**, the Database Engine inserts NULL and does not return an error.

A CHECK constraint returns TRUE when the condition it is checking is not FALSE for any row in the table. A CHECK constraint works at the row level. If a table that has just been created does not have any rows, any CHECK constraint on this table is considered valid. This situation can produce unexpected results, as in the following example.

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_dddbf33a-497d-4fd8-b179-9af00fe4edd0');)

CREATE TABLE CheckTbl (col1 int, col2 int);

GO

CREATE FUNCTION CheckFnctn()

RETURNS int

AS

BEGIN

DECLARE @retval int

SELECT @retval = COUNT(\*) FROM CheckTbl

RETURN @retval

END;

GO

ALTER TABLE CheckTbl

ADD CONSTRAINT chkRowCount CHECK (dbo.CheckFnctn() >= 1 );

GO

The CHECK constraint being added specifies that there must be at least one row in table CheckTbl. However, because there are no rows in the table against which to check the condition of this constraint, the ALTER TABLE statement succeeds.

CHECK constraints are not validated during DELETE statements. Therefore, executing DELETE statements on tables with certain types of check constraints may produce unexpected results. For example, consider the following statements executed on table CheckTbl.

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_f76d8e91-5e49-44dd-808a-94c2a5b975c5');)

INSERT INTO CheckTbl VALUES (10, 10);

GO

DELETE CheckTbl WHERE col1 = 10;

The DELETE statement succeeds, even though the CHECK constraint specifies that table CheckTbl must have at least 1 row.

Arrow icon used with Back to Top link[[Top]](http://technet.microsoft.com/en-us/library/ms187550.aspx#Intro)

[Related Tasks](javascript:void(0))

|  |  |
| --- | --- |
| **NoteNote** | |
| If the table is published for replication, you must make schema changes using the Transact-SQL statement [ALTER TABLE](http://technet.microsoft.com/en-us/library/ms190273.aspx) or SQL Server Management Objects (SMO). When schema changes are made using the Table Designer or the Database Diagram Designer, it attempts to drop and recreate the table. You cannot drop published objects, therefore the schema change will fail. | |
| **Task** | **Topic** |
| Describes how to create a unique constraint. | [Create Unique Constraints](http://technet.microsoft.com/en-us/library/ms190024.aspx) |
| Describes how to modify a unique constraint. | [Modify Unique Constraints](http://technet.microsoft.com/en-us/library/ms191535.aspx) |
| Describes how to delete a unique constraint. | [Delete Unique Constraints](http://technet.microsoft.com/en-us/library/ms188260.aspx) |
| Describes how to disable a check constraint when a replication agent inserts or updates data in your table. | [Disable Check Constraints for Replication](http://technet.microsoft.com/en-us/library/ms190235.aspx) |
| Describes how to disable a check constraint when data is added to, updated in, or deleted from a table. | [Disable Check Constraints with INSERT and UPDATE Statements](http://technet.microsoft.com/en-us/library/ms190661.aspx) |
| Describes how to change the constraint expression or the options that enable or disable the constraint for specific conditions. | [Modify Check Constraints](http://technet.microsoft.com/en-us/library/ms191273.aspx) |
| Describes how to delete a check constraint. | [Delete Check Constraints](http://technet.microsoft.com/en-us/library/ms187626.aspx) |
| Describes how to view the properties of a check constraint. | [Unique Constraints and Check Constraints](http://technet.microsoft.com/en-us/library/ms187550.aspx) |

Arrow icon used with Back to Top link[[Top]](http://technet.microsoft.com/en-us/library/ms187550.aspx#Intro)

**BRIDGE TABLES:**

**COMPARING TWO TABLES:**

**------------**

**-- SQL SERVER COMPARE 2 TABLES FOR ROW & COLUMN DIFFERENCES**

**------------  
  
-- TEMPLATE - SQL Server T-SQL compare two tables  
SELECT Label='Found IN Table1, NOT IN Table2',\* FROM   
(SELECT \* FROM Table1  
 EXCEPT  
 SELECT  \* FROM Table2) x  
UNION ALL  
SELECT Label='Found IN Table2, NOT IN Table1',\* FROM  
(SELECT  \* FROM Table2  
 EXCEPT  
 SELECT \* FROM Table1) y**

**GO**

**NULL:**

It is not a value; it is a “place holder” for a value.

NULL has certain basic properties:

All SQL data types are NULL-able.  This is one reason why IDENTITY is a table property and not a data type.  The other reason is that a table can have only one IDENTITY column in it.  The count of PHYSICAL insertion attempts (NOT  successes) is not an attribute; it is audit meta-data and has no place in RDBMS.    
  
This is why the first implementations of BIT which were assembly language bits were not a data type.  When BIT became a numeric data type, then things were Kosher. 

NULLs propagate.  If you use a NULL in an expression, then result is a NULL.  In numeric expressions, we had questions about priorities, but propagation is vital.  In particular:  
“NULL / 0” = NULL or “division by zero” error?

**How do I work with NULL?**

NULL which can be tricky to work with and requires special treatment in a number of circumstances. In general, never assume that NULL will behave like a 0 in a numeric data field, the empty string in a CHAR field, and so on.

**COALESCE**

What is the differences between ISNULL and COALESC ?

The following example shows how COALESCE selects the data from the first column that has a non-null value.

USE AdventureWorks2012 ;

GO

SELECT Name, Class, Color, ProductNumber,

COALESCE(Class, Color, ProductNumber) AS FirstNotNull

FROM Production.Product ;

GO

COALESCE(expression1,...n) is equivalent to the following CASE expression:

CASE

WHEN (expression1 IS NOT NULL) THEN expression1

WHEN (expression2 IS NOT NULL) THEN expression2

...

ELSE expressionN

END

ISNULL and COALESCE though equivalent, can behave differently. An expression involving ISNULL with non-null parameters is considered to be NOT NULL, while expressions involving COALESCE with non-null parameters is considered to be NULL.

## SQL ISNULL(), NVL(), IFNULL() and COALESCE() Functions

Look at the following "Products" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **ProductName** | **UnitPrice** | **UnitsInStock** | **UnitsOnOrder** |
| 1 | Jarlsberg | 10.45 | 16 | 15 |
| 2 | Mascarpone | 32.56 | 23 |  |
| 3 | Gorgonzola | 15.67 | 9 | 20 |

Suppose that the "UnitsOnOrder" column is optional, and may contain NULL values.

We have the following SELECT statement:

SELECT ProductName,UnitPrice\*(UnitsInStock+UnitsOnOrder)  
FROM Products

In the example above, if any of the "UnitsOnOrder" values are NULL, the result is NULL.

Microsoft's ISNULL() function is used to specify how we want to treat NULL values.

The NVL(), IFNULL(), and COALESCE() functions can also be used to achieve the same result.

In this case we want NULL values to be zero.

Below, if "UnitsOnOrder" is NULL it will not harm the calculation, because ISNULL() returns a zero if the value is NULL:

**SQL Server / MS Access**

SELECT ProductName,UnitPrice\*(UnitsInStock+ISNULL(UnitsOnOrder,0))  
FROM Products

**Oracle**

Oracle does not have an ISNULL() function. However, we can use the NVL() function to achieve the same result:

SELECT ProductName,UnitPrice\*(UnitsInStock+NVL(UnitsOnOrder,0))  
FROM Products

**MySQL**

MySQL does have an ISNULL() function. However, it works a little bit different from Microsoft's ISNULL() function.

In MySQL we can use the IFNULL() function, like this:

SELECT ProductName,UnitPrice\*(UnitsInStock+IFNULL(UnitsOnOrder,0))  
FROM Products

or we can use the COALESCE() function, like this:

SELECT ProductName,UnitPrice\*(UnitsInStock+COALESCE(UnitsOnOrder,0))  
FROM Products

The **COALESCE** function in SQL returns the first non-NULL expression among its arguments.

It is the same as the following [**CASE**](http://www.1keydata.com/sql/sql-case.html) statement:

**SELECT CASE ("column\_name")  
WHEN "expression 1 is not NULL" THEN "expression 1"  
WHEN "expression 2 is not NULL" THEN "expression 2"  
...  
[ELSE "NULL"]  
END  
FROM "table\_name"**

For examples, say we have the following table,

Table ***Contact\_Info***

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Business\_Phone | Cell\_Phone | Home\_Phone |
| Jeff | 531-2531 | 622-7813 | 565-9901 |
| Laura | NULL | 772-5588 | 312-4088 |
| Peter | NULL | NULL | 594-7477 |

and we want to find out the best way to contact each person according to the following rules:

1. If a person has a business phone, use the business phone number.

2. If a person does not have a business phone and has a cell phone, use the cell phone number.

3. If a person does not have a business phone, does not have a cell phone, and has a home phone, use the home phone number.

We can use the **COALESCE** function to achieve our goal:

**SELECT Name, COALESCE(Business\_Phone, Cell\_Phone, Home\_Phone) Contact\_Phone   
FROM Contact\_Info;**

*Result:*

|  |  |
| --- | --- |
| **Name** | **Contact\_Phone** |
| **Jeff** | **531-2531** |
| **Laura** | **772-5588** |
| **Peter** | **594-7477** |

**Problem**  
Many times people come across the Coalesce function and think that it is just a more powerful form of ISNULL. In actuality, I have found it to be one of the most useful functions with the least documentation. In this tip, I will show you the basic use of Coalesce and also some features you probably never new existed.

**Solution**  
Let's start with the documented use of coalesce. According to MSDN, coalesce returns the first non-null expression among its arguments.

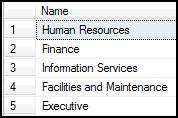
For example,

|  |
| --- |
| SELECT COALESCE(NULL, NULL, NULL, GETDATE()) |

will return the current date. It bypasses the first NULL values and returns the first non-null value.  
  
**Using Coalesce to Pivot**  
If you run the following statement against the AdventureWorks database

|  |
| --- |
| SELECT Name  FROM HumanResources.Department  WHERE (GroupName = 'Executive General and Administration') |

you will come up with a standard result set such as this.



If you want to pivot the data you could run the following command.

|  |
| --- |
| DECLARE @DepartmentName VARCHAR(1000)   SELECT @DepartmentName = COALESCE(@DepartmentName,'') + Name + ';'  FROM HumanResources.Department  WHERE (GroupName = 'Executive General and Administration')   SELECT @DepartmentName AS DepartmentNames |

and get the following result set.

http://www.mssqltips.com/tipimages/1521_image002.jpg

**Using Coalesce to Execute Multiple SQL Statements**  
Once you can pivot data using the coalesce statement, it is now possible to run multiple SQL statements by pivoting the data and using a semicolon to separate the operations. Let's say you want to find the values for any column in the Person schema that has the column name “Name”. If you execute the following script it will give you just that.

|  |
| --- |
| DECLARE @SQL VARCHAR(MAX)   CREATE TABLE #TMP  (Clmn VARCHAR(500),  Val VARCHAR(50))   SELECT @SQL=COALESCE(@SQL,'')+CAST('INSERT INTO #TMP Select ''' + TABLE\_SCHEMA + '.' + TABLE\_NAME + '.'  + COLUMN\_NAME + ''' AS Clmn, Name FROM ' + TABLE\_SCHEMA + '.[' + TABLE\_NAME +  '];' AS VARCHAR(MAX))  FROM INFORMATION\_SCHEMA.COLUMNS  JOIN sysobjects B ON INFORMATION\_SCHEMA.COLUMNS.TABLE\_NAME = B.NAME  WHERE COLUMN\_NAME = 'Name'  AND xtype = 'U'  AND TABLE\_SCHEMA = 'Person'   PRINT @SQL  EXEC(@SQL)   SELECT \* FROM #TMP  DROP TABLE #TMP |

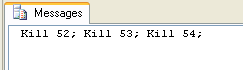
here is the result set.



