

Version: 01.01

Date: 22-Mar-2012

Developed by: Carolina Modval

Verified by: Eugeniu Zincovschi

Advance Level Training Manual –

Endava SQL Discipline

Advance MS SQL Guide

Table of Contents:

[1. Revision History 4](#_Toc320116091)

[2. Introduction 4](#_Toc320116092)

[3. Purpose of the Document 4](#_Toc320116093)

[4. Common Table Expression 5](#_Toc320116094)

[4.1 Guidelines for Creating and Using Common Table Expressions 7](#_Toc320116095)

[4.2 Guidelines for Defining and Using Recursive Common Table Expressions 8](#_Toc320116096)

[5. Temporary/Declared tables 10](#_Toc320116097)

[5.1 Temporary tables 10](#_Toc320116098)

[5.2 Table Variables (Declared tables) 12](#_Toc320116099)

[6. Dynamic and Static Variables 13](#_Toc320116100)

[7. System tables 17](#_Toc320116101)

[8. DB Access Rights 25](#_Toc320116102)

[8.1 SQL GRANT Command 25](#_Toc320116103)

[8.2 SQL REVOKE Command 26](#_Toc320116104)

[8.3 Privileges and Roles 26](#_Toc320116105)

[8.4 Creating Roles 27](#_Toc320116106)

[9. COALESCE 28](#_Toc320116107)

[9.1 Differences between ISNULL and COALESCE 33](#_Toc320116108)

[10. COLLATE 34](#_Toc320116109)

[11. TRY/CATCH 39](#_Toc320116110)

[11.1 Retrieving Error Information 40](#_Toc320116111)

[11.2 Errors Unaffected by a TRY…CATCH Construct 41](#_Toc320116112)

[12. Bibliography 43](#_Toc320116113)

# Revision History

|  |  |  |
| --- | --- | --- |
| **Revision** | **Date of revision** | **Description of modifications** |
| 01.01 | 22-Mar-2011 | Initial version of the document. |

# 2. Introduction

The demand for testers is usually to test the system's functionality through traditional testing methods and to show some technical knowledge is growing. The testers who can master black-box testing, including database testing, are adding more value to projects, but sometimes the functional testers do not have sufficient SQL database skills.

The Endava SQL Discipline has been created to equip the testing engineers with advanced knowledge to do back-end testing via SQL.

# Purpose of the Document

This document aims at helping testing engineers to build more complex, refined SQL queries by providing a set of MS SQL statements and subsequent examples. The document provides a bunch of practical queries that have been simulated using the **MDCH-QA-TRAIN2** QA training machine. The testers involved in the present training are advised to have access to the above QA training server to try running SQL queries by themselves. (If you do not have permission to work with the QA MSSQL database, please, ask your Line Manager.)

The subject of this document is the SQL commands for the Advanced level of the training process, which are as follows:

* Common table expression
* Temporary/Declared tables
* Dynamic/Static Variables
* System tables
* DB Access Rights
* COLLATE
* COALESCE
* TRY/CATCH

# Common Table Expression

A Common Table Expression (CTE) can be thought of as a temporary result set that is defined within the execution scope of a single SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement. A CTE is similar to a derived table in that it is not stored as an object and lasts only for the duration of the query. Unlike a derived table, a CTE can be self-referencing and can be referenced multiple times in the same query.

A CTE can be used to:

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

Using a CTE offers the advantages of improved readability and ease in maintenance of complex queries. The query can be divided into separate, simple, logical building blocks. These simple blocks can then be used to build more complex, interim CTEs until the final result set is generated. CTEs can be defined in user-defined routines, such as functions, stored procedures, triggers, or views.

A CTE is made up of an expression name representing the CTE, an optional column list, and a query defining the CTE. After a CTE is defined, it can be referenced like a table or view can in a SELECT, INSERT, UPDATE, or DELETE statement. A CTE can also be used in a CREATE VIEW statement as part of its defining SELECT statement.

**Basic syntax structure of the Common Table Expression is:**

|  |
| --- |
| WITH expression\_name [ ( column\_name [,...n] ) ]  AS  ( CTE\_query\_definition ) |

The list of column names is optional only if distinct names for all resulting columns are supplied in the query definition.

**Example 1:** Define the CTE expression name and column list. Define the CTE query. Define the outer query referencing the CTE name.

|  |
| --- |
| -- Define the CTE expression name and column list.  WITH Sales\_CTE (SalesPersonID, SalesOrderID, SalesYear)  AS  -- Define the CTE query.  (  SELECT SalesPersonID, SalesOrderID, YEAR(OrderDate) AS SalesYear  FROM Sales.SalesOrderHeader  WHERE SalesPersonID IS NOT NULL  )  -- Define the outer query referencing the CTE name.  SELECT SalesPersonID, COUNT(SalesOrderID) AS TotalSales, SalesYear  FROM Sales\_CTE  GROUP BY SalesYear, SalesPersonID  ORDER BY SalesPersonID, SalesYear; |

This query will retrieve the following results:

|  |  |  |
| --- | --- | --- |
| SalesPersonID | TotalSales | SalesYear |
| 268 | 4 | 2001 |
| 268 | 20 | 2002 |
| 268 | 14 | 2003 |
| 268 | 10 | 2004 |
| 275 | 56 | 2001 |
| 275 | 139 | 2002 |
| 275 | 169 | 2003 |
| 275 | 86 | 2004 |
| 276 | 43 | 2001 |
| 276 | 127 | 2002 |
| 276 | 166 | 2003 |
| 276 | 82 | 2004 |

...

58 rows

Specifies a temporary named result set, known as a common table expression (CTE). This is derived from a simple query and defined within the execution scope of a single SELECT, INSERT, UPDATE, MERGE, or DELETE statement. This clause can also be used in a CREATE VIEW statement as part of its defining SELECT statement. A common table expression can include references to itself. This is referred to as a recursive common table expression.

**Syntax:**

|  |
| --- |
| [ WITH <common\_table\_expression> [ ,...n ] ]  <common\_table\_expression>::=  expression\_name [ ( column\_name [ ,...n ] ) ]  AS  ( CTE\_query\_definition ) |

**Arguments:**

**expression\_name** - is a valid identifier for the common table expression. expression\_name must be different from the name of any other common table expression defined in the same WITH <common\_table\_expression> clause, but expression\_name can be the same as the name of a base table or view. Any reference to expression\_name in the query uses the common table expression and not the base object.

**column\_name** - specifies a column name in the common table expression. Duplicate names within a single CTE definition are not allowed. The number of column names specified must match the number of columns in the result set of the CTE\_query\_definition. The list of column names is optional only if distinct names for all resulting columns are supplied in the query definition.

**CTE\_query\_definition** - specifies a SELECT statement whose result set populates the common table expression. The SELECT statement for CTE\_query\_definition must meet the same requirements as for creating a view, except a CTE cannot define another CTE. For more information, see the Remarks section and [CREATE VIEW (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms187956.aspx).

If more than one CTE\_query\_definition is defined, the query definitions must be joined by one of these set operators: UNION ALL, UNION, EXCEPT, or INTERSECT. For more information about how to use recursive CTE query definitions, see the following "Remarks" section and [Recursive Queries Using Common Table Expressions](http://msdn.microsoft.com/en-us/library/ms186243.aspx).

## Guidelines for Creating and Using Common Table Expressions

The following guidelines apply to non- recursive common table expressions. A CTE must be followed by a single SELECT, INSERT, UPDATE, MERGE, or DELETE statement that references some or all the CTE columns. A CTE can also be specified in a CREATE VIEW statement as part of the defining SELECT statement of the view.

* Multiple CTE query definitions can be defined in a non-recursive CTE. The definitions must be combined by one of these set operators: UNION ALL, UNION, INTERSECT, or EXCEPT.
* A CTE can reference itself and previously defined CTEs in the same WITH clause. Forward referencing is not allowed.
* Specifying more than one WITH clause in a CTE is not allowed. For example, if a CTE\_query\_definition contains a sub query, that sub query cannot contain a nested WITH clause that defines another CTE.
* The following clauses cannot be used in the CTE\_query\_definition:
  + COMPUTE or COMPUTE BY
  + ORDER BY (except when a TOP clause is specified)
  + INTO
  + OPTION clause with query hints
  + FOR XML
  + FOR BROWSE
* When a CTE is used in a statement that is part of a batch, the statement before it must be followed by a semicolon.
* A query referencing a CTE can be used to define a cursor.
* Tables on remote servers can be referenced in the CTE.
* When executing a CTE, any hints that reference a CTE may conflict with other hints that are discovered when the CTE accesses its underlying tables, in the same manner as hints that reference views in queries. When this occurs, the query returns an error. For more information, see [View Resolution](http://msdn.microsoft.com/en-us/library/ms190237.aspx).
* When a CTE is the target of an UPDATE statement, all references to the CTE in the statement must match. For example, if the CTE is assigned an alias in the FROM clause, the alias must be used for all other references to the CTE. Ambiguous CTE references may produce unexpected join behavior and unintended query results. For more information, see [UPDATE](http://msdn.microsoft.com/en-us/library/ms177523.aspx) .

