

Version: 01.01

Date: 14-Dec-2011

Developed by: O.Cozac, C.Modval, I.Mustea-Botezatu

Verified by: Eugeniu Zincovschi

Intermediate Level Training Manual –

Endava SQL Discipline

Intermediate MS SQL Guide

Table of Contents

[1 Revision History 5](#_Toc315703669)

[2 Introduction 5](#_Toc315703670)

[3 Purpose of the Document 5](#_Toc315703671)

[4 Functions 6](#_Toc315703672)

[4.1 Numeric Functions 6](#_Toc315703673)

[4.1.1 Arithmetic Functions 6](#_Toc315703674)

[4.1.2 ROUND 6](#_Toc315703675)

[4.1.3 ABS 7](#_Toc315703676)

[4.2 String Functions 7](#_Toc315703677)

[4.2.1 String Concatenation (+) 8](#_Toc315703678)

[4.2.2 SUBSTRING 9](#_Toc315703679)

[4.2.3 LEFT and RIGHT 10](#_Toc315703680)

[4.2.1 RTRIM and LTRIM 10](#_Toc315703681)

[4.2.2 CHARINDEX 11](#_Toc315703682)

[4.2.1 SUBSTRING and CHARINDEX 12](#_Toc315703683)

[4.2.2 UPPER and LOWER 12](#_Toc315703684)

[4.2.3 LEN 13](#_Toc315703685)

[4.3 Conversion Functions 13](#_Toc315703686)

[4.3.1 CAST 13](#_Toc315703687)

[4.3.2 CONVERT 14](#_Toc315703688)

[4.4 Date Functions 15](#_Toc315703689)

[4.4.1 DATEADD 15](#_Toc315703690)

[4.4.2 DATEDIFF 16](#_Toc315703691)

[4.4.3 DATEPART 17](#_Toc315703692)

[4.4.4 YEAR 17](#_Toc315703693)

[4.4.5 MONTH 18](#_Toc315703694)

[4.4.6 DAY 19](#_Toc315703695)

[4.4.7 GETDATE 19](#_Toc315703696)

[5 Working with Nulls 20](#_Toc315703697)

[6 GROUP BY 22](#_Toc315703698)

[7 HAVING 26](#_Toc315703699)

[8 Create Table 28](#_Toc315703700)

[8.1 Data Types 28](#_Toc315703701)

[8.2 SQL Constraints 29](#_Toc315703702)

[8.2.1 Primary and Foreign Keys 29](#_Toc315703703)

[8.2.2 UNIQUE 30](#_Toc315703704)

[8.2.3 CHECK 31](#_Toc315703705)

[8.2.4 DEFAULT 32](#_Toc315703706)

[9 INSERT INTO…SELECT 33](#_Toc315703707)

[10 Alter Table 35](#_Toc315703708)

[11 Drop Table 35](#_Toc315703709)

[12 Join Tables 36](#_Toc315703710)

[12.1 Cross Join 36](#_Toc315703711)

[12.2 INNER JOIN 37](#_Toc315703712)

[12.3 OUTER JOIN 38](#_Toc315703713)

[12.3.1 LEFT OUTER JOIN 39](#_Toc315703714)

[12.3.2 RIGHT OUTER JOIN 40](#_Toc315703715)

[12.3.3 FULL OUTER JOIN 41](#_Toc315703716)

[12.4 Self-Join 42](#_Toc315703717)

[12.5 Joining More Tables 44](#_Toc315703718)

[13 Set Operations 45](#_Toc315703719)

[13.1 UNION 46](#_Toc315703720)

[13.2 UNION ALL 48](#_Toc315703721)

[13.3 EXCEPT 51](#_Toc315703722)

[13.4 INTERSECT 53](#_Toc315703723)

[14 Sub-Queries 54](#_Toc315703724)

[15 Derived Structures 56](#_Toc315703725)

[15.1 Views 56](#_Toc315703726)