My personal favorite is being able to kill all the transactions in a database using three lines of code. If you have ever tried to restore a database and could not obtain exclusive access, you know how useful this can be.

|  |
| --- |
| DECLARE @SQL VARCHAR(8000)   SELECT @SQL=COALESCE(@SQL,'')+'Kill '+CAST(spid AS VARCHAR(10))+ '; '  FROM sys.sysprocesses  WHERE DBID=DB\_ID('AdventureWorks')   PRINT @SQL --EXEC(@SQL) Replace the print statement with exec to execute |

will give you a result set such as the following.



**Next Steps**

* Whenever I think I may need a cursor, I always try to find a solution using Coalesce first.
* I am sure I just scratched the surface on the many ways this function can be used. Go try and see what all you can come up with. A little innovative thinking can save several lines of code.

**ISNULL:**

If a column may contain a null, always use the ISNULL function to return a 0 so that your summation queries will not crash. Here’s a quick example that you can run on the pubs database:

USE pubs

GO

SELECT SUM(ISNULL(price, 0))

FROM titles

GO

**ISNULL is a T-**[**SQL**](http://www.programmersheaven.com/2/FAQ-TSQL-Null) **function that is often used to replace NULL values with another value**. In the above example, it is used to turn all NULL values into 0. ISNULL always returns its first argument except when it is NULL, in which case it returns its second argument.

**NULL and the COUNT function**

**Most aggregate functions eliminate null values in calculations; one exception is the COUNT function. When using the COUNT function on a column containing NULL values, the NULL values will be eliminated from the calculation. However, if the COUNT function is used with an** [**asterisk**](http://www.programmersheaven.com/2/FAQ-TSQL-Null)**, it will count all rows regardless of whether NULL values are present.**  
If you want the COUNT function to count all rows of a given column, including the null values, use the ISNULL function. The ISNULL function can replace the null value with a valid value.  
  
In fact, the ISNULL function is very valuable for aggregate functions where NULL values affect the results in an erroneous fashion. Remember that when using an asterisk, the COUNT function will calculate all rows. The following sample codes illustrate the impact of null values in the AVG and COUNT aggregate functions:

SELECT AVG(col1) - NULL values are eliminated.

AVG(ISNULL(col1,0)) A2 - With the IsNULL function, NULL is replaced with 0.

COUNT(col1) C1, - NULL values are eliminated.

COUNT(ISNULL(col1,0)) C2, - With the IsNULL function, NULL is replaced with 0.

COUNT(\*) C3 - count(\*)calculates all rows, even those with NULLs

FROM tabcount

**IS NULL:**

To test if a field is NULL, you need to use IS NULL

How can we test for NULL values?

It is **not possible** to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the IS NULL and IS NOT NULL operators instead.

It is **not** possible to compare NULL and 0; they are not equivalent

**NULLIF**

NULLIF ( expression , expression )

Returns a null value if the two specified expressions are equal. NULLIF returns the first expression if the two expressions are not equal. If the expressions are equal, NULLIF returns a null value of the type of the first expression. NULLIF is equivalent to a searched CASE function in which the two expressions are equal and the resulting expression is NULL.

Following is good example of NULLIF and CASE from BOL:  
USE AdventureWorks;  
GO  
SELECT ProductID, MakeFlag, FinishedGoodsFlag,  
NULLIF(MakeFlag,FinishedGoodsFlag)AS 'Null if Equal'  
FROM Production.Product  
WHERE ProductID < 10;  
GO  
SELECT ProductID, MakeFlag, FinishedGoodsFlag,'Null if Equal' =  
CASE  
WHEN MakeFlag = FinishedGoodsFlag THEN NULL  
ELSE MakeFlag  
END  
FROM Production.Product  
WHERE ProductID < 10;  
GO

**UNION:**

**Difference between UNION and FULL OUTER JOIN**

The UNION Statement will retrieve all the lines from both tables. If in table A you have 10 lines and in table B you have 30 lines, it will return 40 lines. But in full outer join, you will only get 30 lines, and also in full outer join, you need a join on the 2 tables.

**------------**

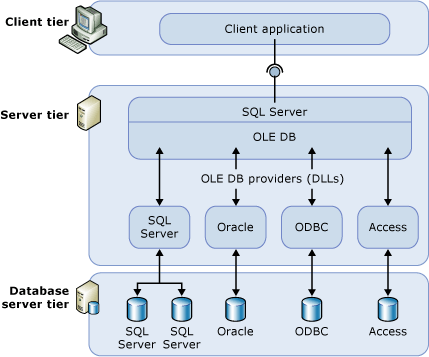
**-- SQL SERVER COMPARE 2 TABLES FOR ROW & COLUMN DIFFERENCES**

**------------  
  
-- TEMPLATE - SQL Server T-SQL compare two tables  
SELECT Label='Found IN Table1, NOT IN Table2',\* FROM   
(SELECT \* FROM Table1  
 EXCEPT  
 SELECT  \* FROM Table2) x  
UNION ALL  
SELECT Label='Found IN Table2, NOT IN Table1',\* FROM  
(SELECT  \* FROM Table2  
 EXCEPT  
 SELECT \* FROM Table1) y**

**GO**

**LINKED SERVERS:**

The following illustration shows the basics of a linked server configuration.



Typically, linked servers are used to handle distributed queries. When a client application executes a distributed query through a linked server, SQL Server parses the command and sends requests to OLE DB. The rowset request may be in the form of executing a query against the provider or opening a base table from the provider.

For a data source to return data through a linked server, the OLE DB provider (DLL) for that data source must be present on the same server as the instance of SQL Server.

If you have a magic wand or a special relationship with some data fairy, all of your data is in the same database on the same server. Most of us aren’t that lucky. Sometimes, the data we need is a non-database format or resides on a different server. SQL Server is flexible enough to offer a number of ways to get the data you need. One of the easiest methods is to directly link to the data via a linked server. (Microsoft SQL Server 2005 Express Edition also supports linked servers.)

### What’s a linked server?

A linked server is simply a connection to an Object Linking and Embedding Database (OLEDB) data source. Technically, OLEDB is a Microsoft standard API for retrieving data from a wide variety of data. If that’s clear as mud, don’t worry. The good news is that it’s flexible enough to link to database and non-database formats, such as a spreadsheet or e-mail client. Simply put, SQL Server supports any OLEDB provider (also called a driver). There’s more good news: You can use Transact-SQL or Management Studio to make the connection. After you create a linked server, SQL Server can log into another database server. **That means you can run queries on a remote server. You have two types of linked server queries at your disposal: ad hoc and permanent.**

### Ad hoc links

Technically, you won’t use the term linked server to identify an ad hoc query. That term really refers to a SQL Server object. However, you will often see the term used to refer to an ad hoc linked query. An ad hoc query opens and closes the connection. A permanent linked server is always available. Use OPENROWSET for infrequent linked tasks, using the following syntax:

OPENROWSET('providername', 'datasource', 'username', 'password', object)

An OPENROWSET link consumes less space in your database. Use valid Transact-SQL statements to manipulate the retrieved data. The arguments are self-explanatory, but remember that datasource is the source’s full path, not just a filename. In addition, the provider provides the instructions that SQL Server needs to get in and grab data. They’re specific to the foreign software you’re accessing.

Now, let’s look at a simple ad hock query that selects all the records from the Employees table in the Microsoft Access sample database, Northwind:

SELECT \*

FROM OPENROWSET('Microsoft.Jet.OLEDB.4.0',

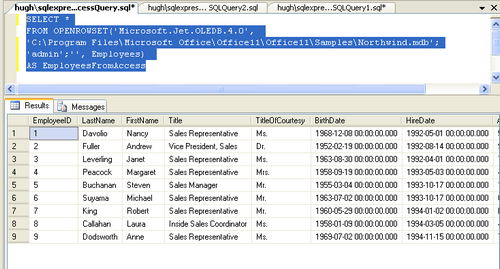
'C:Program FilesMicrosoft OfficeOffice11Office11SamplesNorthwind.mdb';

'admin';'', Employees)

AS EmployeesFromAccess

**Figure A** shows the quickly retrieved results, which depend on the user (admin) having the appropriate permissions. (If you want to run this query, be sure to update the path to Northwind.mdb to accommodate your system.) The provider string is specific to the data engine, Jet. The AS keyword provides a name for the new table inside SQL Server.

#### Figure A: Use OPENROWSET for infrequent queries of foreign data (Express edition).



If the ad hoc query returns an error, you probably need to enable the ad hoc query feature. (SQL Server disables it by default.) To enable ad hoc queries, run the SQL Server Configuration Manager (the SQL Server Surface Area Configuration utility in the Express edition). Click the Surface Area Configuration For Features link and check the Enable OPENROWSET And OPENDATASOURCE Support option. Then, click OK and close the utility. Alternatively, run the sp\_configure stored procedure.

### Permanent links

SQL Server’s linked server object creates a permanent link to a remote server. When the user logs in, SQL Server also logs into the remote server. The first step to retrieving foreign data via a linked server is to let SQL Server know that you plan to talk to another source (server). To do so, execute SQL Server’s sp\_addlinkedserver stored procedure, using the following syntax:

sp\_addlinkedserver server, productname, provider, datasource, location, providerstring, catalog

Refer to **Table A** for an explanation of each of the procedure’s arguments. Notice that a few of the arguments are limited to code.

#### Table A: sp\_addlinkedserver stored procedure arguments

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Argument | Explanation | Default | Management Studio | Parameter Constant |
| server | The local name of the linked server you’re creating | None | Linked Server | @server |
| productname | The product name of the OLEDB data source you’re adding as a linked server | NULL | Product Name | @srvproduct |
| provider | The source’s unique OLEDB identifier | NULL | Provider Name | @provider |
| datasource | The name of the data source | NULL | Data Source | @datasrc |
| location | The location of the linked source file | NULL | N/A | @location |
| providerstring | The OLEDB string that identifies the source | NULL | N/A | @provstr |
| catalog | Catalog used with the connection | NULL | N/A | @catalog |

The arguments aren’t optional, but you won’t use them all together, which explains the NULL value defaults. The provider determines which arguments you need. When using SQL Server, you need only the first two arguments.

Now, let’s suppose you want to create a permanent link to another SQL Server database on the Human Resources server. To do so, you’d use the following statement:

EXEC sp\_addlinkedserver @server = 'EmployeeStats',

@provider = 'SQL Server',

@datasrc = 'Human Resources'

The linked server’s name, EmployeeStats, is a string that you provide to represent the server. It doesn’t exist until you name and create it. The following syntax statements represent the creation of a linked server to several common data sources:

**Microsoft Access**

EXEC sp\_addlinkedserver @server = 'NorthwindDemo',

@srvproduct = 'Microsoft.Jet.OLEDB.4.0',

@provider = 'Microsoft.Jet.OLEDB.4.0',

@datasrc = 'C:LinkedNorthwind.mdb'

**Microsoft Excel**

EXEC sp\_addlinkedserver @server = 'ExcelEmployeeData',

@srvproduct = 'Microsoft.Jet.OLEDB.4.0',

@provider = 'Microsoft.Jet.OLEDB.4.0',

@datasrc = 'C:LinkedEmployees.xls'

**Oracle**

EXEC sp\_addlinkedserver @server = 'OracleDemo',

@srvproduct = 'Oracle',

@provider = 'MSDAORA',

@datasrc = 'OracleAlias'

Oracle is a bit different because you must create an alias that points to the Oracle server rather than specify an actual file (@datasrc). Every situation is different. Your best defense is experience and an informed technical staff.