## Guidelines for Defining and Using Recursive Common Table Expressions

The following guidelines apply to defining a recursive common table expression:

* The recursive CTE definition must contain at least two CTE query definitions, an anchor member and a recursive member. Multiple anchor members and recursive members can be defined; however, all anchor member query definitions must be put before the first recursive member definition. All CTE query definitions are anchor members unless they reference the CTE itself.
* Anchor members must be combined by one of these set operators: UNION ALL, UNION, INTERSECT, or EXCEPT. UNION ALL is the only set operator allowed between the last anchor member and first recursive member, and when combining multiple recursive members.
* The number of columns in the anchor and recursive members must be the same.
* The data type of a column in the recursive member must be the same as the data type of the corresponding column in the anchor member.
* The FROM clause of a recursive member must refer only one time to the CTE expression\_name.
* The following items are not allowed in the CTE\_query\_definition of a recursive member:
  + SELECT DISTINCT
  + GROUP BY
  + HAVING
  + Scalar aggregation
  + TOP
  + LEFT, RIGHT, OUTER JOIN (INNER JOIN is allowed)
  + Sub\_queries
  + A hint applied to a recursive reference to a CTE inside a CTE\_query\_definition.

The following guidelines apply to using a recursive common table expression:

* All columns returned by the recursive CTE are nullable regardless of the nullability of the columns returned by the participating SELECT statements.
* An incorrectly composed recursive CTE may cause an infinite loop. For example, if the recursive member query definition returns the same values for both the parent and child columns, an infinite loop is created. To prevent an infinite loop, you can limit the number of recursion levels allowed for a particular statement by using the MAXRECURSION hint and a value between 0 and 32,767 in the OPTION clause of the INSERT, UPDATE, MERGE, DELETE, or SELECT statement. This lets you control the execution of the statement until you resolve the code problem that is creating the loop. The server-wide default is 100. When 0 is specified, no limit is applied. Only one MAXRECURSION value can be specified per statement.
* A view that contains a recursive common table expression cannot be used to update data.
* Cursors may be defined on queries using CTEs. The CTE is the select\_statement argument that defines the result set of the cursor. Only fast forward-only and static (snapshot) cursors are allowed for recursive CTEs.
* Tables on remote servers may be referenced in the CTE. If the remote server is referenced in the recursive member of the CTE, a spool is created for each remote table so the tables can be repeatedly accessed locally.
* Analytic and aggregate functions in the recursive part of the CTE are applied to the set for the current recursion level and not to the set for the CTE. Functions like ROW\_NUMBER operate only on the subset of data passed to them by the current recursion level and not the entire set of data passed to the recursive part of the CTE. For more information, see [K. Using analytical functions in a recursive CTE](http://msdn.microsoft.com/en-us/library/ms175972.aspx#bkmkusinganalyticalfunctionsinarecursivecte).

**Example 2:** **Using a common table expression to limit counts and report averages.** The following example shows the average number of sales orders for all years for the sales representatives.

|  |
| --- |
| WITH Sales\_CTE (SalesPersonID, NumberOfOrders)  AS  (  SELECT SalesPersonID, COUNT(\*)  FROM Sales.SalesOrderHeader  WHERE SalesPersonID IS NOT NULL  GROUP BY SalesPersonID  )  SELECT AVG(NumberOfOrders) AS "Average Sales Per Person"  FROM Sales\_CTE;  GO |

This query will retrieve the following results:

|  |
| --- |
| Average Sales Per Person |
| 223 |

**Example 3:** Using a recursive common table expression to display multiple levels of recursion.

|  |
| --- |
| USE AdventureWorks;  GO  WITH DirectReports(ManagerID, EmployeeID, Title, EmployeeLevel) AS  (  SELECT ManagerID, EmployeeID, Title, 0 AS EmployeeLevel  FROM dbo.MyEmployees  WHERE ManagerID IS NULL  UNION ALL  SELECT e.ManagerID, e.EmployeeID, e.Title, EmployeeLevel + 1  FROM dbo.MyEmployees AS e  INNER JOIN DirectReports AS d  ON e.ManagerID = d.EmployeeID  )  SELECT ManagerID, EmployeeID, Title, EmployeeLevel  FROM DirectReports  ORDER BY ManagerID;  GO |

This query will retrieve the following results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ManagerID | EmployeeID | Title | EmployeeLevel | |
| NULL | 1 | Chief Executive Officer | | 0 |
| 1 | 273 | Vice President of Sales | | 1 |
| 16 | 23 | Marketing Specialist | | 3 |
| 273 | 16 | Marketing Manager | | 2 |
| 273 | 274 | North American Sales Manager | | 2 |
| 273 | 285 | Pacific Sales Manager | | 2 |
| 274 | 275 | Sales Representative | | 3 |
| 274 | 276 | Sales Representative | | 3 |
| 285 | 286 | Sales Representative | | 3 |

# Temporary/Declared tables

The syntax for creating table variables is quite similar to creating either regular or temporary tables.  The only differences involve a naming convention unique to variables in general, and the need to declare the table variable as you would any other local variable.

## Temporary tables

Temporary tables are created in tempdb. The name "temporary" is slightly misleading, for even though the tables are instantiated in tempdb, they are backed by physical disk and are even logged into the transaction log.  They act like regular tables in that you can query their data via SELECT queries and modify their data via UPDATE, INSERT, and DELETE statements.  If created inside a stored procedure they are destroyed upon completion of the stored procedure.  Furthermore, the scope of any particular temporary table is the session in which it is created; meaning it is only visible to the current user.  Multiple users could create a temp table named #TableX and any queries run simultaneously would not affect one another - they would remain autonomous transactions and the tables would remain autonomous objects.  You may notice that my sample temporary table name started with a "#" sign.  This is the identifier for SQL Server that it is dealing with a temporary table.

The syntax for creating a temporary table is identical to creating a physical table in Microsoft SQL Server with the exception of the aforementioned pound sign (#):

**Example 1:** **Create temporary table**

|  |
| --- |
| CREATE TABLE dbo.#Cars  (  Car\_id int NOT NULL,  ColorCode varchar(10),  ModelName varchar(20),  Code int,  DateEntered datetime  ) |

Temporary tables act like physical tables in many ways.  You can create indexes and statistics on temporary tables.  You can also apply Data Definition Language (DDL) statements against temporary tables to add constraints, defaults, and referential integrity such as primary and foreign keys.  You can also add and drop columns from temporary tables.  For example, if I wanted to add a default value to the DateEntered column and create a primary key using the Car\_id field I would use the following syntax:

**Example 2:** **Alter temporary table**

|  |
| --- |
| ALTER TABLE dbo.#Cars  ADD  CONSTRAINT [DF\_DateEntered] DEFAULT (GETDATE()) FOR [DateEntered],  PRIMARY KEY CLUSTERED  (  [Car\_id]  ) ON [PRIMARY]  GO |

**Example 3: Create/ insert values/select/drop temporary table**

Temporary tables are usually preferred over table variables for a few important reasons: they behave more like physical tables in respect to indexing and statistics creation and lifespan.  An interesting limitation of table variables comes into play when executing code that involves a table variable.  The following two blocks of code both create a table called #Cars and @Cars.  A row is then inserted into the table and the table is finally queried for its values.

|  |
| --- |
| CREATE TABLE dbo.#Cars  (  Car\_id int NOT NULL,  ColorCode varchar(10),  ModelName varchar(20),  Code int ,  DateEntered datetime  )    INSERT INTO dbo.#Cars (Car\_id, ColorCode, ModelName, Code, DateEntered)  VALUES (1,'BlueGreen', 'Austen', 200801, GETDATE())  SELECT Car\_id, ColorCode, ModelName, Code, DateEntered FROM dbo.#Cars  DROP TABLE dbo.[#Cars] |

This query will retrieve the following results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Car\_id | ColorCode | ModelName | Code | DateEntered |
| 1 | BlueGreen | Austen | 200801 | * + 1. 2:16:58.733 |

## Table Variables (Declared tables)

The syntax for creating table variables is quite similar to creating either regular or temporary tables.  The only differences involve a naming convention unique to variables in general, and the need to declare the table variable as you would any other local variable:

**Example 1:** **Create table variables**

|  |
| --- |
| DECLARE @Cars table (  Car\_id int NOT NULL,  ColorCode varchar(10),  ModelName varchar(20),  Code int,  DateEntered datetime  ) |

As you can see the syntax bridges local variable declaration (DECLARE @variable\_name variable\_data\_type) and table creation (column\_name, data\_type, nullability).  As with any other local variable, the table variable must be prefixed with an "@" sign.

|  |
| --- |
| **NOTE:**   * 1. Table variables can not have Non-Clustered Indexes. * 2. You can not create constraints in table variables. * 3. You can not create default values on table variable columns. * 4. Statistics can not be created against table variables. |

**Similarities with temporary tables include:**

|  |
| --- |
| **NOTE:**   * 1. Instantiated in tempdb. * 2. Clustered indexes can be created on table variables and temporary tables. * 3. Both are logged in the transaction log. * 4. Just as with temp and regular tables, users can perform all Data Modification Language (DML) queries against a table variable:  SELECT, INSERT, UPDATE, and DELETE. |

**Example 2: Usage of table variable:**

|  |
| --- |
| DECLARE @Cars TABLE     (     Car\_id int NOT NULL,      ColorCode varchar(10),      ModelName varchar(20),      Code int ,     DateEntered datetime     )    INSERT INTO @Cars (Car\_id, ColorCode, ModelName, Code, DateEntered)  VALUES (1,'BlueGreen', 'Austen', 200801, GETDATE())   SELECT Car\_id, ColorCode, ModelName, Code, DateEntered FROM @Cars |

The results differ, depending upon how you run the code.  If you run the entire block of code the following results are returned:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Car\_id | ColorCode | ModelName | Code | DateEntered |
| 1 | BlueGreen | Austen | 200801 | 2012-02-20 07:26:48.693 |

**However, you receive an error if you don't execute all the code simultaneously:**

|  |
| --- |
| Msg 1087, Level 15, State 2, Line 1  Must declare the table variable "@Cars" |

What is the reason for this behavior?  It is quite simple.  A table variable's lifespan is only for the duration of the transaction that it runs in.  If we execute the DECLARE statement first, then attempt to insert records into the @Cars table variable we receive the error because the table variable has passed out of existence.  The results are the same if we declare and insert records into @Cars in one transaction and then attempt to query the table.  If you notice, we need to execute a DROP TABLE statement against #Cars.  This is because the table persists until the session ends or until the table is dropped. So, it would appear that I don't advocate the use of table variables.  That is not true.  They serve a very useful purpose in returning results from table value functions.