[15.1.1 With Check Option 57](#_Toc315703727)

[15.1.2 Create Views 57](#_Toc315703728)

[15.1.3 Update Views 60](#_Toc315703729)

[15.1.4 Delete Views 60](#_Toc315703730)

[15.2 Inline Views 61](#_Toc315703731)

[15.3 Common Table Expression 62](#_Toc315703732)

[16 Programming Logic 64](#_Toc315703733)

[16.1 IF…ELSE 64](#_Toc315703734)

[16.2 IF…EXISTS 66](#_Toc315703735)

[16.3 CASE 67](#_Toc315703736)

[16.4 BEGIN TRAN…END TRAN 70](#_Toc315703737)

[17 Bibliography 73](#_Toc315703738)

[A. APPENDIX: Numeric Functions 74](#_Toc315703739)

[B. APPENDIX: Data Types 74](#_Toc315703740)

[C. APPENDIX: Implicit Conversions 76](#_Toc315703741)

[D. APPENDIX: Create View Examples 77](#_Toc315703742)

# Revision History

|  |  |  |
| --- | --- | --- |
| **Revision** | **Date of revision** | **Description of modifications** |
| 01.01 | 14-Dec-2011 | Initial version of the document. |

# 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 intermediate 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:

* FUNCTIONS
* GROUP BY
* HAVING
* CREATE TABLE
* DATA TYPES
* CONSTRAINTS
* INSERT INTO…SELECT
* ALTER
* DROP
* JOINS
* UNION
* EXCEPT
* INTERSECT
* SUB-QUERIES
* VIEWS
* IF…ELSE
* IF…EXISTS
* CASE
* BEGIN…END TRANSACTION

# Functions

This chapter introduces several useful built-in functions, which can be either names or row-level functions. Row-level functions operate one row at a time. In SQL Server, the row-level functions can be grouped into four major types:

* [Numeric](#_Numeric_Functions)
* [String](#_String_Functions)
* [Conversion](#_Conversion_Functions)
* [Date](#_Date_Functions)

## Numeric Functions

Numeric functions are used for calculations. They transform a value found in a row of a table.

### Arithmetic Functions

Strictly speaking, arithmetic transformations on values are functions performed in a result set. For an arithmetic example, we will use the SalesOrderHeader table, applying subtraction, division, multiplication and add:

use AdventureWorksLT\_EZ

select top 10 a.SubTotal, a.SubTotal+100 as "SubTotal + 100",

a.TaxAmt, a.TaxAmt - 50 as "TaxAmt - 50",

a.Freight, a.Freight/100 as "Freight/100",

a.TotalDue, a.TotalDue\*0.33 as "TotalDue\*0.33"

from SalesLT.SalesOrderHeader a

The column under arithmetic operation is a “read-only” value or “display-only” result set value. The SubTotal, TaxAmt, Freight and TotalDue fields in the SalesOrderHeader table are not actually changing. We are only displaying the results of arithmetic operations applied to the columns. To actually change column value in the SalesOrderHeader table, we would have to use the UPDATE command.

Any other combination of arithmetic operations may be performed on numeric data in result sets.

### ROUND

For any function to perform a transformation, it is necessary to specify:

* **Function name** – the name of the function, and
* **Argument list** – the “input” to the function.

The ROUND function, a built-in function in SQL Server, rounds numbers to a specified number of decimal places. This function has the basic form:

**ROUND(target value, precision)** where “target value” is what you want to round, and “precision” is the number of decimal places that you desire.

use AdventureWorksLT\_EZ

select top 10 a.SubTotal, round(a.SubTotal,2)

from SalesLT.SalesOrderHeader a

|  |  |
| --- | --- |
| SubTotal | RoundedValues |
| 1427.592 | 1427.59 |
| 78.81 | 78.81 |
| 38418.69 | 38418.69 |
| 39785.33 | 39785.33 |
| 83858.43 | 83858.43 |
| 108561.8 | 108561.8 |
| 57634.63 | 57634.63 |
| 78029.69 | 78029.69 |
| 1141.578 | 1141.58 |
| 3398.166 | 3398.17 |