SQL Server throws a monkey wrench into the works because it doesn’t return an error if you use the wrong syntax. Instead, any attempt to use the linked server simply fails, but the failure might not be obvious. My best recommendation is to test a query that you know should return data. If you get nothing, you know the linked server isn’t really connecting and the problem is most likely the arguments you fed sp\_addlinkedserver to create the linked server.

**Typically, SUM(..) OVER(..) is most useful for calculating a percentage of a total for each row.** For example, for each Order we can calculate the percentage of that order's orderAmount compared to the customer's total orderAmount:

select customerID, productID, orderDate, orderAmount,

orderAmount / sum(orderAmount) OVER (Partition by CustomerID) as Pct

from Orders

customerID productID orderDate orderAmount Pct

----------- ----------- ----------------------- ------------ -------

1 1 2007-01-01 00:00:00.000 20.00 0.2197

1 2 2007-01-02 00:00:00.000 30.00 0.3296

1 2 2007-01-05 00:00:00.000 23.00 0.2527

1 3 2007-01-04 00:00:00.000 18.00 0.1978

2 1 2007-01-03 00:00:00.000 74.00 0.6271

2 1 2007-01-06 00:00:00.000 34.00 0.2881

2 2 2007-01-08 00:00:00.000 10.00 0.0847

(7 row(s) affected)

Of course, be sure that you don't encounter any divide by zero errors by using a CASE if necessary.

**KPIs**

* Gross Profit Margins
* Net Profit Margins
* Current Ratio
* Inventory Turnover
* Return on Equity
* Accounts Payable Turnover
* Debt-to-equity Ratio
* Net Working Capital
* Labor loader gross margin
* AR Days outstanding
* AR Aging
* Application Conversion Ratio
* Closing Ratio
* Closing %
* Approval ratio
* ROI
* Top Agencies
* YOY Retention
* YOY Attrition

**T-SQL Query Tuning**

Execution plans can tell you how a query will be executed, or how a query was executed. They are, therefore, the DBA's primary means of troubleshooting a poorly performing query. Rather than guess at why a given query is performing thousands of scans, putting your I/O through the roof, you can use the execution plan to identify the exact piece of SQL code that is causing the problem. For example, it may be scanning an entire table-worth of data when, with the proper index, it could simply backpack out only the rows you need. All this and more is displayed in the execution plan.

All graphical plans are read from the right to the left and from the top to the bottom. That's important to know so that you can understand other concepts such as understanding how a hash join works.

### Execution Plan Example

Working through an execution plan to understand what is happening in the query and what needs to get fixed is the primary reason for generating execution plans. Take the following query for example:

[view source](http://sqlserverpedia.com/wiki/Examining_Query_Execution_Plans#viewSource)



[print](http://sqlserverpedia.com/wiki/Examining_Query_Execution_Plans#printSource)[?](http://sqlserverpedia.com/wiki/Examining_Query_Execution_Plans#about)

01.SELECT  soh.[SalesOrderID]

02.    ,soh.[OrderDate]

03.    ,soh.[ShipDate]

04.    ,sod.[ProductID]

05.    ,sod.[OrderQty]

06.    ,sod.[UnitPrice]

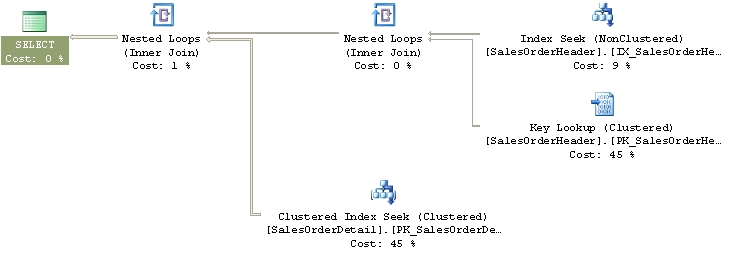
07.    ,soh.[CustomerID]

08.FROM    [Sales].[SalesOrderHeader] AS soh

09.        JOIN [Sales].[SalesOrderDetail] AS sod

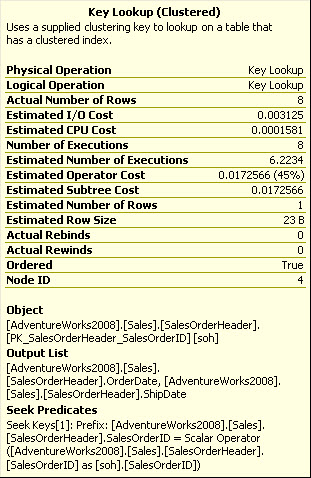
10.            ON soh.[SalesOrderID] = sod.[SalesOrderID]

11.WHERE soh.[CustomerID] = 29559;

This generates the following execution plan: [](http://sqlserverpedia.com/w/index.php?title=File:KeyLookup.jpg&filetimestamp=20090211212243)

Starting at the right and the top you see an Index Seek (NonClustered) against the index named [SalesOrderHeader].[IX\_SalesOrderHeader\_CustomerId]. This feeds data out to a Nested Loop (Inner Join). Working down you can see a Key Lookup (Clustered) operation against the PK\_SalesOrderHeader\_SalesOrderID. This is a classic key lookup, or what used to be called, a bookmark lookup. More on that later. You can see that the data feeds back up to the Nested Loop and then that feeds on down to another Nested Loop operator. Below that is a Clustered Index Seek (Clustered) against the [PK\_SalesOrderDetail\_SalesOrderId] primary key. Finally the data flow goes out to the SELECT operator. That's the basic information available within the execution plan. Lots more detail is also available.

Hover with the mouse over one of the operators and you will get a tool tip, different for each operation type, showing some of the detail behind the operator. Displayed below is the tool tip for the Key Lookup operator:

[](http://sqlserverpedia.com/w/index.php?title=File:KeyLookupTip.jpg&filetimestamp=20090921151244)

At the very top of the tool tip is a description of the operator. In this case, "Uses a supplied clustering key to lookup on a table that has a clustered index." Most operators will include this description, telling you what the operator does within the execution plan. After that, most operations will have a varying number, and type, of fields within the tool tip, supplying different kind of information. An example of one of the common fields is Estimated Operator Cost. You'll see this in most tool tips for most operators. A piece of information that is specific to this operator (although not unique to this operator) is the Seek Predicate information at the bottom of the tool tip.

But the most interesting piece of information for the Key Lookup operator is that it exists within this execution plan. It exists because, while the index on CustomerID is sufficient to get a specific set of rows returned to the application, all the columns needed are not contained on the index. Because the data is stored on the clustered index, and additional set of seeks are required to retrieve the data, which is joined with the information retrieved from the index on CustomerID through the Nested Loop join operation.

To see even more information about the operators in the execution plan, right click an operator and select "Properties" from the drop down menu. This will open a complete properties sheet. Much of the data on the properties sheet is the same as that available in the Tool Tip, but even more is on display in the property sheet.

The graphical output in query analyzer is extremely helpful. Moving your mouse pointer over one of the icons will give you details about the type of operation performed. This utility also lets you create and update statistics, and, create, modify or drop existing indexes. If the statistics are missing or out of date the graphical output will show the table or index in red.

Getting used to various icons might take you a little while, however, the good news is that each of the icons provides a tool tip giving you a brief explanation of what it is for when you hold your mouse pointer over it. It is not recommended that you memorize the meaning of each icon, after looking at this graphical plan you will be able to tell if your query has a problem. The icon that you rarely want to see is a table scan - it looks like a table with a blue arrow in the middle of it.

**COLLATION**

# Predicates (Transact-SQL)

* [SQL Server 2008 R2](http://msdn.microsoft.com/en-us/library/ms189523(v=sql.105).aspx)
* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms189523(v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms189523(v=sql.90).aspx)
* Is an expression that evaluates to TRUE, FALSE, or UNKNOWN. Predicates are used in the search condition of [WHERE](http://msdn.microsoft.com/en-us/library/ms188047.aspx) clauses and [HAVING](http://msdn.microsoft.com/en-us/library/ms180199.aspx) clauses, the join conditions of [FROM](http://msdn.microsoft.com/en-us/library/ms177634.aspx) clauses, and other constructs where a Boolean value is required.

## Difference Between Sql Server VARCHAR and NVARCHAR Data TypeQuantcast

Below table lists out the major difference between the VARCHAR and NVARCHAR Data Type in Sql Server:

|  |  |  |
| --- | --- | --- |
|  | Varchar[(n)] | NVarchar[(n)] |
| Basic Definition | Non-Unicode **Var**iable Length **char**acter data type.  Example: DECLARE @FirstName AS VARCHAR(50) = ‘BASAVARAJ’ SELECT @FirstName | U**N**icode **Var**iable Length **char**acter data type. It can store both non-Unicode and Unicode (i.e. Japanese, Korean etc) characters.  Example: DECLARE @FirstName AS NVARCHAR(50)= ‘BASAVARAJ’ SELECT @FirstName |
| No. of Bytes required for each character | It takes 1 byte per character  Example: DECLARE @FirstName AS VARCHAR(50) = ‘BASAVARAJ’ SELECT @FirstName AS FirstName,DATALENGTH(@FirstName) AS Length **Result:** FirstName Length BASAVARAJ **9** | It takes 2 bytes per Unicode/Non-Unicode character.  Example: DECLARE @FirstName AS NVARCHAR(50)= ‘BASAVARAJ’ SELECT @FirstName AS FirstName,DATALENGTH(@FirstName) AS Length  **Result:** FirstName Length BASAVARAJ **18** |
| Optional Parameter **n** range | Optional Parameter n value can be from 1 to 8000.Can store maximum 8000 Non-Unicode characters. | Optional Parameter n value can be from 1 to 4000.Can store maximum 4000 Unicode/Non-Unicode characters |
| If Optional Parameter **n** is not specified in the variable declaration or column definition | If Optional parameter value **n**is not specified in the variable declaration or column definition then it is considered as 1.  Example: DECLARE @firstName VARCHAR =‘BASAVARAJ’ SELECT @firstName FirstName,DATALENGTH(@firstName) Length  **Result:** FirstName Length **B 1** | If Optional parameter value n is not specified in the variable declaration or column definition then it is considered as 1.  Example: DECLARE @firstName NVARCHAR =‘BASAVARAJ’ SELECT @firstName FirstName,DATALENGTH(@firstName) Length  **Result:** FirstName Length **B 2** |
| If Optional Parameter **n** is not specified in while using CAST/CONVERT functions | When this optional parameter n is not specified while using the CAST/CONVERT functions, then it is considered as 30.Example: DECLARE @firstName VARCHAR(35) =‘BASAVARAJ PRABHU BIRADAR INDIA ASIA’ SELECT CAST(@firstName AS VARCHAR) FirstName,DATALENGTH(CAST(@firstName AS VARCHAR)) Length **Result:** FirstName Length BASAVARAJ PRABHU BIRADAR INDIA 30 | When this optional parameter n is not specified while using the CAST CONVERT functions, then it is considered as 30.Example: DECLARE @firstName NVARCHAR(35) =‘BASAVARAJ PRABHU BIRADAR INDIA ASIA’ SELECT CAST(@firstName AS NVARCHAR) FirstName,DATALENGTH(CAST(@firstName AS NVARCHAR)) Length Result: FirstName Length BASAVARAJ PRABHU BIRADAR INDIA 60 |
| Which one to use? | If we know that data to be stored in the column or variable doesn’t have any Unicode characters. | If we know that the data to be stored in the column or variable can have Unicode characters. |
| Storage Size | Takes no. of bytes equal to the no. of Characters entered plus two bytes extra for defining offset. | Takes no. of bytes equal to twice the no. of Characters entered plus two bytes extra for defining offset. |

As both of these are variable length datatypes, so irrespective of the length (i.e. optional parameter ***n*** value) defined in the variable declaration/column definition it will always take the no. of bytes required for the actual charcters stored. The value of ***n*** defines maximum no. of characters that can be stored.