# Dynamic and Static Variables

A local variable is an object that can hold a single data value of a specific type. Variables in batches and scripts are typically used:

* As a counter either to count the number of times a loop is performed or to control how many times the loop is performed.
* To hold a data value to be tested by a control-of-flow statement.
* To save a data value to be returned by a stored procedure return code or function return value.

If all arguments are NULL, COALESCE returns NULL.

|  |
| --- |
| **Note:** The names of some system functions begin with two at signs (@@). Although in earlier versions of Microsoft SQL Server, the @@functions are referred to as global variables, they are not variables and do not have the same behaviours as variables. The @@functions are system functions, and their syntax usage follows the rules for functions. |

The DECLARE statement initializes a Transact-SQL variable by:

* Assigning a name. The name must have a single @ as the first character.
* Assigning a system-supplied or user-defined data type and a length. For numeric variables, a precision and scale are also assigned. For variables of type XML, an optional schema collection may be assigned.
* Setting the value to NULL.

For example, the following DECLARE statement creates a dinamic variable named @mycounter with an int data type.

|  |
| --- |
| DECLARE @MyCounter int; |

To declare more than one local variable, use a comma after the first local variable defined, and then specify the next local variable name and data type. For example, the following DECLARE statement creates three local variables named @LastName, @FirstName and @StateProvince, and initializes each to NULL:

**Example 1:** **Declare dynamic variables.**

|  |
| --- |
| DECLARE @LastName nvarchar(30), @FirstName nvarchar(20), @StateProvince nchar(2); |

**Example 2:** 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:

|  |
| --- |
| USE AdventureWorks;  GO  DECLARE @EmpIDVariable int;  SELECT @EmpIDVariable = MAX(EmployeeID)  FROM HumanResources.Employee;  PRINT @EmpIDVariable |

**Result:** 290

**Sets the specified local variable, created by using the DECLARE @local\_variable statement, to the specified value.**

**Example 3:** **Declare static variables.** The scope of a variable is the range of statements that can reference the variable. The scope of a variable lasts from the point it is declared until the end of the batch or stored procedure in which it is declared. For example, the following script generates a syntax error because the variable is declared in one batch and referenced in another:

|  |
| --- |
| USE AdventureWorks;  GO  DECLARE @MyVariable int;  SET @MyVariable = 1; --static variable  -- Terminate the batch by using the GO keyword.  GO  -- @MyVariable has gone out of scope and no longer exists.  -- This SELECT statement generates a syntax error because it is  -- no longer legal to reference @MyVariable.  SELECT BusinessEntityID, NationalIDNumber, JobTitle  FROM HumanResources.Employee  WHERE BusinessEntityID = @MyVariable; |

**Result:**

|  |
| --- |
| Msg 137, Level 15, State 2, Line 7  Must declare the scalar variable "@MyVariable". |

**Example 4:** Variables have local scope and are only visible within the batch or procedure where they are defined. In the following example, the nested scope created for execution of sp\_executesql does not have access to the variable declared in the higher scope and returns and error.

|  |
| --- |
| DECLARE @MyVariable int;  SET @MyVariable = 1;  EXECUTE sp\_executesql N'SELECT @MyVariable'; -- this produces an error |

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.

**Example 4:** 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:

|  |
| --- |
| USE AdventureWorks;  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 |

Following results are returned:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Alberts | Amy | European Sales Manager | Renton | Washington | United States |
| Pak | Jae | Sales Representative | Cambridge | England | United Kingdom |
| Alberts | Amy | European Sales Manager | Renton | Washington | United States |

**Example 6:** The following script creates a small test table and populates it with 26 rows. The script uses a variable to do three things:

* Control how many rows are inserted by controlling how many times the loop is executed.
* Supply the value inserted into the integer column.
* Function as part of the expression that generates letters to be inserted into the character column.

|  |
| --- |
| -- Create the table.  CREATE TABLE TestTable (cola int, colb char(3));  GO  SET NOCOUNT ON;  GO  -- Declare the variable to be used.  DECLARE @MyCounter int;  -- Initialize the variable.  SET @MyCounter = 0;  -- Test the variable to see if the loop is finished.  WHILE (@MyCounter < 26)  BEGIN;  -- Insert a row into the table.  INSERT INTO TestTable VALUES  -- Use the variable to provide the integer value  -- for cola. Also use it to generate a unique letter  -- for each row. Use the ASCII function to get the  -- integer value of 'a'. Add @MyCounter. Use CHAR to  -- convert the sum back to the character @MyCounter  -- characters after 'a'.  (@MyCounter,  CHAR( ( @MyCounter + ASCII('a') ) )  );  -- Increment the variable to count this iteration  -- of the loop.  SET @MyCounter = @MyCounter + 1;  END;  GO  SET NOCOUNT OFF;  GO  -- View the data.  SELECT cola, colb  FROM TestTable;  GO  DROP TABLE TestTable;  GO |

Following results are returned:

|  |  |
| --- | --- |
| cola | colb |
| 0 | a |
| 1 | b |
| 2 | c |
| 3 | d |
| 4 | e |
| 5 | f |
| 6 | g |
| 7 | h |

…

26 rows

# System tables

The database system contains a series of system tables with information about database objects and their connection to each other, statistics system tables, monitor system tables and loader system tables.

The following information describes how to evaluate the system tables.

* When you specify SELECT statements for the system tables, you have to bear in mind some [general information](http://help.sap.com/saphelp_nw73/helpdata/en/14/13490e956844a2b6531244252826a9/content.htm)
* To find out which system table contains information for a particular database term, consult the list of [database terms](http://help.sap.com/saphelp_nw73/helpdata/en/45/1bc65398615352e10000000a114a6b/content.htm).
* To find out which information a particular system table can provide, see the examples of SELECT statements in the section on [evaluating system tables](http://help.sap.com/saphelp_nw73/helpdata/en/44/f6ed71f60b41e2e10000000a1553f6/content.htm). The examples make use of the [demo data for the SQL Tutorial](http://help.sap.com/saphelp_nw73/helpdata/en/8c/28b4ec2db945849b74310a0ba7bc83/frameset.htm).
* For a complete definition of some of the most important system tables, see [Definition of the System Tables](http://help.sap.com/saphelp_nw73/helpdata/en/45/2c5756ed062b30e10000000a1553f7/content.htm).

If you want to evaluate system tables, note the following information:

|  |
| --- |
| **NOTE:**   * When system tables are evaluated, the system only outputs information for objects of which the current database user is the owner, or for which the database user has at least one privilege. This authorization concept may mean that the table definition visible for the current database user differs from the actual definition. The definition of a view table is only visible for the owner of the view table. * When you query the system table(s), you should enter conditions that describe the required object as precisely as possible. Entering the object owner considerably speeds up the search for the relevant information. * When you specify search commands, you should specify equivalence conditions where possible. Specifying LIKE conditions is less effective. * For performance reasons, when you query information from system tables, you should use not only the SQL statement SELECT \* but also limit the number of output columns to those columns that you actually require.When you query statistical information from system tables, in particular, additional actions are performed to determine column values in the database system when certain output columns are requested. As a result, you should only have the system determine this column information if you really need it. * Simple identifiers are always created in the database instance in upper-case letters, irrespective of how they were entered when the data was defined.If you use simple identifiers in a search condition, you must enter the single quotes that are typical for specifying literals. |

System information — through INFORMATION\_SCHEMA views and/or through system stored procedures and functions. One of the ways to obtain system information is direct querying of the system tables — that is, tables and views that contain information about the current database (e.g., sysobjects, sysindexes, sysusers, etc. — up to a total of 19 tables). Those stored in Master database contain information about the RDBMS itself.

While it is possible — for a user with sufficient privileges — to query these views and tables, Microsoft strongly discourages such practice, stating that the system tables are for the exclusive use of the SQL Server itself, and that the names and structures might change in future releases. Our advice is to resist the temptation of using this "backdoor" but instead use legitimate interfaces to obtain information.

The INFORMATION\_SCHEMA system views are defined in each database contained in RDBMS. They are based on system tables, which should not be queried directly.