### ABS

The ABS function returns the absolute value of a numeric expression. For example, if you wanted to find the absolute value of –1000, use the next query:

SELECT ABS(-1000) AS [Absolute Value]

The query will retun:

|  |
| --- |
| Absolute Value |
| 1000 |

Find more frequently used in-row numeric functions, see the [Numeric Functions](#_APPENDIX:_Numeric_Functions) appendix.

## String Functions

In this chapter, we will use string functions for the extraction of parts of a string, functions to find the length of a string, and functions to find matching characters in strings.

String functions are not aggregates; they are row-level functions, as they operate on one value in one row at a time. String functions are read-only functions and will not change the underlying data in the database unless the UPDATE command is used.

### String Concatenation (+)

String concatenation, which means to connect things together can be performed using the " + " operator.

For example, we have first and last name specified in two different columns:

select top 10 a.FirstName, a.LastName

from SalesLT.Customer a

The output is:

|  |  |
| --- | --- |
| FirstName | LastName |
| Orlando | Gee |
| Keith | Harris |
| Donna | Carreras |
| Janet | Gates |
| Lucy | Harrington |
| Rosmarie | Carroll |
| Dominic | Gash |
| Kathleen | Garza |
| Katherine | Harding |
| Johnny | Caprio |

The result-set of concatenating the First Name and Last Name columns, delimited by comma is as follows:

select top 10 (a.FirstName + ', ' + a.LastName) as 'Fisrt Name + Last name'

from SalesLT.Customer a

|  |
| --- |
| First Name + Last Name |
| Orlando, Gee |
| Keith, Harris |
| Donna, Carreras |
| Janet, Gates |
| Lucy, Harrington |
| Rosmarie, Carroll |
| Dominic, Gash |
| Kathleen, Garza |
| Katherine, Harding |
| Johnny, Caprio |

### SUBSTRING

The SUBSTRING function returns part of a string. The SUBSTRING function syntax is as follows:

SUBSTRING (stringexpression, startposition, length)

where:

* **stringexpression** is the column that we will be using,
* **startposition** tells where in the “stringexpression” to start retrieving characters from
* **length** tells how many characters to extract.

All three parameters are required in SUBSTRING function.

For example, consider the following:

select top 10 a.FirstName, a.LastName, (SUBSTRING(a.LastName, 1,1)+'.') as 'First Letter of LName'

from SalesLT.Customer a

|  |  |  |
| --- | --- | --- |
| FirstName | LastName | First Letter of LastName |
| Orlando | Gee | G. |
| Keith | Harris | H. |
| Donna | Carreras | C. |
| Janet | Gates | G. |
| Lucy | Harrington | H. |
| Rosmarie | Carroll | C. |
| Dominic | Gash | G. |
| Kathleen | Garza | G. |
| Katherine | Harding | H. |
| Johnny | Caprio | C. |

### LEFT and RIGHT

These functions return a portion of a string, starting from either the left or right side of “stringexpression.” The LEFT and RIGHT function syntax is as bellow:

**LEFT(stringexpression, n)**

**RIGHT(stringexpression, n)**

select top 5 a.FirstName, a.LastName, LEFT(a.LastName, 2) as 'Left'

from SalesLT.Customer a

|  |  |  |
| --- | --- | --- |
| FirstName | LastName | Left |
| Orlando | Gee | Ge |
| Keith | Harris | Ha |
| Donna | Carreras | Ca |
| Janet | Gates | Ga |
| Lucy | Harrington | Ha |

select top 5 a.FirstName, a.LastName, RIGHT(a.LastName, 2) as 'Right'

from SalesLT.Customer a

|  |  |  |
| --- | --- | --- |
| FirstName | LastName | Right |
| Orlando | Gee | ee |
| Keith | Harris | is |
| Donna | Carreras | as |
| Janet | Gates | es |
| Lucy | Harrington | on |

### RTRIM and LTRIM

The RTRIM and LTRIM functions remove spaces from the right side (RTRIM) or left side (LTRIM) of a string. You may need to use these functions when working with fixed-length data types (CHAR and NCHAR) or to clean up flat-file data before it is loaded from a staging database into a data warehouse.