**TIMESTAMP/ROWVERSION**

**timestamp** is a  data type that exposes automatically generated binary numbers, which are guaranteed to be unique within a database. **timestamp** is used typically as a mechanism for version-stamping table rows. The storage size is 8 bytes.

##### Remarks

The Transact-SQL **timestamp** data type is not the same as the **timestamp** data type defined in the SQL-92 standard. The SQL-92 **timestamp** data type is equivalent to the Transact-SQL **datetime** data type.

A future release of Microsoft® SQL Server™ may modify the behavior of the Transact-SQL **timestamp** data type to align it with the behavior defined in the standard. At that time, the current **timestamp** data type will be replaced with a **rowversion** data type.

Microsoft® SQL Server™ 2000 introduces a **rowversion** synonym for the **timestamp** data type. Use **rowversion** instead of **timestamp** wherever possible in DDL statements. **rowversion** is subject to the behaviors of data type synonyms. For more information, see [Data Type Synonyms](http://technet.microsoft.com/en-us/library/aa258273(v=sql.80).aspx).

In a CREATE TABLE or ALTER TABLE statement, you do not have to supply a column name for the **timestamp** data type:

CREATE TABLE ExampleTable (PriKey int PRIMARY KEY, timestamp)

If you do not supply a column name, SQL Server generates a column name of **timestamp**. The **rowversion** data type synonym does not follow this behavior. You must supply a column name when you specify **rowversion**.

A table can have only one **timestamp** column. The value in the **timestamp** column is updated every time a row containing a **timestamp** column is inserted or updated. This property makes a **timestamp** column a poor candidate for keys, especially primary keys. Any update made to the row changes the **timestamp** value, thereby changing the key value. If the column is in a primary key, the old key value is no longer valid, and foreign keys referencing the old value are no longer valid. If the table is referenced in a dynamic cursor, all updates change the position of the rows in the cursor. If the column is in an index key, all updates to the data row also generate updates of the index.

In SQL Server, ROWVERSION and TIMESTAMP data types represent **automatically generated** binary numbers, unique within the database.

ROWVERSION and TIMESTAMP are **synonyms**, ROWVERSION is available since SQL Server 2005, while TIMESTAMP is deprecated and will be removed in a future version of SQL Server [http://www.sqlines.com/_media/exclamation.png](http://www.sqlines.com/_detail/exclamation.png?id=sql-server:datatypes:timestamp_rowversion).

ROWVERSION (TIMESTAMP) is an **incrementing** 8-byte binary number, and unlike Oracle TIMESTAMP data type, it does not store **any datetime** related information.

You can use timestamp columns to build custom data replication and synchronization solutions.

## ROWVERSION and TIMESTAMP Overview

Summary information:

|  |  |  |
| --- | --- | --- |
| **Syntax** | ROWVERSION | Available since SQL Server 2005 |
| TIMESTAMP | Deprecated [http://www.sqlines.com/_media/exclamation.png](http://www.sqlines.com/_detail/exclamation.png?id=sql-server:datatypes:timestamp_rowversion) |
| **Generated Numbers** | Unique within a database | |
| **Storage Size** | 8 bytes | |
| **Internal Representation** | BINARY(8) | For non-nullable |
| VARBINARY(8) | For nullable |
| **Number of Columns Allowed** | One per table | |

**Last Update**: Microsoft SQL Server 2012

## ROWVERSION and TIMESTAMP Details

When you insert or update a ROWVERSION/TIMESTAMP column, a new unique number **across** the database is generated and assigned to the column.

You can use @@DBTS function to get **the current (last used)** timestamp value:

**SQL Server**:

-- Get the last-used timestamp value

SELECT @@DBTS;

/\* 0x00000000000007D0 \*/

Now let's create 2 table with ROWVERSION columns and insert rows:

**SQL Server**:

-- First table with ROWVERSION column

CREATE TABLE states

(

id CHAR(2),

name VARCHAR(90),

rv ROWVERSION

);

-- Second table with ROWVERSION column

CREATE TABLE cities

(

name VARCHAR(90),

state CHAR(2),

rv ROWVERSION

);

INSERT INTO states (id, name) VALUES ('CA', 'California');

INSERT INTO cities (name, state) VALUES ('San Mateo', 'CA');

Now querying the ROWVERSION columns in these 2 tables, you can see that timestamps were **sequentially** generated:

**SQL Server**:

SELECT rv FROM states;

/\* 0x00000000000007D1 \*/

SELECT rv FROM cities;

/\* 0x00000000000007D2 \*/

**DATEDIFF**

--first day of this month, and six months before that.

select DATEADD(mm, DATEDIFF(mm,0,getdate()), 0),

DATEADD(mm,-6,DATEADD(mm, DATEDIFF(mm,0,getdate()), 0))

--last day of month 6 montsh ago, last day of previous month

select

DATEADD(dd,-1,DATEADD(mm,-5,DATEADD(mm, DATEDIFF(mm,0,getdate()), 0))),

DATEADD(dd,-1,DATEADD(mm, DATEDIFF(mm,0,getdate()), 0))

datediff(mm, getdate(), Closed\_Date) = -1 --This code will return the previous month

datediff(mm, getdate(), Closed\_Date) = 0 --This code will return the current month

datediff(mm, getdate(), Closed\_Date) BETWEEN -3 AND 0 --This code will return the range of the past 3 months including the current month

The **DATEDIFF** date function returns the number of date and time bourndaries crossed between two specified dates.  The syntax of the **DATEDIFF** date function is as follows:

DATEDIFF ( <datepart>, <startdate>, <enddate> )

The <datepart> parameter specifies on which part of the date to calculate the difference.  Valid values are YEAR or YYYY or YY, QUARTER or QQ or Q, MONTH or MM or M, DAYOFYEAR or DY or Y, DAY or DD or D, WEEK or WK or WW, WEEKDAY or DW or W, HOUR or HH, MINUTE or MI or N, SECOND or SS or S and MILLISECOND or ms.  
The <startdate> parameter is the starting date for the calculation and is an expression that returns a DATETIME or SMALLDATETIME value, or a character string in a date format.  The <enddate> parameter is the ending date for the calculation and is an expression that returns a DATETIME or SMALLDATETIME value, or a character string in a date format.  
Here are a few uses of the **DATEDIFF** date function:

**Usage #1: Calculate Age**

DECLARE @BirthDate DATETIME = '1932/06/12'

SELECT DATEDIFF(YEAR, @BirthDate, GETDATE()) -

CASE WHEN MONTH(@BirthDate) < MONTH(GETDATE()) OR

(MONTH(@BirthDate) = MONTH(GETDATE()) AND DAY(@BirthDate) <= DAY(GETDATE()))

THEN 0 ELSE 1 END AS [Age]

**Usage #2: Get Date Part of a DATETIME Value**

SELECT DATEADD(DD, DATEDIFF(DD, 0, GETDATE()), 0) AS [Date Part Only]

**Usage #3 : Get First Day of the Month, Quarter and Year**

SELECT DATEADD(MM, DATEDIFF(MM, 0, GETDATE()), 0) AS [First Day of the Month]

SELECT DATEADD(Q, DATEDIFF(Q, 0, GETDATE()), 0) AS [First Day of the Quarter]

SELECT DATEADD(YEAR, DATEDIFF(YEAR, 0, GETDATE()), 0) AS [First Day of the Year]

**Usage #4 : Get Last Day of the Month, Quarter and Year**

SELECT DATEADD(MM, DATEDIFF(MM, 0, GETDATE()) + 1, 0) - 1 AS [Last Day of the Month]

SELECT DATEADD(Q, DATEDIFF(Q, 0, GETDATE()) + 1, 0) - 1 AS [Last Day of the Quarter]

SELECT DATEADD(YEAR, DATEDIFF(YEAR, 0, GETDATE()) + 1, 0) - 1 AS [Last Day of the Year]

**Usage #5 : Get First Day of the Following Month, Quarter and Year**

SELECT DATEADD(MM, DATEDIFF(MM, 0, GETDATE()) + 1, 0) AS [First Day of Next Month]

SELECT DATEADD(Q, DATEDIFF(Q, 0, GETDATE()) + 1, 0) AS [First Day of Next Quarter]

SELECT DATEADD(YEAR, DATEDIFF(YEAR, 0, GETDATE()) + 1, 0) AS [First Day of Next Year]

**Usage #6 : Get Last Day of the Following Month, Quarter and Year**

SELECT DATEADD(MM, DATEDIFF(MM, 0, GETDATE()) + 2, 0) - 1 AS [Last Day of Next Month]

SELECT DATEADD(Q, DATEDIFF(Q, 0, GETDATE()) + 2, 0) - 1 AS [Last Day of Next Quarter]

SELECT DATEADD(YEAR, DATEDIFF(YEAR, 0, GETDATE()) + 2, 0) - 1 AS [Last Day of Next Year]

**Usage #7 : Get First Day of the Previous Month, Quarter and Year**

SELECT DATEADD(MM, DATEDIFF(MM, 0, GETDATE()) - 1, 0) AS [First Day Of Previous Month]

SELECT DATEADD(Q, DATEDIFF(Q, 0, GETDATE()) - 1, 0) AS [First Day Of Previous Quarter]

SELECT DATEADD(YEAR, DATEDIFF(YEAR, 0, GETDATE()) - 1, 0) AS [First Day Of Previous Year]

**Usage #8 : Get Last Day of the Previous Month, Quarter and Year**

SELECT DATEADD(MM, DATEDIFF(MM, 0, GETDATE()), 0) - 1 AS [Last Day of Previous Month]

SELECT DATEADD(Q, DATEDIFF(Q, 0, GETDATE()), 0) - 1 AS [Last Day of Previous Quarter]

SELECT DATEADD(YEAR, DATEDIFF(YEAR, 0, GETDATE()), 0) - 1 AS [Last Day of Previous Year]

The DATEDIFF function calculates the period of time in dateparts between the second and first of two dates you specify. In other words, it finds an interval between two dates. The result is a signed integer value equal to date2 - date1 in date parts.

The DATEADD function adds an interval to a date you specify. For example, if the due dates of all orders in the SalesOrderHeader table slipped 3 days, you could obtain the new dates with the following statement:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_dc0fe814-a878-49cf-a032-ab94410b44e7');)

USE AdventureWorks;

GO

SELECT DATEADD(day, 3, DueDate)

FROM Sales.SalesOrderHeader;

GO

If the date parameter is a **smalldatetime** data type, the result is also a **smalldatetime**. You can use DATEADD to add seconds or milliseconds to a **smalldatetime** value, but doing this is meaningful only if the result date returned by DATEADD changes by at least 1 minute.

**QUERY OPTIMIZATION:**

* **When you create a SQL Server Profiler trace, one of the events you can collect is called: MISC: Execution Plan. This information (in text form) shows the execution plan used by the query optimizer to execute the query.**
* **From within Query Analyzer, you can run the command SET SHOWPLAN\_TEXT ON. Once you run this command, any query you execute in this Query Analyzer sessions will not be run, but a text-based version of the query plan will be displayed. If the query you are running uses temp tables, then you will have to run the command, SET STATISTICS PROFILE ON before running the query.**

**Of these options, I prefer using the “Show Execution Plan”, which produces a graphical output and considers current server operations.**

**If you see any of the following in an execution plan**, you should consider them warning signs and investigate them for potential performance problems. Each of them are less than ideal from a performance perspective.