Table 4.1 - Contains information about MS SQL Server - INFORMATION\_SCHEMA views.

|  |  |
| --- | --- |
| INFORMATION\_SCHEMA View | INFORMATION\_SCHEMA View |
| CHECK\_CONSTRAINTS | All check constraints for the current database; based on sysobjects and syscomments system tables. Shows only constraints for which the current user has permission. |
| COLUMN\_DOMAIN\_USAGE | All user-defined data types; based on sysobjects, syscolumns, and systypes system tables. Shows only data-types for which the current user has permission. |
| COLUMN\_PRIVILEGES | All privileges either granted to or by the user; based on sysprotects, sysobjects, and syscolumns system tables. Shows only privileges for which the current user has permission. |
| COLUMNS | Every column in every table in the current database accessible to the user; based on sysobjects, spt\_data type\_info, systypes, syscolumns, syscomments, sysconfigures, and syscharsets system tables. Shows only columns for which the current user has permission. |
| CONSTRAINT\_COLUMN\_USAGE | Every column in the database that has constraint put on it; based on sysobjects, syscolumns, and systypes system tables. Shows only columns for which the current user has permission. |
| CONSTRAINT\_TABLE\_USAGE | Each table in the current database that has a constraint defined on it; based on sysobjects system table. Shows only tables for which the current user has permission. |
| DOMAIN\_CONSTRAINTS | Every user-defined data type that has a rule bound to it; based on sysobjects and systypes system tables. Shows only types for which the current user has permission. |
| DOMAINS | User-Defined Types declared in the current database; based on spt\_data type\_info, systypes, syscomments, sysconfigures, and syscharsets system tables. Shows only UDT(s) for which the current user has permission. |
| KEY\_COLUMN\_USAGE | Each column declared as a key (primary or foreign) for every table in the current database; based on sysobjects, syscolumns, sysreferences, spt\_values, and sysindexes system tables. Shows only key columns for which the current user has permission. |
| PARAMETERS | Each parameter defined for a user-defined stored procedure or function; also shows return parameter for a function; based on sysobjects and syscolumns system tables. Shows only information for the stored procedures and functions for which the current user has permission. |
| REFERENTIAL\_CONSTRAINTS | Each foreign key constraint defined in the current database; based on sysreferences, sysindexes, and sysobjects system tables. Shows only constraints for which the current user has permission. |
| ROUTINES | Every stored procedure or function defined in the current database; based on sysobjects and sysscolumns system tables. Shows only functions for which the current user has permission. |
| ROUTINE\_COLUMNS | User functions that are table-valued (containing a SELECT statement); based on sysobjects and syscolumns system tables. Shows only functions for which the current user has permission. |
| SCHEMATA | All databases accessible to a user; based on sysdatabases, sysconfigures, and syscharsets system tables. Shows only databases for which the current user has permission. |
| TABLE\_CONSTRAINTS | Table constraints defined in the current database; based on sysobjects system table. Shows only constraints for which the current user has permission. |
| INFORMATION\_SCHEMA View | Contains Information About |
| TABLE\_PRIVILEGES | Each table privilege either granted to or by the user; based on sysobjects and sysprotects system tables. Shows only constraints for which the current user has permission. |
| TABLES | Every table defined in the current database; based on sysobjects system table. Shows only tables for which the current user has permission. |
| VIEW\_COLUMN\_USAGE | Each column used in a view definition; based on sysobjects and sysdepends system tables. Shows only columns for which the current user has permission. |
| VIEW\_TABLE\_USAGE | Tables used as a base table for the views; based on sysobjects and sysdepends system tables. Shows only tables for which the current user has permission. |
| VIEWS | Views in the current database accessible to a user; based on sysobjects and syscomments system tables. |

The INFORMATION\_SCHEMA views are queried just like any other view or table in the database with one important distinction: the view name must be preceded with the INFORMATION\_SCHEMA qualifier. Each view contains several columns, and search conditions can be specified on the columns these views contain. Below are several examples illustrating the use of the INFORMATION\_SCHEMA views.

**Example 1:** The following query returns information about every column. (The results displayed were shortened somewhat, because the query returns all rows for each table in the database, including 19 system tables, used by the SQL Server to keep track of the objects; only 4 columns were requested).

|  |
| --- |
| SELECT table\_name,column\_name, column\_default, data\_type FROM information\_schema.columns |

This query will retrieve the following results:

|  |  |  |  |
| --- | --- | --- | --- |
| table\_name | column\_name | column\_default | data\_type |
| spt\_fallback\_db | xserver\_name | NULL | varchar |
| spt\_fallback\_db | xdttm\_ins | NULL | datetime |
| spt\_fallback\_db | xdttm\_last\_ins\_upd | NULL | datetime |
| spt\_fallback\_db | xfallback\_dbid | NULL | smallint |
| spt\_fallback\_db | name | NULL | varchar |
| spt\_fallback\_db | dbid | NULL | smallint |

…

1031 rows

**Example 2:** **How do I find all the tables that do not have a clustered index in a specified database?**

|  |
| --- |
| USE AdventureWorks  SELECT SCHEMA\_NAME(t.schema\_id) AS schema\_name, t.name AS table\_name  FROM sys.tables AS t  WHERE NOT EXISTS  (  SELECT \* FROM sys.indexes AS i  WHERE i.object\_id = t.object\_id  AND i.type = 1 -- or type\_desc = 'CLUSTERED'  )  ORDER BY schema\_name, table\_name;  GO |

This query will retrieve the following results:

|  |  |
| --- | --- |
| schema\_name | table\_name |
| dbo | CollationTest |
| dbo | DatabaseLog |
| dbo | MSpeer\_request |
| dbo | MSpeer\_response |
| dbo | MSpeer\_topologyrequest |
| dbo | MSpeer\_topologyresponse |
| dbo | MSpub\_identity\_range |
| dbo | NewProducts |
| dbo | Sales |

...

20 rows

**Example 3: How do I find all the tables that do not have a primary key?**

|  |
| --- |
| use AdventureWorks  SELECT SCHEMA\_NAME(t.schema\_id) AS schema\_name  ,t.name AS table\_name  FROM sys.tables t  WHERE object\_id NOT IN  (  SELECT parent\_object\_id  FROM sys.key\_constraints  WHERE type\_desc = 'PRIMARY\_KEY\_CONSTRAINT' -- or type = 'PK'  ); |

This query will retrieve the following results:

|  |  |
| --- | --- |
| schema\_name | table\_name |
| myspace | product |
| dbo | MSpeer\_request |
| dbo | MSpeer\_response |
| dbo | MSpeer\_topologyrequest |
| dbo | MSpeer\_topologyresponse |
| dbo | MSpeer\_originatorid\_history |
| dbo | MSpeer\_conflictdetectionconfigresponse |
| tSQLt | TestMessage |
| tSQLt | Run\_LastExecution |

…

25 rows

**Example 4:** **How do I find all the tables that do not have an index?**

|  |
| --- |
| Use AdventureWorks  SELECT SCHEMA\_NAME(schema\_id) AS schema\_name  ,name AS table\_name  FROM sys.tables  WHERE OBJECTPROPERTY(object\_id,'IsIndexed') = 0  ORDER BY schema\_name, table\_name; |

This query will retrieve the following results:

|  |  |
| --- | --- |
| schema\_name | table\_name |
| dbo | CollationTest |
| dbo | MSpeer\_request |
| dbo | MSpeer\_response |
| dbo | MSpeer\_topologyrequest |
| dbo | MSpeer\_topologyresponse |
| dbo | NewProducts |

…

15 rows

**Example 5:** **How do I find all the tables that have an identity column?**

|  |
| --- |
| USE AdventureWorks;  SELECT SCHEMA\_NAME(schema\_id) AS schema\_name  , t.name AS table\_name  , c.name AS column\_name  FROM sys.tables AS t  JOIN sys.identity\_columns c ON t.object\_id = c.object\_id  ORDER BY schema\_name, table\_name; |

The results may find below:

|  |  |  |
| --- | --- | --- |
| schema\_name | table\_name | column\_name |
| dbo | AWBuildVersion | SystemInformationID |
| dbo | DatabaseLog | DatabaseLogID |
| dbo | ErrorLog | ErrorLogID |
| dbo | MSpeer\_conflictdetectionconfigrequest | id |
| dbo | MSpeer\_lsns | id |
| dbo | MSpeer\_request | id |
| dbo | MSpeer\_topologyrequest | id |
| dbo | NewProducts | ProductID |
| dbo | sysarticles | artid |
| dbo | sysdiagrams | diagram\_id |
| dbo | syspublications | pubid |
| HumanResources | Department | DepartmentID |
| HumanResources | Employee | EmployeeID |

…

51 rows

**Example 6:** **How do I find all the stored procedures in a database?**

|  |
| --- |
| USE AdventureWorks;  GO  SELECT name AS procedure\_name  ,SCHEMA\_NAME(schema\_id) AS schema\_name  ,type\_desc  ,create\_date  ,modify\_date  FROM sys.procedures; |

This query will retrieve the following results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| procedure\_name | schema\_name | type\_desc | create\_date | modify\_date |
| uspPrintError | dbo | SQL\_STORED\_PROCEDURE | 2010-10-26 00:33:17.590 | 2010-10-26 00:33:17.590 |
| uspLogError | dbo | SQL\_STORED\_PROCEDURE | 2010-10-26 00:33:17.590 | 2010-10-26 00:33:17.590 |
| DropClass | tSQLt | SQL\_STORED\_PROCEDURE | 2011-02-02 02:54:39.840 | 2011-02-02 02:54:39.840 |