The syntax is simple:

RTRIM(<string>)

LTRIM(<string>)

The data in the AdventureWorks2008 does not contain any extra spaces, so we created “the artificial” spaces. Find an example below of treating these.

Using RTRIM and LTRIM

select top 5 a.FirstName, a.LastName,

('\*' + ' ' + a.FirstName + ' ' + a.LastName + ' ')+ '\*' Full\_Name\_WithSpaces,

'\*' + ltrim( ' ' + a.FirstName + ' ' + a.LastName + ' ') + '\*' as Full\_Name\_Ltrim,

'\*' + RTRIM( ' ' + a.FirstName + ' ' + a.LastName + ' ') + '\*' as Full\_Name\_RTRIM

from SalesLT.Customer a

Find results in the table below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| FirstName | LastName | Full\_Name\_WithSpaces | Full\_Name\_Ltrim | Full\_Name\_RTRIM |
| Orlando | Gee | \* Orlando Gee \* | \*Orlando Gee \* | \* Orlando Gee\* |
| Keith | Harris | \* Keith Harris \* | \*Keith Harris \* | \* Keith Harris\* |
| Donna | Carreras | \* Donna Carreras \* | \*Donna Carreras \* | \* Donna Carreras\* |
| Janet | Gates | \* Janet Gates \* | \*Janet Gates \* | \* Janet Gates\* |
| Lucy | Harrington | \* Lucy Harrington \* | \*Lucy Harrington \* | \* Lucy Harrington\* |

### CHARINDEX

The CHARINDEX function returns the starting position of a specified pattern. For example, if you wish to find the position of a space in the CompanyName column in the SalesLT.Customer table, use:

select top 5 a.CompanyName, CHARINDEX(' ', a.CompanyName) as 'Position of Blank'

from SalesLT.Customer a

|  |  |
| --- | --- |
| CompanyName | Position of Blank |
| A Bike Store | 2 |
| Progressive Sports | 12 |
| Advanced Bike Components | 9 |
| Modular Cycle Systems | 8 |
| Metropolitan Sports Supply | 13 |

### SUBSTRING and CHARINDEX

Try a more complex query using SUBSTRING and CHARINDEX functions in one statement:

use AdventureWorksLT\_EZ

select top 5 a.CompanyName,

(SUBSTRING(a.CompanyName, 1,(CHARINDEX(' ', a.CompanyName)-1)) + ','

+ SUBSTRING(a.CompanyName, CHARINDEX(' ', a.CompanyName), 2))as 'Abreviation'

from SalesLT.Customer a

The result-set is:

|  |  |
| --- | --- |
| CompanyName | Abreviation |
| A Bike Store | A, B |
| Progressive Sports | Progressive, S |
| Advanced Bike Components | Advanced, B |
| Modular Cycle Systems | Modular, C |
| Metropolitan Sports Supply | Metropolitan, S |

### UPPER and LOWER

To produce all the output in uppercase or in lowercase, you can use the UPPER or LOWER functions. For example, use of these two functions will produce the following results:

use AdventureWorksLT\_EZ

select top 5 a.CompanyName, lower(a.CompanyName) as 'Lower Case',

upper(SUBSTRING(a.CompanyName, 1,(CHARINDEX(' ', a.CompanyName)-1) ) + '.')as 'Upper Case'

from SalesLT.Customer a

|  |  |  |
| --- | --- | --- |
| CompanyName | Lower Case | Upper Case |
| A Bike Store | a bike store | A. |
| Progressive Sports | progressive sports | PROGRESSIVE. |
| Advanced Bike Components | advanced bike components | ADVANCED. |
| Modular Cycle Systems | modular cycle systems | MODULAR. |
| Metropolitan Sports Supply | metropolitan sports supply | METROPOLITAN. |