* *Index or table scans*: May indicate a need for better or additional indexes.
* *Bookmark Lookups*: Consider changing the current clustered index, consider using a covering index, limit the number of columns in the SELECT statement.
* *Filter*: Remove any functions in the WHERE clause, don’t include views in your Transact-SQL code, may need additional indexes.
* *Sort*: Does the data really need to be sorted? Can an index be used to avoid sorting? Can sorting be done at the client more efficiently?

It is not always possible to avoid these, but the more you can avoid them, the faster query performance will be. [7.0, 2000, 2005] *Updated 8-5-2005*

\*\*\*\*\*

**If you have a stored procedure, or other batch Transact-SQL code that uses temp tables, you cannot use the “Display Estimated Execution Plan” option in the Query Analyzer or Management Studio to evaluate it**. Instead, you must actually run the stored procedure or batch code. This is because when a query is run using the “Display Estimated Execution Plan” option, it is not really run, and temp tables are not created. Since they are not created, any references to them in the code will fail, which prevents an estimated execution plan from being created.

On the other hand, if you use a table variable instead of a temp table, you can use the “Display Estimated Execution Plan” option [7.0, 2000, 2005] *Updated 8-5-2005*

\*\*\*\*\*

**If you have a very complex query you are analyzing in Query Analyzer or Management Studio** as a graphical query execution plan, the resulting plan can be very difficult to view and analyze. You may find it easier to break down the query into its logical components, analyzing each component separately. [7.0, 2000, 2005] *Updated 8-5-2005*

\*\*\*\*\*

**The results of a graphical query execution plan are not always easy to read and interpret**. Keep the following in mind when viewing a graphical execution plan:

* In very complex query plans, the plan is divided into many parts, with each part listed one on top of the other on the screen. Each part represents a separate process or step that the query optimizer has to perform in order to get to the final results.
* Each of the execution plan steps is often broken down into smaller sub-steps. Unfortunately, they are displayed on the screen from right to left. This means you must scroll to the far right of the graphical query plan to see where each step starts.
* Each of the sub-steps and steps is connected by an arrow, showing the path (order) taken of the query when it was executed.
* Eventually, all of the parts come together at the top left side of the screen.
* If you move your cursor above any of the steps or sub-steps, a pop-up windows is displayed, providing more detailed information about this particular step or sub-step.
* If you move your cursor over any of the arrows connecting the steps and sub-steps, you see a pop-up window showing how many records are being moved from one step or sub-step to another step or sub-step.

The arrows that connect one icon to another in a graphical query plan have different thicknesses. **The thickness of the arrow indicates the relative cost in the number of rows and row size of the data moving between each icon**. The thicker the arrow, the more the relative cost is.

You can use this indicator as a quick gauge as to what is happening within the query plan of your query. You will want to pay extra attention to thick arrows in order to see how it affects the performance of your query. For example, thick lines should be at the right of the graphical execution plan, not the left. If you see them on the left, this could indicate that too many rows are being returned, and that the query execution plan is less than optimal. [7.0, 2000, 2005] *Updated 9-19-2005*

\*\*\*\*\*

**In an execution plan, each part of it is assigned a percentage cost**. This represents how much this part costs in resource use, relative to the rest of the execution plan. When you analyze an execution plan, you should focus your efforts on those parts that have the largest percentage cost. This way, you focus your limited time on those areas that have the greatest potential for a return on your time investment. [7.0, 2000, 2005] *Updated 9-19-2005*

\*\*\*\*\*

**In an execution plan, you may have noticed that some parts of the plan are executed more than once**. As part of your analysis of an execution plan, you should focus some of your time on any part that takes more than one execution, and see if there is any way to reduce the number of executions performed. The fewer executions that are performed, the faster the query will be executed. [7.0, 2000, 2005] *Updated 9-19-2005*

\*\*\*\*\*

In an execution plan you will see references to **I/O and CPU cost**. These don’t have a “real” meaning, such as representing the use of a specific amount of resources. These figures are used by the Query Optimizer to help it make the best decision. But there is one meaning you can associate with them, and that is that a smaller I/O or CPU cost uses less server resources than a higher I/O or CPU cost. [7.0, 2000, 2005] *Updated 9-19-2005*

\*\*\*\*\*

When you examine a graphical SQL Server query execution plan, **one of the more useful things to look for are how indexes were used (if at all) by the query optimizer to retrieve data from tables from a given query**. By finding out if an index was used, and how it was used, you can help determine if the current indexes are allowing the query to run as well as it possibly can.

When you place the cursor over a table name (and its icon) in a graphical execution plan and display the pop-up window, you will see one of several messages. These messages tell you if and how an index was used to retrieve data from a table. They include:

* **Table Scan:** If you see this message, it means there was no clustered index on the table and that no index was used to look up the results. **Literally, each row in the table had to be examined**. If a table is relatively small, table scans can be very fast, sometimes faster than using an index.   
    
  So the first thing you want to do, when you see that a table scan has been performed, is to see how many rows there are in the table. If there are not many, then a table scan may offer the best overall performance. But if this table is large, then a table scan will most likely take a long time to complete, and performance will suffer. In this case, you need to look into adding an appropriate index(s) to the table that the query can use.   
    
  **Let’s say that you have identified a query that uses a table scan, but you also discover that there is an appropriate nonclustered index, but it is not being used. What does that mean, and why isn’t the index being used? If the amount of data to be retrieved is large, relative to the size of the table, or if the data is not selective (which means that there are many rows with the same values in the same column), a table scan is often performed instead of an index seek because it is faster. For example, if a table has 10,000 rows, and the query returns 1,000 of them, then a table scan of a table with no clustered index will be faster than trying to use a non-clustered index. Or, if the table had 10,000 rows, and 1,000 of the rows have the same value in the same column (the column being used in the WHERE clause), a table scan is also faster than using a non-clustered index.**  
  When you view the pop-up window when you move the cursor over a table in a graphical query plan, notice the “Estimated Row Count” number. This number is the query optimizer’s best guess on how many rows will be retrieved. If a table scan was done, and this number is very high, this tells you that the table scan was done because a high number of records were returned, and that the query optimizer believed that it was faster to perform a table scan than use the available non-clustered index.
* **Index Seek:** When you see this, it means that the query optimizer used a non-clustered index on the table to look up the results. Performance is generally very quick, especially when few rows are returned.
* **Clustered Index Seek:** If you see this, this means that the query optimizer was able to use a clustered index on the table to look up the results, and performance is very quick. In fact, this is the fastest type of index lookup SQL Server can do.
* **Clustered Index Scan:** A clustered index scan is like a table scan, except that it is done on a table that has a clustered index. Like a regular table scan, a clustered index scan may indicate a performance problem. Generally, they occur for two different reasons. First, there may be too many rows to retrieve, relative to the total number of rows in the table. See the “Estimated Row Count” to verify this. Second, it may be due to the column queried in the WHERE clause may not be selective enough. In any event, a clustered index scan is generally faster than a standard table scan, as not all records in the table always have to be searched when a clustered index scan is run, unlike a standard table scan. Generally, the only thing you can do to change a clustered index scan to a clustered index seek is to rewrite the query so that it is more restrictive and fewer rows are returned.

**Query Analyzer and Management Studio are not the only tools that can generate and display query execution plans** for queries. The SQL Server Profiler can also display them, albeit in text format only. One of the advantages of using Profiler instead of Query Analyzer or Management Studio to display execution plans is that it can do so for a great many queries from your actual production work, instead of running one at a time.

To capture and display query execution plans using Profiler, you must create a trace using the following configuration:

*Events to Capture*

* Performance: Execution Plan
* Performance: Show Plan All
* Performance: Show Plan Statistics
* Performance: Show Plan Text

*Data Columns to Display*

* StartTime
* Duration
* TextData
* CPU
* Reads
* Writes

*Filters*

* Duration. You will want to specify a maximum duration, such as 5 seconds, so that you don’t get flooded with too much data.

Of course, you can capture more information than is listed above in your trace; the above is only a guideline. But keep in mind that you don’t want to capture too much data, as this could have a negative effect on your server’s performance as the trace is being run

**Difference between UNION and UNION ALL**

We use union and union all to combine result sets which return the same number of columns with same data types. And no, it is not same as full outer join.

Lets take an example with one column.

Table1

Col1

1

2

3

4

Table2

Col2

3

4

5

6

Now, when I write

SELECT Col1 FROM Table1

UNION

SELECT Col2 FROM Table2

That would give me 1,2,3,4,5,6 - So it has removed the duplicates.

If I do

SELECT Col1 FROM Table1

UNION ALL

SELECT Col2 FROM Table2

It would return me 1,2,3,4,3,4,5,6 - It has got the duplicates with it.

**Difference between DELETE and TRUNCATE**

**TRUNCATE V/s DELETE**

|  |  |
| --- | --- |
| **Truncate** | **Delete** |
| TRUNCATE is a DDL command | DELETE is a DML command |
| TRUNCATE TABLE always locks the table and page but not each row | DELETE statement is executed using a row lock, each row in the table is locked for deletion |
| Cannot use Where Condition | We can specify filters in where clause |
| **It Removes all the data** | **It deletes specified data if where condition exists.** |
| TRUNCATE TABLE cannot activate a trigger because the operation does not log individual row deletions. | Delete activates a trigger because the operation are logged individually. |
| Faster in performance wise, because it is minimally logged in transaction log. | Slower than truncate because, it maintain logs for every record |
|  |  |
| Drop all object’s statistics and marks like High Water Mark free extents and leave the object really empty with the first extent. zero pages are left in the table | keeps object’s statistics and all allocated space. After a DELETE statement is executed,the table can still contain empty pages. |
| TRUNCATE TABLE removes the data by deallocating the data pages used to store the table data and records only the page deallocations in the transaction lo | The DELETE statement removes rows one at a time and records an entry in the transaction log for each deleted row |
| If the table contains an identity column, the counter for that column is reset to the seed value that is defined for the column | DELETE retain the identity |
| Restrictions on using Truncate Statement 1. Are referenced by a FOREIGN KEY constraint. 2. Participate in an indexed view. 3. Are published by using transactional replication or merge replication. | Delete works at row level, thus row level constrains apply |

**DELETE**  
1. DELETE is a DML Command.  
2. DELETE statement is executed using a row lock, each row in the table is locked for deletion.  
3. We can specify filters in where clause  
4. It deletes specified data if where condition exists.  
5. Delete activates a trigger because the operation are logged individually.  
6. Slower than truncate because, it keeps logs.  
7**. Rollback is possible**.  
  
**TRUNCATE**  
1. TRUNCATE is a DDL command.  
2. TRUNCATE TABLE always locks the table and page but not each row.  
3. Cannot use Where Condition.  
4. It Removes all the data.  
5. TRUNCATE TABLE cannot activate a trigger because the operation does not log individual row deletions.  
6. Faster in performance wise, because it doesn't keep any logs.  
7. **Rollback is not possible.**  
  
DELETE and TRUNCATE both can be rolled back when used with TRANSACTION.  
  
If Transaction is done, means COMMITED, then we can not rollback TRUNCATE command, but we can still rollback DELETE command from LOG files, as DELETE write records them in Log file in case it is needed to rollback in future from LOG files.

CHECK constraints are not validated during DELETE statements. Therefore, executing DELETE statements on tables with certain types of check constraints may produce unexpected results. For example, consider the following statements executed on table CheckTbl.