…

55 rows

**Example 7: How do I find all the user-defined functions in a database?**

|  |
| --- |
| USE AdventureWorks;  SELECT name AS function\_name  ,SCHEMA\_NAME(schema\_id) AS schema\_name  ,type\_desc  ,create\_date  ,modify\_date  FROM sys.objects  WHERE type\_desc LIKE '%FUNCTION%';  GO |

This query will retrieve the following results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function\_name | Schema  \_name | type\_desc | create\_date | modify\_date |
| ufnLeadingZeros | dbo | SQL\_SCALAR\_FUNCTION | 2010-10-26 00:33:17.607 | 2010-10-26 00:33:17.607 |
| private\_getForeignKeyDefinition | tSQLt | SQL\_INLINE\_TABLE\_  VALUED\_FUNCTION | 2011-02-02 02:54:39.873 | 2011-02-02 02:54:39.873 |
| GetTestResultFormatter | tSQLt | SQL\_SCALAR\_FUNCTION | 2011-02-02 02:54:39.947 | 2011-02-02 02:54:39.947 |

…

28 rows

**Example 8:** **How do I find all views in a database?**

|  |
| --- |
| USE AdventureWorks;  SELECT name AS view\_name  ,SCHEMA\_NAME(schema\_id) AS schema\_name  ,OBJECTPROPERTYEX(object\_id,'IsIndexed') AS IsIndexed  ,OBJECTPROPERTYEX(object\_id,'IsIndexable') AS IsIndexable  ,create\_date,modify\_date  FROM sys.views; |

This query will retrieve the following results:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| view\_name | schema\_name | IsIndexed | IsIndexable | create\_date | modify\_date |
| TEST | dbo | 0 | 0 | 2011-12-27 07:53:07.497 | 2011-12-27 07:53:07.497 |
| TEST\_DD | dbo | 0 | 0 | 2011-12-28 00:50:53.157 | 2011-12-28 00:51:42.080 |
| private\_TestClasses | tSQLt | 0 | 0 | 2011-02-02 02:54:40.410 | 2011-02-02 02:54:40.410 |
| vAdditionalContactInfo | Person | 0 | 0 | 2010-10-26 00:33:51.510 | 2010-10-26 00:33:51.510 |

…

21 rows

# DB Access Rights

DCL commands are used to enforce database security in a multiple user database environment. Two types of DCL commands are GRANT and REVOTE. Only Database Administrator's or owner's of the database object can provide/remove privileges on a database object.

## SQL GRANT Command

**The Syntax for the GRANT command is:**

|  |
| --- |
| GRANT privilege\_name  ON object\_name  TO {user\_name |PUBLIC |role\_name}  [WITH GRANT OPTION];**)** |

* ***privilege\_name*** is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT.
* ***object\_name*** is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE.
* ***user\_name*** is the name of the user to whom an access right is being granted.
* ***user\_name*** is the name of the user to whom an access right is being granted.
* ***PUBLIC*** is used to grant access rights to all users.
* ***ROLES*** are a set of privileges grouped together.
* ***WITH GRANT OPTION*** - allows a user to grant access rights to other users.

**Example 1:** This command grants a SELECT permission on employee table to user1. You should use the WITH GRANT option carefully because for example if you GRANT SELECT privilege on employee table to user1 using the WITH GRANT option, then user1 can GRANT SELECT privilege on employee table to another user, such as user2 etc. Later, if you REVOKE the SELECT privilege on employee from user1, still user2 will have SELECT privilege on employee table.

|  |
| --- |
| GRANT SELECT ON employee TO user1; |

## 8.2 SQL REVOKE Command

The REVOKE command removes user access rights or privileges to the database objects.

***The Syntax for the REVOKE command is:***

|  |
| --- |
| REVOKE privilege\_name  ON object\_name  FROM {user\_name |PUBLIC |role\_name} |

**For Example:** REVOKE SELECT ON employee FROM user1;This command will REVOKE a SELECT privilege on employee table from user1.When you REVOKE SELECT privilege on a table from a user, the user will not be able to SELECT data from that table anymore. However, if the user has received SELECT privileges on that table from more than one user, he/she can SELECT from that table until everyone who granted the permission revokes it. You cannot REVOKE privileges if they were not initially granted by you.

## 8.3 Privileges and Roles

Privilege defines the access rights provided to a user on a database object. There are two types of privileges.

**1) System privileges** - This allows the user to CREATE, ALTER, or DROP database objects.   
**2) Object privileges** - This allows the user to EXECUTE, SELECT, INSERT, UPDATE, or DELETE data from database objects to which the privileges apply.

Table 8.3.1: Few CREATE system privileges are listed below:

|  |  |
| --- | --- |
| ****System Privileges**** | ****Description**** |
| CREATE object | allows users to create the specified object in their own schema. |
| CREATE ANY object | allows users to create the specified object in any schema. |

**The above rules also apply for ALTER and DROP system privileges.**

Table 8.3.2: Few of the object privileges are listed below:

|  |  |
| --- | --- |
| ****Object Privileges**** | ****Description**** |
| INSERT | allows users to insert rows into a table. |
| SELECT | allows users to select data from a database object. |
| UPDATE | allows user to update data in a table. |
| EXECUTE | allows user to execute a stored procedure or a function. |

**Roles:** Roles are a collection of privileges or access rights. When there are many users in a database it becomes difficult to grant or revoke privileges to users. Therefore, if you define roles, you can grant or revoke privileges to users, thereby automatically granting or revoking privileges.

Table 8.3.3: Some of the privileges granted to the system roles are as given below:

|  |  |
| --- | --- |
| ****System Role**** | ****Privileges Granted to the Role**** |
| CONNECT | CREATE TABLE, CREATE VIEW, CREATE SYNONYM, CREATE SEQUENCE, CREATE SESSION etc. |
| RESOURCE | CREATE PROCEDURE, CREATE SEQUENCE, CREATE TABLE, CREATE TRIGGER etc. The primary usage of the RESOURCE role is to restrict access to database objects. |
| DBA | ALL SYSTEM PRIVILEGES |

## 8.4 Creating Roles

**The Syntax to create a role is:**

|  |
| --- |
| CREATE ROLE role\_name  [IDENTIFIED BY password]; |

It's easier to GRANT or REVOKE privileges to the users through a role rather than assigning a privilege directly to every user. If a role is identified by a password, then, when you GRANT or REVOKE privileges to the role, you definitely have to identify it with the password.

*We can GRANT or REVOKE privilege to a role as below.*

**Example 1:** To grant CREATE TABLE privilege to a user by creating a testing role. First, create a testing Role

|  |
| --- |
| CREATE ROLE testing |

**Example 2:** Second, grant a CREATE TABLE privilege to the ROLE testing. You can add more privileges to the ROLE.

|  |
| --- |
| GRANT CREATE TABLE TO testing; |

**Example 3:** Third, grant the role to a user.

|  |
| --- |
| GRANT testing TO user1; |

**Example 4:** To revoke a CREATE TABLE privilege from testing ROLE, you can write:

|  |
| --- |
| REVOKE CREATE TABLE FROM testing; |

# COALESCE

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.

**Syntax:**

|  |
| --- |
| COALESCE ( expression [ ,...n ] ) |

Expression - is an [expression](http://msdn.microsoft.com/en-us/library/ms190718.aspx) of any type. If all arguments are NULL, COALESCE returns NULL.

|  |
| --- |
| **NOTE:** At least one of the null values must be a typed NULL. |

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. In SQL Server, to index expressions involving COALESCE with non-null parameters, the computed column can be persisted using the PERSISTED column attribute as in the following statement:

**Example 1:** For example, will return the current date.  It bypasses the first NULL values and returns the first non-null value.

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

Find results in the table below:

|  |
| --- |
| CurrentDate |
| 2012-02-20 05:30:47.343 |

**Example 2: 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:

|  |
| --- |
| Name |
| Human Resources |
| Finance |
| Information Services |
| Facilities and Maintenance |
| Executive |

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 |

You will get the following result set:

|  |
| --- |
| Department Name |
| Human Resources;Finance;Information Services;Facilities and Maintenance;Executive; |

**Example 3:** You have a table of persons whose columns include FirstName, MiddleName and LastName. If you want to print their complete names as single strings, here's how to do it with COALESCE():

|  |
| --- |
| SELECT FirstName + ' ' +COALESCE(MiddleName,'')+ ' '+COALESCE(LastName,'')  from[AdventureWorksLT].[SalesLT].[Customer] |

The result set will be the following:

|  |
| --- |
| Name |
| Orlando N. Gee |
| Keith Harris |
| Donna F. Carreras |
| Janet M. Gates |

…

847 rows selected

**Example 4: 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 |

This query will retrieve the following results:

|  |  |
| --- | --- |
| Clmn | Val |
| Person.AddressType.Name | Archive |
| Person.AddressType.Name | Billing |
| Person.AddressType.Name | Home |
| Person.AddressType.Name | Main Office |
| Person.AddressType.Name | Primary |
| Person.AddressType.Name | Shipping |
| Person.ContactType.Name | Accounting Manager |

…

445 rows selected

**Example 5:** The following example shows how COALESCE selects the data from the first column that has a non-null value.

|  |
| --- |
| SELECT Name, Class, Color, ProductNumber,  COALESCE(Class, Color, ProductNumber) AS FirstNotNull  FROM [AdventureWorks].[Production].[Product]; |