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_f76d8e91-5e49-44dd-808a-94c2a5b975c5');)

INSERT INTO CheckTbl VALUES (10, 10);

GO

DELETE CheckTbl WHERE col1 = 10;

The DELETE statement succeeds, even though the CHECK constraint specifies that table CheckTbl must have at least 1 row.

**IF STMTS using SPs**

If performance were important I'd opt for:

IF @Color = 'Red' AND @Flavor = 'Sweet'

BEGIN

EXEC dbo.usp\_RedSweetStuff

END

IF @Color = 'Red' AND @Flavor <> 'Sweet'

BEGIN

EXEC dbo.usp\_RedStuffThatIsntSweet

END

IF @Color <> 'Red' AND @Flavor = 'Sweet'

BEGIN

EXEC dbo.usp\_NotRedSweetStuff

END

IF @Color <> 'Red' AND @Flavor <> 'Sweet'

BEGIN

EXEC dbo.usp\_NotRedStuff

END

**IF ELSE STMTS**

if (@something = @somethingElse) begin

select \* from someTable

end

else begin

if (@anotherThing = @someOtherThing) begin

select \* from someOtherTable

end

else begin

select \* from aCompletelyDifferentTable

end

end

**IF table EXISTS STMTS**

IF (EXISTS (SELECT \*

FROM INFORMATION\_SCHEMA.TABLES

WHERE TABLE\_SCHEMA = 'TheSchema'

AND TABLE\_NAME = 'TheTable'))

BEGIN

--Do Stuff

END

**STRING inside a STRING**

To get a string to work inside another string use four single quotes.

EXAMPLE:

EXEC( 'CREATE VIEW v\_cbi\_TicketSummary

AS

SELECT si.ref AS TicketNumber, si.sdSummary AS Summary, si.submitterName,

(CASE WHEN si.orgServDeskDefnFK IS NULL THEN ''''

ELSE so.ref

END) AS Organization,

(CASE WHEN sm.displayName IS NULL THEN ''''

ELSE sm.displayName

END) AS machine,

(CASE WHEN si.sdStatusFK IS NULL THEN ''''

ELSE ss1.codeRef

END) AS status,

(CASE WHEN si.sdPriorityFK IS NULL THEN ''''

ELSE sp.codeRef

END) AS Priority,

sm.machGroupGuid,sm.agentGuidStr, sm.displayName, sm.groupName, sm.machName

FROM kasadmin.SDIncident AS si

INNER JOIN kasadmin.orgServDeskDefn AS osdd

ON si.orgServDeskDefnFK = osdd.id

LEFT OUTER JOIN dbo.partnerUser AS pu

ON si.editingpartnerUserFK = pu.id')

## Extracting Account Name and Domain from Email using T-SQL

By [Vivek](http://www.sqlserver007.com/author/Vivek/) on June 20th, 2009

Storing the list of email subscribers is quite often seen in the web based companies. Once the emails are stored they are segmented based on the customer interests and products they have purchased on the website.  
In this article I will show the simple step of extracting the Account Name and Domain Name using T-sql.

DECLARE @email NVARCHAR(MAX)  
DECLARE @EmailAccount NVARCHAR(100)  
DECLARE @EmailDomain NVARCHAR(100)

SET @email = ‘george@yahoo.com’

SET @EmailAccount = Lower(LTrim(Substring(@Email, 1, Charindex(‘@’,@Email, 1) – 1)))  
SET @EmailDomain = Lower(RTrim(Substring(@Email, Charindex(‘@’,@Email, 1) + 1, 1000)))

PRINT @EmailAccount  
PRINT @EmailDomain

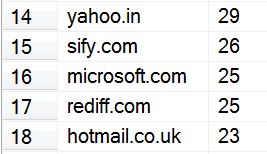
## SQL SERVER – Selecting Domain from Email Address

June 18, 2011 by [pinaldave](http://blog.sqlauthority.com/author/pinaldave/)

Recently I came across a quick need where I needed to retrieve domain of the email address. The email address is in the database table. I quickly wrote following script which will extract the domain and will also count how many email addresses are there with the same domain address.

SELECT RIGHT(Email, LEN(Email) - CHARINDEX('@', email)) Domain ,  
COUNT(Email) EmailCount  
FROM   dbo.email  
WHERE  LEN(Email) > 0  
GROUP BY RIGHT(Email, LEN(Email) - CHARINDEX('@', email))  
ORDER BY EmailCount DESC

Above script will select the domain after @ character. Please note, if there is more than one @ character in the email, this script will not work as that email address is already invalid.



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* declare

@email varchar(50),

@domain

varchar(50)

set

@email = 'johndoe@nl.eu.whatever.com'

set

@domain = substring(@email, charindex('@', @email)+1, 50)

set

@domain = substring(@domain, 1, len(@domain) - charindex('.', reverse(@domain)))

print

@domain

This gets the subdomains as well, if you only want the top domain you'll have to do sonthing similar to get the bit after the last dot.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*OR\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* Code Snippet

set @domain = substring(@email, charindex(['@'](mailto:'@'), @email)+1, (len(@email) - charindex('.', reverse(@email))) - charindex(['@'](mailto:'@'), @email))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*OR\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DECLARE @eMail varchar(100)  
SET @eMail = ['john.doe@nl.eu.whatever.com'](mailto:'john.doe@nl.eu.whatever.com')  
SELECT parsename( substring( @eMail, ( charindex( ['@'](mailto:'@'), @eMail )), len( @eMail )), 2 )

***Microsoft T-SQL***

***Performance Tuning***

***Part 3: Query Optimization Strategies***

1. Datatype tuning

2. Tuning through database and table partitioning

3. Indexing strategies

4. Query optimizer strategies

5. SHOWPLAN output and analysis

6. Optimizer hints and Join techniques

7. Query tuning tips & tricks

**SUBQUERIES OPTIMIZATION**

As a good rule of thumb try to replace all subqueries with joins. The optimizer may

sometimes automatically *flatten out* subqueries and replace them with regular or outer

joins. But it doesn’t always do a good job at that. Explicit joins give the optimizer more

options to choose the order of tables and find the best possible plan. When you optimize

a particular query investigate if getting rid of subqueries makes a difference.

**UNION VS. UNION ALL**

Whenever possible use *UNION ALL* instead of *UNION*. The difference is that *UNION*

has a “side effect” of eliminating all duplicate rows and sorting results, which *UNION*

*ALL* doesn’t do. Selecting a distinct result requires building a temporary worktable,

storing all rows in it and sorting before producing the output. (Displaying the showplan

on a *SELECT DISTINCT* query will reveal a *stream aggregation* is taking place,

consuming as much as 30% of the resources used to process the query.) In some cases

that’s exactly what you need to do, then *UNION* is your friend. But if you don’t expect

any duplicate rows in the result set, then use *UNION ALL*. It simply selects from one

table or a join, and then selects from another, attaching results to the bottom of the first

result set. *UNION ALL* requires no worktable and no sorting (unless other unrelated

conditions cause that). In most cases it’s much more efficient. One more potential

problem with *UNION* is the danger of flooding tempdb database with a huge worktable.

It may happen if you expect a large result set from a *UNION* query.

**FUNCTIONS AND EXPRESSIONS THAT SUPPRESS INDEXES**

When you apply built-in functions or expressions to indexed columns, the optimizer

cannot use indexes on those columns. Try to rewrite these conditions in such a way that

index keys are not involved in any expression.

**UPDATE ... FROM AND DELETE ... FROM**

T-SQL offers an extension to ANSI-SQL syntax for *UPDATE* and *DELETE* commands

that may be very efficient in many cases. It allows you to specify a *FROM* clause and

join several tables in an *UPDATE* or *DELETE* command.

**SET NOCOUNT ON**

The phenomenon of speeding up T-SQL code by using *SET NOCOUNT ON* was

discussed at length in the last white paper; however, it bears repeating. You have

already noticed that successful queries return a system message about the number of

rows that they affect. In many cases you don’t need this information. Command *SET*

*NOCOUNT ON* allows you to suppress the message for all subsequent transactions in

your session, until you issue the *SET NOCOUNT OFF* command. We know that this is a

double-negative, but T-SQL was not created by English majors.

This option has more than a cosmetic effect on the output generated by your script. It

reduces the amount of information passed from the server to the client. Therefore, it

helps to lower network traffic and improves the overall response time of your

transactions. Time to pass a single message may be negligible, but think about a script

that executes some queries in a loop and sends Kilobytes of useless information to a

user.

As an example, the enclosed file has a T-SQL batch that inserts 9999 rows into the

*big\_sales* table.

insert into big\_sales tbl.sql

When run with *SET NOCOUNT OFF*, the elapsed time was 5176 milliseconds. When

run with *SET NOCOUNT ON*, the elapsed time was 1620 milliseconds!

Consider adding *SET NOCOUNT ON* at the beginning

**TOP AND SET ROWCOUNT**

*The TOP clause of the SELECT statement limits the number of rows returned by a*

*single query, while the SET ROWCOUNT* limits the number of rows affected by all

subsequent queries. These commands provide great efficiencies in numerous

programming tasks.

*SET ROWCOUNT* sets the maximum number of rows that may be affected by a

*SELECT*, *INSERT*, *UPDATE*, or *DELETE* statement. The setting is immediately effective

upon execution of the command and only impacts the current session. In order to

remove this limit, execute *SET ROWCOUNT 0*.

Some practical tasks are much more efficient to program with *TOP* or *SET ROWCOUNT*

than with standard SQL commands. Let us demonstrate it on several examples.

**ASSUMPTIONS ABOUT TEMPORARY TABLE SIZE**

Temporary tables created at run-time within a stored procedure can be problematic. In

SQL Server 7.0, the optimizer was unable to accurately estimate the size of a temporary

table, instead assuming that the temporary table had only 100 rows and uses 10 data

pages. Obviously, this may be wrong in many cases. Now, SQL Server 2000 is stronger

with temporary tables. However, the same advice holds true – create and populate the

temp table, including building indexes and constraints, before executing any conditional

processing on the temp table.

One problem you may encounter is that the optimizer may refuse to recognize indexes

and foreign key constraints that you build on dynamic temporary tables (those created

with # or ##). It *will* recognize constraints and indexes built explicitly using a *CREATE*

*TABLE* statement in the *tempdb* database, as well as non-foreign key constraints on

dynamically built temporary tables.

Don’t forget that all transactions against temporary tables are logged in the *tempdb*

transaction log. Although logging has reduced overhead in *tempdb* and may be as much as

four times as fast as a comparable transaction against a permanent table, you should still

take this into account when sizing the *tempdb* transaction log.

In order to allow the optimizer to take actual table size into account you can use a

technique where you split your code into a separate stored procedures or T-SQL

batches, especially by explicity creating the temp table and its supporting indexes before

any conditional code is executed. The optimizer will then know the size of the temporary

table and whether any good indexes exist before the procedure is executed and will

choose the best plan based on the accurate information.

**LOOP OPTIMIZATION**

**Move Invariant Operations Outside of the Loop**

If you are familiar with other programming languages, then you are probably aware of

loop optimization techniques. You should try to put all operations outside of the loop if

they don’t change inside. This reduces the amount of unnecessary repetitive work. SQL

Server optimizer doesn’t automatically recognize such inefficiencies and clean the code

for you (compilers of some other languages do). You have to write efficient loops

yourself as in the following example.

These scripts print a table of square roots for all numbers from 1 to 100.