You will get the following result set:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Class | Color | ProductNumber | FirstNotNull |
| Adjustable Race | NULL | NULL | AR-5381 | **AR-5381** |
| Bearing Ball | NULL | NULL | BA-8327 | **BA-8327** |
| BB Ball Bearing | NULL | NULL | BE-2349 | **BE-2349** |
| Headset Ball Bearings | NULL | NULL | BE-2908 | **BE-2908** |
| Blade | NULL | NULL | BL-2036 | **BL-2036** |
| LL Crankarm | L | Black | CA-5965 | **L** |
| ML Crankarm | M | Black | CA-6738 | **M** |
| HL Crankarm | NULL | Black | CA-7457 | **Black** |
| Chainring Bolts | NULL | Silver | CB-2903 | **Silver** |
| Chainring Nut | NULL | Silver | CN-6137 | **Silver** |
| Chainring | NULL | Black | CR-7833 | **Black** |

*...*

504 rows selected

**Example 6:**In the following example, the wages table includes three columns that contain information about the yearly wages of the employees: the hourly wage, salary, and commission. However, an employee receives only one type of pay. To determine the total amount paid to all employees, use COALESCE to receive only the nonnull value found in hourly\_wage, salary, and commission.

|  |
| --- |
| USE tempdb;  IF OBJECT\_ID('dbo.wages') IS NOT NULL  DROP TABLE wages;  GO  CREATE TABLE dbo.wages  (  emp\_id tinyint identity,  hourly\_wage decimal NULL,  salary decimal NULL,  commission decimal NULL,  num\_sales tinyint NULL  );  GO  INSERT dbo.wages (hourly\_wage, salary, commission, num\_sales)  VALUES  (10.00, NULL, NULL, NULL),  (20.00, NULL, NULL, NULL),  (30.00, NULL, NULL, NULL),  (40.00, NULL, NULL, NULL),  (NULL, 10000.00, NULL, NULL),  (NULL, 20000.00, NULL, NULL),  (NULL, 30000.00, NULL, NULL),  (NULL, 40000.00, NULL, NULL),  (NULL, NULL, 15000, 3),  (NULL, NULL, 25000, 2),  (NULL, NULL, 20000, 6),  (NULL, NULL, 14000, 4);  GO  SELECT CAST(COALESCE(hourly\_wage \* 40 \* 52,  salary,  commission \* num\_sales) AS money) AS 'Total Salary'  FROM dbo.wages  ORDER BY 'Total Salary';  GO |

This query will retrieve the following results:

|  |
| --- |
| Total Salary |
| 10000,00 |
| 20000,00 |
| 20800,00 |
| 30000,00 |
| 40000,00 |
| 41600,00 |
| 45000,00 |
| 50000,00 |
| 56000,00 |
| 62400,00 |
| 83200,00 |
| 120000,00 |

## 9.1 Differences between ISNULL and COALESCE

COALESCE basically translates to CASE expression and ISNULL is a built-in implemented in the database engine. Both ISNULL and COALESCE can be used to get the same results but there are some differences.

|  |
| --- |
| **NOTE:**  1. Data type determination of the resulting expression - ISNULL uses the first parameter type, COALESCE follows the CASE expression rules and returns type of value with highest precedence  2. The NULL ability of result expression is different for ISNULL and COALESCE. ISNULL return value is always considered NOT NULL able (assuming the return value is a non-nullable one) where as COALESCE is not. So the expressions ISNULL(NULL, 1) and COALESCE(NULL, 1) although equivalent have different NULL ability values. This makes a difference if you are using these expressions in computed columns and creating key constraints or making return value of a scalar UDF deterministic so that it can be indexed. Otherwise, you can have ISNULL or COALESCE return NULL value just fine.  3. Validations for ISNULL and COALESCE is also different. For example, NULL value for ISNULL is converted to int whereas for COAELSCE you have to provide a type.  ISNULL(NULL, NULL) -- is int  COALESCE(NULL, NULL) -- Will throw an error; COALESCE(CAST(NULL as int), NULL) -- it valid and returns int  4. ISNULL takes only 2 parameters whereas COALESCE takes variable number of parameters  5. COALESCE is based on the ANSI SQL standard whereas ISNULL is a proprietary TSQL function  6. You could get different plans for queries using ISNULL & COALESCE if the expressions involve scalar sub-queries. This will make a performance difference and queries with COALESCE often fare worse here. |

**Example 1:** Below isan example with ISNULL and COALASCE to see the difference between them.

|  |
| --- |
| DECLARE @foo VARCHAR(5)  SET @foo = NULL  SELECT ISNULL(@foo, '123456789') as IsnullResult  SELECT COALESCE(@foo, '123456789') as CoalesceResult |

This query will retrieve the following results:

|  |
| --- |
| IsnullResult |
| 12345 |

|  |
| --- |
| CoalesceResult |
| 123456789 |

# COLLATE

A clause that can be applied to a database definition or a column definition to define the collation or to a character string expression to apply a collation cast.

**Syntax:**

|  |
| --- |
| COLLATE {< collation\_name > | database\_default}  < collation\_name > :: =  { Windows\_collation\_name } | { SQL\_collation\_name } |

**Arguments:**

*collation\_name* - is the name of the collation to be applied to the expression, column definition, or database definition. collation\_name can be only a specified *Windows\_collation\_name* or a *SQL\_collation\_name*.

*Windows\_collation\_name* - is the collation name for Windows collation. See Windows Collation Names.

*SQL\_collation\_name* - is the collation name for a SQL collation. See SQL Collation Names. *database\_default* - Causes the COLLATE clause to inherit the collation of the current database.

|  |
| --- |
| **NOTE:**  The COLLATE clause can be specified at several levels, including the following:  **1. Creating or altering a database.**  You can use the COLLATE clause of the CREATE DATABASE or ALTER DATABASE statement to specify the default collation of the database. You can also specify a collation when you create a database using SQL Server Enterprise Manager. If you do not specify a collation, the database is assigned the default collation of the SQL Server instance.  **2. Creating or altering a table column.**  You can specify collations for each character string column using the COLLATE clause of the CREATE TABLE or ALTER TABLE statement. You can also specify a collation when you create a table using SQL Server Enterprise Manager. If you do not specify a collation, the column is assigned the default collation of the database.  **3. Casting the collation of an expression.**  You can use the COLLATE clause to cast a character expression to a certain collation. Character literals and variables are assigned the default collation of the current database. Column references are assigned the definition collation of the column.  For the collation of an expression, see [Collation Precedence](http://msdn.microsoft.com/en-us/library/aa258272%28v=sql.80%29.aspx).  **Note:** The COLLATE clause can be applied only for the char, varchar, text, nchar, nvarchar, and ntext data types. |

The collation of an identifier depends on the level at which it is defined. Identifiers of instance-level objects, such as logins and database names, are assigned the default collation of the instance. Identifiers of objects within a database, such as tables, views, and column names, are assigned the default collation of the database. For example, two tables with names differing only in case may be created in a database with case-sensitive collation, but may not be created in a database with case-insensitive collation.

Collations are generally identified by a collation name. The exception is in Setup where you do not specify a collation name for Windows collations, but instead specify the collation designator, and then select check boxes to specify binary sorting or dictionary sorting that is either sensitive or insensitive to either case or accents.

You can execute the system function fn\_helpcollations to retrieve a list of all the valid collation names for Windows collations and SQL collations:

|  |
| --- |
| SELECT \*  FROM ::fn\_helpcollations() |

SQL Server can support only code pages that are supported by the underlying operating system. When you perform an action that depends on collations, the SQL Server collation used by the referenced object must use a code page supported by the operating system running on the computer. These actions can include:

* Specifying a default collation for a database when you create or alter the database.
* Specifying a collation for a column when creating or altering a table.

When restoring or attaching a database, the default collation of the database and the collation of any char, varchar, and text columns or parameters in the database must be supported by the operating system.

Code page translations are supported for char and varchar data types, but not for text data type. Data loss during code page translations is not reported. If the collation specified or the collation used by the referenced object, uses a code page not supported by Windows®, SQL Server issues error.

**Example 1:** The example below illustrates databases with different collation:

|  |
| --- |
| use master  go  create database BIN collate Latin1\_General\_BIN  go  create database CI\_AI\_KS collate Latin1\_General\_CI\_AI\_KS  go  create database CS\_AS\_KS\_WS collate Latin1\_General\_CS\_AS\_KS\_WS  go |

**Example 2:** The example below illustrates how to set the **collation at the column level**:

|  |
| --- |
| CREATE TABLE [dbo].[CollationTest](  [keyColumn] [bigint] NULL,  [strColumn] [varchar](max) COLLATE Traditional\_Spanish\_CI\_AI NULL  ) ON [PRIMARY] |

Collation controls the way string values are sorted. The default collation sorts alphabetically using the standard Latin character set. Other collations will sort in different orders. For example, a traditional Spanish collation places words beginning with "ch" at the end of a list of words beginning with "c." It is not necessary to change the default collation to use non-Latin characters. All shared MS SQL databases support Unicode characters, allowing the storage of a variety of alphabets.

Modifying collation settings is an advanced database task and should only be performed by experienced database developers. If you determine it is necessary to change your collation, we recommend modifying it at the table or column level. Making this change at the database level disables some Hosting Control Center functionality, including the installation and removal of ASP Schema objects. Modifying at the table or column level avoids the majority of known compatibility issues with non-standard collation settings.

**Example 3:** The example below illustrates how to set the **collation at the columns level**:

|  |
| --- |
| Create table Mytable (  [colu] char(10) COLLATE Albanian\_CI\_AI\_KS\_WS NULL,  [Maydate] [char] (8) COLLATE Korean\_Wansung\_Unicode\_CS\_AS\_KS NOT NULL ,  [Risk\_Rating] [char] (2) COLLATE SQL\_Latin1\_General\_CP1\_CI\_AS NOT NULL  ) |

**Example 4:** Comparing characters on the databases with different collation. When we run the below code in CI\_AI\_KS and CS\_AS\_KS\_WS the results will be completely different.

|  |
| --- |
| declare @Accentvar1 char(1)  declare @Accentvar2 char(1)  declare @Casevar1 char(1)  declare @Casevar2 char(1)  set @casevar1 ='A'  set @casevar2 ='a'  set @Accentvar1 ='a'  set @Accentvar2 ='á'  if @casevar1 = @casevar2  begin  print "A and a are treated same"  end  else  begin  print "A and a are not treated same"  end  if @Accentvar1 = @Accentvar2  begin  print "A and á are treated same"  end  else  begin  print "A and á are not treated same"  end |

When we execute these statements on a CI\_AI\_KS database, the results are similar to those shown below.

A and a are treated same

A and á are treated same

When we execute these statements on a CS\_AS\_KS\_WS database, the results are similar to those shown below.

A and a are not treated same

A and á are not treated same

**Example 5:** Simulating case sensitivity in a case in-sensitive database. It is often necessary to simulate case sensitivity in a case insensitive database. The example below shows how you can achieve that.

|  |
| --- |
| Use CI\_AI\_KS  go  declare @var1 varchar(10)  declare @var2 varchar(10)  set @var1 ='A'  set @var2 ='a'  if ASCII(@var1) = ASCII(@var2)  print "A and a are treated same"  else  print "A and a are not same" |

However, the function ASCII cannot be used for words. In order to achieve the same functionality of simulating case sensitiveness, we can use the varbinary data type.

|  |
| --- |
| Use CI\_AI\_KS  go  declare @var1 varchar(10)  declare @var2 varchar(10)  set @var1 ='Good'  set @var2 ='gooD'  if cast(@var1 as varbinary) = cast(@var2 as varbinary)  print "Good and gooD are treated same"  else  print "Good and gooD are not treated same" |

Collations specify the rules for how strings of character data are sorted and compared, based on the norms of particular languages and locales.

**Example 3:** Let see following example.

|  |
| --- |
| USE AdventureWorks  GO  /\* Create Test Table \*/  CREATE TABLE TestTable (FirstCol VARCHAR(10))  GO  /\* Check Database Column Collation \*/  SELECT name, collation\_name  FROM sys.columns  WHERE OBJECT\_ID IN ( SELECT OBJECT\_ID  FROM sys.objects  WHERE type = 'U'  AND name = 'TestTable')  GO  /\* Change the database collation \*/  ALTER TABLE TestTable  ALTER COLUMN FirstCol VARCHAR(10)  COLLATE SQL\_Latin1\_General\_CP1\_CS\_AS NULL  GO  /\* Check Database Column Collation \*/  SELECT name, collation\_name  FROM sys.columns  WHERE OBJECT\_ID IN ( SELECT OBJECT\_ID  FROM sys.objects  WHERE type = 'U'  AND name = 'TestTable')  GO  /\* Database Cleanup \*/  DROP TABLE TestTable  GO |

When ran above script will give two resultset. First resultset is before column’s collation is changed and it represents default collation of database. Second result set is after column’s collation is changed and it represents newly defined collation.

|  |  |
| --- | --- |
| name | collation\_name |
| FirstCol | SQL\_Latin1\_General\_CP1\_CI\_AS |

|  |  |
| --- | --- |
| name | collation\_name |
| FirstCol | SQL\_Latin1\_General\_CP1\_CS\_AS |

# TRY/CATCH

Implements error handling for that is similar to the exception handling in the Microsoft Visual C# and Microsoft Visual C++ languages. A group of Transact-SQL statements can be enclosed in a TRY block. If an error occurs in the TRY block, control is passed to another group of statements that is enclosed in a CATCH block.

**Syntax**

|  |
| --- |
| BEGIN TRY  { sql\_statement | statement\_block }  END TRY  BEGIN CATCH  [ { sql\_statement | statement\_block } ]  END CATCH  [ ; ] |

**Arguments:**

*sql\_statement* - is any Transact-SQL statement.

*statement\_block* - any group of Transact-SQL statements in a batch or enclosed in a BEGIN…END block.

**Remarks:**

|  |
| --- |
| **NOTE:**   1. A TRY…CATCH construct catches all execution errors that have a severity higher than 10 that do not close the database connection. 2. A TRY block must be immediately followed by an associated CATCH block. Including any other statements between the END TRY and BEGIN CATCH statements generates a syntax error. 3. A TRY…CATCH construct cannot span multiple batches. A TRY…CATCH construct cannot span multiple blocks of Transact-SQL statements. For example, a TRY…CATCH construct cannot span two BEGIN…END blocks of Transact-SQL statements and cannot span an IF…ELSE construct. 4. If there are no errors in the code that is enclosed in a TRY block, when the last statement in the TRY block has finished running, control passes to the statement immediately after the associated END CATCH statement. If there is an error in the code that is enclosed in a TRY block, control passes to the first statement in the associated CATCH block. If the END CATCH statement is the last statement in a stored procedure or trigger, control is passed back to the statement that called the stored procedure or fired the trigger. 5. When the code in the CATCH block finishes, control passes to the statement immediately after the END CATCH statement. Errors trapped by a CATCH block are not returned to the calling application. If any part of the error information must be returned to the application, the code in the CATCH block must do so by using mechanisms such as SELECT result sets or the RAISERROR and PRINT statements. For more information about how to use RAISERROR with TRY…CATCH, see [Using TRY...CATCH in Transact-SQL](http://msdn.microsoft.com/en-us/library/ms179296.aspx). 6. TRY…CATCH constructs can be nested. Either a TRY block or a CATCH block can contain nested TRY…CATCH constructs. For example, a CATCH block can contain an embedded TRY…CATCH construct to handle errors encountered by the CATCH code. 7. Errors encountered in a CATCH block are treated like errors generated anywhere else. If the CATCH block contains a nested TRY…CATCH construct, any error in the nested TRY block will pass control to the nested CATCH block. If there is no nested TRY…CATCH construct, the error is passed back to the caller. 8. TRY…CATCH constructs catch unhandled errors from stored procedures or triggers executed by the code in the TRY block. Alternatively, the stored procedures or triggers can contain their own TRY…CATCH constructs to handle errors generated by their code. For example, when a TRY block executes a stored procedure and an error occurs in the stored procedure, the error can be handled in the following ways:  * If the stored procedure does not contain its own TRY…CATCH construct, the error returns control to the CATCH block associated with the TRY block that contains the EXECUTE statement. * If the stored procedure contains a TRY…CATCH construct, the error transfers control to the CATCH block in the stored procedure. When the CATCH block code finishes, control is passed back to the statement immediately after the EXECUTE statement that called the stored procedure.  1. GOTO statements cannot be used to enter a TRY or CATCH block. GOTO statements can be used to jump to a label inside the same TRY or CATCH block or to leave a TRY or CATCH block. 2. The TRY…CATCH construct cannot be used in a user-defined function. |

## Retrieving Error Information

In the scope of a CATCH block, the following system functions can be used to obtain information about the error that caused the CATCH block to be executed:

* ERROR\_NUMBER() returns the number of the error.
* ERROR\_SEVERITY() returns the severity.
* ERROR\_STATE() returns the error state number.
* ERROR\_PROCEDURE() returns the name of the stored procedure or trigger where the error occurred.
* ERROR\_LINE() returns the line number inside the routine that caused the error.
* ERROR\_MESSAGE() returns the complete text of the error message. The text includes the values supplied for any substitutable parameters, such as lengths, object names, or times.

These functions return NULL if they are called outside the scope of the CATCH block. Error information can be retrieved by using these functions from anywhere within the scope of the CATCH block.

**Example 1:** See the example below for retrieving error information. For example, the following script shows a stored procedure that contains error-handling functions. In the CATCH block of a TRY…CATCH construct, the stored procedure is called and information about the error is returned.

|  |
| --- |
| USE AdventureWorks;  GO  -- Verify that the stored procedure does not already exist.  IF OBJECT\_ID ( 'usp\_GetErrorInfo', 'P' ) IS NOT NULL  DROP PROCEDURE usp\_GetErrorInfo;  GO  -- Create procedure to retrieve error information.  CREATE PROCEDURE usp\_GetErrorInfo  AS  SELECT  ERROR\_NUMBER() AS ErrorNumber  ,ERROR\_SEVERITY() AS ErrorSeverity  ,ERROR\_STATE() AS ErrorState  ,ERROR\_PROCEDURE() AS ErrorProcedure  ,ERROR\_LINE() AS ErrorLine  ,ERROR\_MESSAGE() AS ErrorMessage;  GO  BEGIN TRY  -- Generate divide-by-zero error.  SELECT 1/0;  END TRY  BEGIN CATCH  -- Execute error retrieval routine.  EXECUTE usp\_GetErrorInfo;  END CATCH; |

The results will be the following:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ErrorNumber | ErrorSeverity | ErrorState | ErrorProcedure | ErrorLine | ErrorMessage |
| 8134 | 16 | 1 | NULL | 4 | Divide by zero error encountered. |

## Errors Unaffected by a TRY…CATCH Construct

TRY…CATCH constructs do not trap the following conditions:

* Warnings or informational messages that have a severity of 10 or lower.
* Errors that have a severity of 20 or higher that stop the SQL Server Database Engine task processing for the session. If an error occurs that has severity of 20 or higher and the database connection is not disrupted, TRY…CATCH will handle the error.
* Attentions, such as client-interrupt requests or broken client connections.
* When the session is ended by a system administrator by using the KILL statement.