**INEFFICIENT LOOP OPERATIONS OPTIMIZED SCRIPT**

SET NOCOUNT ON

DECLARE @message VARCHAR(25),

@counter SMALLINT

SELECT @counter = 0

WHILE @counter < 100

BEGIN

SET @counter = @counter + 1

SET @message = REPLICATE( '-', 25 )

PRINT @message

SET @message =

str( @counter, 10 ) +

str( SQRT( CONVERT( FLOAT,

@counter ) ), 10, 4 )

PRINT @message

END

SET NOCOUNT ON

DECLARE @separator VARCHAR(25),

@message VARCHAR(25),

@counter SMALLINT

SET @counter = 0,

@separator = REPLICATE( '-', 25 )

WHILE @counter < 100

BEGIN

SET @counter = @counter + 1

PRINT @separator

SET @message =

Str( @counter, 10 ) +

Str( SQRT( CONVERT( FLOAT,

@counter ) ), 10, 4 )

PRINT @message

END

Elapsed time: 40 ms Elapsed time: 30 ms

**Replace Loops With Queries**

It may often be possible to replace loops with SQL queries. A single query is almost

always more efficient than multiple iterations because relational databases based upon

set operations.

For instance, we could rewrite the loop shown in the previous example as follows:

SELECT REPLICATE( '-', 25 ) + '

' + STR( ( a.id - 1 ) \* 10 + b.id, 10 )

+ STR( SQRT( CONVERT( FLOAT, ( a.id - 1 ) \* 10 + b.id ) ), 10, 4 )

FROM sysobjects a, sysobjects b

WHERE a.id <= 10

AND b.id <= 10

This command *executes in 0 ms* compared to 30 ms and 40 ms for each of the earlier

scripts!

This script uses the fact that there are rows in table *sysobjects* with *id* column values of

1 through 10. If we join this table to itself and apply filters on column *id* values to take 10

rows from one instance of *sysobjects* and 10 rows from the second instance, then we

get 100 rows (10 times 10). In order to produce numbers 1 through 100 we use

expression (a.id–1)\*10+b.id. The code may look tricky, but it returns the same results

much faster than a loop.

**QUERYING AGAINST COMPOSITE KEYS**

In an earlier white paper and e-seminar, I proposed that composite keys are problematic

for SQL Server. Composite indexes, as you will recall, are composed of several columns

of a table. The problem is that composite indexes are used from leftmost column to right.

The following examples show that SQL Server 2000 now handles poorly ordered

*WHERE* clauses much better than earlier versions of the product. That is, in earlier

versions of the product SQL Server might ignore indexes when all the columns of an

index were addressed in the *WHERE* clause solely because the columns were not

referenced in the same order as they appeared in the index. This is no longer a problem

in SQL Server 2000. However, the problem still impacts how SQL Server chooses query

plans and may cause the optimizer to choose less than optimal plans.

Consider this composite index that contains three columns:

ALTER TABLE add CONSTRAINT [UPKCL\_sales] PRIMARY KEY CLUSTERED

([stor\_id], [ord\_num], [title\_id] )

Depending on your *WHERE* clause conditions, SQL Server may use all or fewer

columns of the index, or not use the index at all, as shown below:

**WHERE CLAUSE**

**CONDITIONS**

**QUERY PLAN**

WHERE stor\_id = @a

AND ord\_num = @b

AND title\_id = @c

|--Clustered Index

Seek(OBJECT:([pubs].[dbo].[big\_sales].[UPKCL\_big\_sales]),

SEEK:([big\_sales].[stor\_id]=[@a]

AND [big\_sales].[ord\_num]=[@b]

AND [big\_sales].[title\_id]=[@c])

ORDERED FORWARD)

WHERE stor\_id = @a

AND ord\_num = @b

|--Clustered Index

Seek(OBJECT:([pubs].[dbo].[big\_sales].[UPKCL\_big\_sales]),

SEEK:([big\_sales].[stor\_id]=[@a]

AND [big\_sales].[ord\_num]=[@b]

ORDERED FORWARD)

WHERE ord\_num = @b

AND stor\_id = @a

|--Clustered Index

Seek(OBJECT:([pubs].[dbo].[big\_sales].[UPKCL\_big\_sales]),

SEEK:([big\_sales].[stor\_id]=[@a]

AND [big\_sales].[ord\_num]=[@b])

ORDERED FORWARD)

Compare this to the previous query and you can see they are the same plan.

WHERE stor\_id = @a |--Clustered Index

Seek(OBJECT:([pubs].[dbo].[big\_sales].[UPKCL\_big\_sales]),

SEEK:([big\_sales].[stor\_id]=[@a]

ORDERED FORWARD)

**Microsoft T-SQL Performance Tuning. Part 3: Query Optimization Strategies - Kevin Kline, Andrew Zanevsky,**

**and Lee Gould, Quest Software, Inc. 25**

**WHERE CLAUSE**

**CONDITIONS**

**QUERY PLAN**

WHERE stor\_id = @a

AND title\_id = @c

|--Clustered Index

Seek(OBJECT:([pubs].[dbo].[big\_sales].[UPKCL\_big\_sales]),

SEEK:([big\_sales].[stor\_id]=[@a]),

WHERE:([big\_sales].[title\_id]=[@c])

ORDERED FORWARD)

This query was not able to use the third column of the clustered index and

instead had to use a separate nonclustered index on *title\_id*.

WHERE ord\_num = @b

AND title\_id = @c

|--Bookmark Lookup(BOOKMARK:([Bmk1000]),

OBJECT:([pubs].[dbo].[big\_sales]))

|--Index Seek(OBJECT:([pubs].[dbo].[big\_sales].

[ndx\_sales\_ttlID]),

SEEK:([big\_sales].[title\_id]=[@c]),

WHERE:([big\_sales].[ord\_num]=[@b])

ORDERED FORWARD)

This query was not able to use the clustered index at all, but was able to

find a highly performant alternate plan.

**Table 1. Usage of Composite Key Columns**

The key point to remember is that you should know the order of columns appearing

within a composite index. Once you know the order of the columns, you should always

structure your *WHERE* clause to analyze columns starting with the leftmost column in

the composite index and work towards the right.

**SUMMARY**

This white paper has presented a collection of tips and trick to help you get the most out

of your queries on a SQL Server 2000 database. Some ideas presented in the white

paper include:

• Subquery optimization

• *UNION* versus *UNION ALL*

• Functions and expressions that suppress indexes

• The advantages of *UPDATE…FROM* and *DELETE…FROM* over ANSI

standard syntax

• *SET NOCOUNT ON*

• *TOP* and *SET ROWCOUNT*

• Temporary table considerations

• Loop optimization

• Recap of querying against concatenated keys

And the key thing to remember in summary is to test, test, retest!

The code snippet below uses a CASE WHEN to display values based on if there is a matching record in another table.

SELECT CASE WHEN C.SR\_Service\_RecID IS NULL

THEN 'NO'

ELSE 'YES'

END AS 'HasConfig',V.\*

FROM v\_cbi\_Tracked\_Tickets v

LEFT JOIN SR\_Config C

ON C.SR\_Service\_RecID = V.SR\_Service\_RecID

where   Closed\_Flag = 'False'

[**Dot Net Tricks**](http://www.dotnet-tricks.com/)

# Difference between CTE and Temp Table and Table Variable

Posted By : Shailendra Chauhan, 15 Sep 2013

Updated On : 04 Oct 2013

  Keywords : cte vs temp table vs table variable, local temp table vs global temp table, when to use cte and temp table and table variable

Temp Table or Table variable or CTE are commonly used for storing data temporarily in SQL Server. In this article, you will learn the differences among these three.

## CTE

CTE stands for Common Table expressions. It was introduced with SQL Server 2005. It is a temporary result set and typically it may be a result of complex sub-query. **Unlike temporary table its life is limited to the current query.** It is defined by using WITH statement. CTE improves readability and ease in maintenance of complex queries and sub-queries. Always begin CTE with semicolon.

### A sub query without CTE is given below :

1. **SELECT \* FROM (**
2. **SELECT Addr.Address, Emp.Name, Emp.Age From Address Addr**
3. **Inner join Employee Emp on Emp.EID = Addr.EID) Temp**
4. **WHERE Temp.Age > 50**
5. **ORDER BY Temp.NAME**

### By using CTE above query can be re-written as follows :

1. **;With CTE1(Address, Name, Age)--Column names for CTE, which are optional**
2. **AS**
3. **(**
4. **SELECT Addr.Address, Emp.Name, Emp.Age from Address Addr**
5. **INNER JOIN EMP Emp ON Emp.EID = Addr.EID**
6. **)**
7. **SELECT \* FROM CTE1 --Using CTE**
8. **WHERE CTE1.Age > 50**
9. **ORDER BY CTE1.NAME**

### When to use CTE

1. This is used to store result of a complex sub query for further use.
2. This is also used to create a recursive query.
3. one really nice feature with CTEs is that you can let it call itself recursively and nest up trees of parent-child relationships.

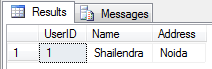
## Temporary Tables

In SQL Server, temporary tables are created at run-time and you can do all the operations which you can do on a normal table. These tables are created inside Tempdb database. Based on the scope and behavior temporary tables are of two types as given below-

1. **Local Temp Table**

Local temp tables are only available to the SQL Server session or connection (means single user) that created the tables. These are automatically deleted when the session that created the tables has been closed. Local temporary table name is stared with single hash ("#") sign.

* 1. **CREATE TABLE #LocalTemp**
  2. **(**
  3. **UserID int,**
  4. **Name varchar(50),**
  5. **Address varchar(150)**
  6. **)**
  7. **GO**
  8. **insert into #LocalTemp values ( 1, 'Shailendra','Noida');**
  9. **GO**
  10. **Select \* from #LocalTemp**

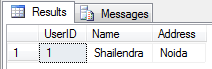
****

The scope of Local temp table exist to the current session of current user means to the current query window. If you will close the current query window or open a new query window and will try to find above created temp table, it will give you the error.

1. **Global Temp Table**

Global temp tables are available to all SQL Server sessions or connections (means all the user). These can be created by any SQL Server connection user and these are automatically deleted when all the SQL Server connections have been closed. Global temporary table name is started with double hash ("##") sign.

* 1. **CREATE TABLE ##GlobalTemp**
  2. **(**
  3. **UserID int,**
  4. **Name varchar(50),**
  5. **Address varchar(150)**
  6. **)**
  7. **GO**
  8. **insert into ##GlobalTemp values ( 1, 'Shailendra','Noida');**
  9. **GO**
  10. **Select \* from ##GlobalTemp**

****

Global temporary tables are visible to all SQL Server connections while Local temporary tables are visible to only current SQL Server connection.

## Table Variable

This acts like a variable and exists for a particular batch of query execution. It gets dropped once it comes out of batch. This is also created in the Tempdb database but not the memory. This also allows you to create primary key, identity at the time of Table variable declaration but not non-clustered index.

1. **GO**
2. **DECLARE @TProduct TABLE**
3. **(**
4. **SNo INT IDENTITY(1,1),**
5. **ProductID INT,**
6. **Qty INT**
7. **)**
8. **--Insert data to Table variable @Product**
9. **INSERT INTO @TProduct(ProductID,Qty)**
10. **SELECT DISTINCT ProductID, Qty FROM ProductsSales ORDER BY ProductID ASC**
11. **--Select data**
12. **Select \* from @TProduct**
14. **--Next batch**
15. **GO**
16. **Select \* from @TProduct --gives error in next batch**

#### Note

1. Temp Tables are physically **created in the Tempdb database**. These tables act as the normal table **and also can have constraints, index like normal tables.**
2. CTE is a named temporary result set which is used to manipulate the complex sub-queries data. This exists for the scope of statement. **This is created in memory rather than Tempdb database**. You cannot create any index on CTE.
3. Table Variable acts like a variable and exists for a particular batch of query execution. It gets dropped once it comes out of batch. This is also created in the Tempdb database but not the memory.