The following types of errors are not handled by a CATCH block when they occur at the same level of execution as the TRY…CATCH construct:

* Compile errors, such as syntax errors, that prevent a batch from running.
* Errors that occur during statement-level recompilation, such as object name resolution errors that occur after compilation because of deferred name resolution.

These errors are returned to the level that ran the batch, stored procedure, or trigger. If an error occurs during compilation or statement-level recompilation at a lower execution level (for example, when executing sp\_executesql or a user-defined stored procedure) inside the TRY block, the error occurs at a lower level than the TRY…CATCH construct and will be handled by the associated CATCH block. For more information, see [Using TRY...CATCH in Transact-SQL](http://msdn.microsoft.com/en-us/library/ms179296.aspx).

**Example 1:** The following example shows how an object name resolution error generated by a SELECT statement is not caught by the TRY…CATCH construct, but is caught by the CATCH block when the same SELECT statement is executed inside a stored procedure.

|  |
| --- |
| BEGIN TRY  -- Table does not exist; object name resolution  -- error not caught.  SELECT \* FROM [AdventureWorksLT].[SalesLT].[SalesOrderHeader];  END TRY  BEGIN CATCH  SELECT  ERROR\_NUMBER() AS ErrorNumber  ,ERROR\_MESSAGE() AS ErrorMessage;  END CATCH |

**Result:** The error is not caught and control passes out of the TRY…CATCH construct to the next higher level. Running the SELECT statement inside a stored procedure will cause the error to occur at a level lower than the TRY block. The error will be handled by the TRY…CATCH construct.

**Example 2:** Create a stored procedure that will cause an object resolution error.

|  |
| --- |
| -- Verify that the stored procedure does not exist.  IF OBJECT\_ID ( N'usp\_ExampleProc', N'P' ) IS NOT NULL  DROP PROCEDURE usp\_ExampleProc;  GO  -- Create a stored procedure that will cause an  -- object resolution error.  CREATE PROCEDURE usp\_ExampleProc  AS  SELECT \* FROM NonexistentTable;  GO  BEGIN TRY  EXECUTE usp\_ExampleProc;  END TRY  BEGIN CATCH  SELECT  ERROR\_NUMBER() AS ErrorNumber  ,ERROR\_MESSAGE() AS ErrorMessage;  END CATCH; |

This query will retrieve the following results:

|  |  |
| --- | --- |
| ErrorNumber | ErrorMessage |
| 208 | Invalid object name 'NonexistentTable'. |

**Example 3:** Using TRY…CATCH in a transaction. The following example shows how a TRY…CATCH block works inside a transaction. The statement inside the TRY block generates a constraint violation error.

|  |
| --- |
| USE AdventureWorks;  GO  BEGIN TRANSACTION;  BEGIN TRY  -- Generate a constraint violation error.  DELETE FROM Production.Product  WHERE ProductID = 980;  END TRY  BEGIN CATCH  SELECT  ERROR\_NUMBER() AS ErrorNumber  ,ERROR\_SEVERITY() AS ErrorSeverity  ,ERROR\_STATE() AS ErrorState  ,ERROR\_PROCEDURE() AS ErrorProcedure  ,ERROR\_LINE() AS ErrorLine  ,ERROR\_MESSAGE() AS ErrorMessage;  IF @@TRANCOUNT > 0  ROLLBACK TRANSACTION;  END CATCH;  IF @@TRANCOUNT > 0  COMMIT TRANSACTION;  GO |

This query will retrieve the following result:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Error  Number | Error  Severity | Error  State | Error  Procedure | Error  Line | ErrorMessage |
| 547 | 16 | 0 | NULL | 5 | The DELETE statement conflicted with the REFERENCE constraint "FK\_ProductCostHistory\_Product\_ProductID". The conflict occurred in database "AdventureWorks", table "Production.ProductCostHistory", column 'ProductID'. |

**Example 4:** Also returns a table (record-set) but can contain multiple TSQL statements or scripts and is defined in BEGIN END block.

|  |
| --- |
| CREATE FUNCTION [dbo].[GetContactSales](@ContactID int)  RETURNS @retSalesInfo TABLE (  [ContactID] INT NOT NULL,  [SalesOrderID] INT NULL,  [ProductID] INT NULL,  [Name] NVARCHAR(50) NULL,  [OrderDate] DATETIME NULL,  [DueDate] DATETIME NULL,  [ShipDate] DATETIME NULL,  [TotalDue] MONEY NULL,  [Status] TINYINT NULL,  [SalesPersonID] INT NULL)  AS  BEGIN  IF @ContactID IS NOT NULL  BEGIN  INSERT @retSalesInfo  SELECT  h.[ContactID],  h.[SalesOrderID],  p.[ProductID],  p.[Name],  h.[OrderDate],  h.[DueDate],  h.[ShipDate],  h.[TotalDue],  h.[Status],  h.[SalesPersonID]  FROM [AdventureWorks].[Sales].[SalesOrderHeader] AS h  JOIN [AdventureWorks].[Sales].[SalesOrderDetail] AS d ON d.SalesOrderID = h.SalesOrderID  JOIN Production.Product AS p ON p.ProductID = d.ProductID  WHERE ContactID = @ContactID  END  -- Return the recordsets  RETURN  END    --// Usage:  SELECT \* FROM GetContactSales(100) |

This query will retrieve the following result:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Contact  ID | Sales  OrderID | Product  ID | Name | OrderDate | DueDate | ShipDate | TotalDue | Status | Sales  PersonID |
| 100 | 57021 | 712 | AWC Logo Cap | 2003-11-01 00:00:00.000 | 2003-11-13 00:00:00.000 | 2003-11-08 00:00:00.000 | 265,6878 | 5 | 285 |
| 100 | 63139 | 712 | AWC Logo Cap | 2004-02-01 00:00:00.000 | 2004-02-13 00:00:00.000 | 2004-02-08 00:00:00.000 | 481,5265 | 5 | 285 |
| 100 | 63139 | 877 | Bike Wash - Dissolver | 2004-02-01 00:00:00.000 | 2004-02-13 00:00:00.000 | 2004-02-08 00:00:00.000 | 481,5265 | 5 | 285 |
| 100 | 69398 | 877 | Bike Wash - Dissolver | 2004-05-01 00:00:00.000 | 2004-05-13 00:00:00.000 | 2004-05-08 00:00:00.000 | 1107,7067 | 5 | 285 |
| 100 | 69398 | 859 | Half-Finger Gloves, M | 2004-05-01 00:00:00.000 | 2004-05-13 00:00:00.000 | 2004-05-08 00:00:00.000 | 1107,7067 | 5 | 285 |
| 100 | 57021 | 858 | Half-Finger Gloves, S | 2003-11-01 00:00:00.000 | 2003-11-13 00:00:00.000 | 2003-11-08 00:00:00.000 | 265,6878 | 5 | 285 |
| 100 | 51702 | 876 | Hitch Rack - 4-Bike | 2003-08-01 00:00:00.000 | 2003-08-13 00:00:00.000 | 2003-08-08 00:00:00.000 | 1980,8809 | 5 | 285 |
| 100 | 63139 | 996 | HL Bottom Bracket | 2004-02-01 00:00:00.000 | 2004-02-13 00:00:00.000 | 2004-02-08 00:00:00.000 | 481,5265 | 5 | 285 |
| 100 | 63139 | 715 | Long-Sleeve Logo Jersey, L | 2004-02-01 00:00:00.000 | 2004-02-13 00:00:00.000 | 2004-02-08 00:00:00.000 | 481,5265 | 5 | 285 |

…

20 rows

# Bibliography

1. MSDN Library SQL Online Help Guide (<http://msdn.microsoft.com/en-us/library/bb545450.aspx>);
2. Temporary Tables – (<http://decipherinfosys.wordpress.com/2007/05/04/temporary-tables-ms-sql-server/>);
3. Temporary table/Table variables(<http://www.mssqltips.com/sqlservertip/1556/differences-between-sql-server-temporary-tables-and-table-variables/>);
4. Table variables – ([*http://www.simple-talk.com/sql/t-sql-programming/temporary-tables-in-sql-server/*](http://www.simple-talk.com/sql/t-sql-programming/temporary-tables-in-sql-server/));
5. Collasce - (<http://www.mssqltips.com/sqlservertip/1521/the-many-uses-of-coalesce-in-sql-server/> );
6. Collate - (<http://www.databasejournal.com/features/mssql/article.php/3302341/SQL-Server-and-Collation.htm>) ;
7. DB Access Rights (Grant / Revoke) – (<http://beginner-sql-tutorial.com/sql-grant-revoke-privileges-roles.htm>)