###### Recommended Articles !!

1. [Difference between Primary Key and Unique Key](http://www.dotnet-tricks.com/Tutorial/sqlserver/V2bS260912-Difference-between-Primary-Key-and-Unique-Key.html)

2. [Difference between Primary Key and Foreign Key](http://www.dotnet-tricks.com/Tutorial/sqlserver/TENc260912-Difference-between-Primary-Key-and-Foreign-Key.html)

3. [Drop all tables, stored procedure, views and triggers](http://www.dotnet-tricks.com/Tutorial/sqlserver/T685270912-Drop-all-tables,-stored-procedure,-views-and-triggers.html)

4. [Swap the values of two columns in SQL Server](http://www.dotnet-tricks.com/Tutorial/sqlserver/PW8W071112-Swap-the-values-of-two-columns-in-SQL-Server.html)

5. [Clear List of Recent SQL Server Connection From SQL Server Management Studio](http://www.dotnet-tricks.com/Tutorial/sqlserver/E1L1171112-Clear-List-of-Recent-SQL-Server-Connection-From-SQL-Server-Management-Studio-.html)

6. [SQL Server Naming Conventions and Standards](http://www.dotnet-tricks.com/Tutorial/sqlserver/a54b241112-SQL-Server-Naming-Conventions-and-Standards.html)

7. [How to insert values to identity column in SQL Server](http://www.dotnet-tricks.com/Tutorial/sqlserver/4UXH230213-How-to-insert-values-to-identity-column-in-SQL-Server.html)

8. [Difference between inner join and equi join and natural join](http://www.dotnet-tricks.com/Tutorial/sqlserver/1VEW230213-Difference-between-inner-join-and-equi-join-and-natural-join.html)

9. [Understanding Case Expression in SQL Server with Example](http://www.dotnet-tricks.com/Tutorial/sqlserver/1MS1120313-Understanding-Case-Expression-in-SQL-Server-with-Example.html)

10. [Enable Intellisense in SQL Server 2005, 2008](http://www.dotnet-tricks.com/Tutorial/sqlserver/0KMT071112-Enable-Intellisense-in-SQL-Server-2005,-2008.html)

A key is a single or combination of multiple fields in a table. Its is used to fetch or retrieve records/data-rows from data table according to the condition/requirement. Keys are also used to create relationship among different database tables or views.

## Types of SQL Keys

We have following types of keys in SQL which are used to fetch records from tables and to make relationship among tables or views.

1. **Super Key**

Super key is a set of one or more than one keys that can be used to identify a record uniquely in a table.**Example :** Primary key, Unique key, Alternate key are subset of Super Keys.

1. **Candidate Key**

A Candidate Key is a set of one or more fields/columns that can identify a record uniquely in a table. There can be multiple Candidate Keys in one table. Each Candidate Key can work as Primary Key.

**Example:** In below diagram ID, RollNo and EnrollNo are Candidate Keys since all these three fields can be work as Primary Key.

1. **Primary Key**

Primary key is a set of one or more fields/columns of a table that uniquely identify a record in database table. It can not accept null, duplicate values. Only one Candidate Key can be Primary Key.

1. **Alternate key**

A Alternate key is a key that can be work as a primary key. Basically it is a candidate key that currently is not primary key.

**Example:** In below diagram RollNo and EnrollNo becomes Alternate Keys when we define ID as Primary Key.

1. **Composite/Compound Key**

Composite Key is a combination of more than one fields/columns of a table. It can be a Candidate key, Primary key.

1. **Unique Key**

Uniquekey is a set of one or more fields/columns of a table that uniquely identify a record in database table. It is like Primary key but it can accept only one null value and it can not have duplicate values. For more help refer the article [Difference between primary key and unique key](http://www.dotnet-tricks.com/Tutorial/sqlserver/V2bS260912-Difference-between-Primary-Key-and-Unique-Key.html).

1. **Foreign Key**

Foreign Key is a field in database table that is Primary key in another table. It can accept multiple null, duplicate values. For more help refer the article [Difference between primary key and foreign key](http://www.dotnet-tricks.com/Tutorial/sqlserver/TENc260912-Difference-between-Primary-Key-and-Foreign-Key.html).

**Example :** We can have a DeptID column in the Employee table which is pointing to DeptID column in a department table where it a primary key.

1. **Natural Key**

## Surrogate Keys vs Natural Keys for Primary Key?

Posted by [decipherinfosys](http://decipherinfosys.wordpress.com/) on February 1, 2007

This topic probably is one of those that you cannot get any two database developers/DBAs to agree upon.  Everyone has their own opinion about this and it is also one of the most discussed topics over the web when it comes to data modeling.  Rather than taking any side :-), we are just listing out our experiences when it comes to chosing between a surrogate key vs the natural keys for the tables.

Surrogate Key:

Surrogate keys are keys that have no “business” meaning and are solely used to identify a record in the table.  Such keys are either database generated (example: Identity in SQL Server, Sequence in Oracle, Sequence/Identity in DB2 UDB etc.) or **system generated values** (like generated via a table in the schema).

Natural Key:

Keys are natural if the attribute it represents is used for identification independently of the database schema.  What this basically means is that the keys are natural if people use them example: **Invoice-Numbers, Tax-Ids, SSN** etc.

***Design considerations for choosing the Primary Key***:

**Primary Key should meet the following requirements:**

1. It should be not null, Unique and should apply to all rows.
2. It should be minimal (i.e. less number of columns in the PK: ideally it should be 1, if using composite keys, then make sure that those are surrogates and using integer family data-types).
3. It should be stable over a period of time (should not change i.e. update to the PK columns should not happen).

Keeping these in mind, here are the pros and cons of Surrogate vs. Natural keys:

**Surrogate Key**

I prefer surrogate keys to be DB controlled rather than being controlled via a next-up table in the schema since that is a more scalable approach.

**Pros:**

1. Business Logic is not in the keys.
2. Small 4-byte key (the surrogate key will most likely be an integer and SQL Server for example requires only 4 bytes to store it, if a bigint, then 8 bytes).
3. Joins are very fast.
4. No locking contentions because of unique constraint (this refers to the waits that get developed when two sessions are trying to insert the same unique business key) as the surrogates get generated by the DB and are cached – very scalable.

**Cons:**

1. An additional index is needed.  In SQL Server, the PK constraint will always creates a unique index, in Oracle, if an index already exists, PK creation will use that index for uniqueness enforcement (not a con in Oracle).
2. Cannot be used as a search key.
3. If it is database controlled, for products that support multiple databases, different implementations are needed, example: identity in SS2k, before triggers and sequences in Oracle, identity/sequence in DB2 UDB.
4. Always requires a join when browsing the child table(s).

**Natural Key**

**Pros:**

1. No additional Index.
2. Can be used as a search key.

**Cons:**

1. If not chosen wisely (business meaning in the key(s)), then over a period of time additions may be required to the PK and modifications to the PK can occur.
2. If using strings, joins are a bit slower as compared to the int data-type joins, storage is more as well.  Since storage is more, less data-values get stored per index page.  Also, reading strings is a two step process in some RDBMS: one to get the actual length of the string and second to actually perform the read operation to get the value.
3. Locking contentions can arise if using application driven generation mechanism for the key.
4. Can’t enter a record until value is known since the value has some meaning.

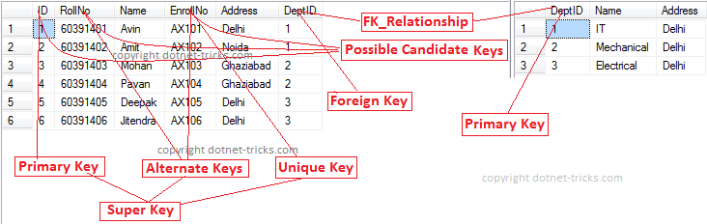
**Choosing Surrogate vs. Natural Keys:**

There is no rule of thumb in this case. It has to be evaluated table by table:

1. If we can identify an appropriate natural key that meets the three criteria for it to be a PK column, we should use it.  Look-up tables and configuration tables are typically ok.
2. Data-Type for the PK: the smaller the better, choose an integer or a short-character data type.   It also ensures that the joins will be faster.  This becomes even more important if you are going to make the PK as a clustered index since non-clustered indexes are built off the clustered index.  RDBMS processes integer data values faster than the character data values because it converts characters to ASCII equivalent values before processing, which is an extra step.

**Defined Keys -**

1. **CREATE TABLE Department**
2. **(**
3. **DeptID int PRIMARY KEY,**
4. **Name varchar (50) NOT NULL,**
5. **Address varchar (200) NOT NULL, )**
6. **CREATE TABLE Student**
7. **(**
8. **ID int PRIMARY KEY,**
9. **RollNo varchar(10) NOT NULL,**
10. **Name varchar(50) NOT NULL,**
11. **EnrollNo varchar(50) UNIQUE,**
12. **Address varchar(200) NOT NULL,**
13. **DeptID int FOREIGN KEY REFERENCES Department(DeptID)**
14. **)**



#### Note

1. Practically in database, we have only three types of keys Primary Key, Unique Key and Foreign Key. Other types of keys are only concepts of RDBMS that we need to know.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*NEW STUFF\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**COMMON SYSTEM STORED PROCEDURES:**

EXEC sp\_helpconstraint

sp\_configure

sp\_addlinkedserver

sp\_send\_dbmail

sp\_help

SP\_who

**COVERED INDEX includes the columns for a specific query.**

SELECT

[TransUnique],[DimFacilityKey],[DimGuarantorKey],[DimPatientKey]\*

FROM dbo.AthenaFactTrans

WHERE TransType = 'charge'

AND DimFinancialClassKey IS NULL

CREATE NONCLUSTERED INDEX [IX\_Sample]

ON [dbo].[AthenaFactTrans] ([TransType],[DimFinancialClassKey]) --<--this covers the WHERE statement

INCLUDE ([TransUnique],[DimFacilityKey],[DimGuarantorKey],[DimPatientKey]) --<--this covers the columns returned

**USING COALESCE**

UPDATE MyTarget

SET MyTarget.InsuranceName = Staging.[Insurance Package Name],

MyTarget.AnSCICode         = COALESCE(AthenaFinClassCorrections.AthenaFC, Staging.ANSCIcode),

MyTarget.FinancialClass    = COALESCE(AthenaFinClassCorrections.HppFC,Staging.FinancialClass)

FROM dbo.DimInsurance MyTarget

LEFT OUTER JOIN AthenaFinClassCorrections ON MyTarget.InsuranceNo = AthenaFinClassCorrections.InsuranceNo

INNER JOIN dbo.Staging\_DimInsurance Staging

ON  MyTarget.InsuranceNo = Staging.[Insurance Package ID]

WHERE MyTarget.AnSCICode <> COALESCE(AthenaFinClassCorrections.AthenaFC, Staging.ANSCIcode)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**QUERY TO CHECK FOR A UNIQUE KEY IN A TABLE**

SELECT EAEOBGRUP,EAEOBCODE,EAPRIORITY

FROM dbo.EOBActionCodes

GROUP BY EAEOBGRUP,EAEOBCODE,EAPRIORITY

HAVING COUNT(\*) >1

**The query results should return no records is these three columns are the unique key for a table**

**DIFFERENCE BETWEEN A #TEMP TABLE AND A cte**

**#temp table:**

* I can INSERT,UPDATE the #temp table
* I can query a result set from the #temp table

**CTE:**

* I **cannot** INSERT,UPDATE the CTE
* I **cannot** query a result set from the CTE