### LEN

The LEN function returns the length (number of characters) of a desired string excluding trailing blanks. For example, to list the lengths of the full names (including any spaces) in the salesLT.Customer table, build the statement as follows:

select top 5 a.CompanyName, LEN(a.CompanyName) as 'Company Name Lenght'

from SalesLT.Customer a

The result-set is the following:

|  |  |
| --- | --- |
| CompanyName | Company Name Lenght |
| A Bike Store | 12 |
| Progressive Sports | 18 |
| Advanced Bike Components | 24 |
| Modular Cycle Systems | 21 |
| Metropolitan Sports Supply | 26 |

## Conversion Functions

Conversion functions are used to convert data from one data type to another.

Sometimes data in a table is stored with one data type, and you need to have the data in another data type. SQL Server provides three functions for converting data types—CAST, CONVERT, and STR. In this guide we will use CAST and CONVERT functions.

### CAST

The CAST result-set can be used for:

* Concatenating strings
* Performing unions of tables
* Performing mathematical operations on columns that were defined as
* Characters but actually contain numbers that need to be calculated

|  |
| --- |
| **NOTE:** Some conversions are automatic and implicit, so using CAST may not always be necessary. For example, converting between numbers with types INT, SMALLINT, TINYINT, FLOAT, NUMERIC, etc., is done automatically and implicitly as long as an overflow does not occur. But converting numbers with decimal places to integer data types truncates values to the right of the decimal place without a warning, so you should use CAST if a loss of precision is possible. |

The syntax for the CAST function is:

CAST (original expression AS desired datatype)

**Example:**

select top 5 a.OrderQty,

cast(a.OrderQty as varchar(5)) + ' Number of Items' as 'CastApplied'

from SalesLT.SalesOrderDetail a

|  |
| --- |
| **NOTE:** Please refer to **Appendix C. – Implicit Conversions** in order to see which formats may be converted. |

The result-set is the following:

|  |  |
| --- | --- |
| OrderQty | CastApplied |
| 3 | 3 Number of Items |
| 1 | 1 Number of Items |
| 1 | 1 Number of Items |
| 4 | 4 Number of Items |
| 2 | 2 Number of Items |

### CONVERT

Like the CAST function, the CONVERT function is also used to explicitly convert to a given data type. The general syntax for the CONVERT function is:

CONVERT(<new data type>,<value>)

**Example:**

use AdventureWorksLT\_EZ

select top 5 a.OrderQty,

convert(varchar(5), a.OrderQty ) + 'item(s)'

as 'CONVERT is applied'

from SalesLT.SalesOrderDetail a

The result-set is as follows:

|  |  |
| --- | --- |
| OrderQty | CONVERT is applied |
| 3 | 3 item(s) |
| 1 | 1 item(s) |
| 1 | 1 item(s) |
| 4 | 4 item(s) |
| 2 | 2 item(s) |

The functions CAST and CONVERT have very different syntaxes, but they accomplish the same results in this situation.

|  |
| --- |
| **NOTE:** The majority of programmers prefer CAST over CONVERT because CAST is it is compliant with the ANSI SQL-99 standard. |

## Date Functions

Date functions operate on date data to extract days or years or calculate dates.

SQL Server provides several date functions like DAY, MONTH, YEAR, DATEADD, DATEDIFF, DATEPART, and GETDATE that can be used for extracting and manipulating dates (add dates, find differences between dates, find the day/month/year from dates etc.).

### DATEADD

The DATEADD function produces a date by adding a specified number to a specified part of a date. The date parts are: DD – for day, MM – for month, and YY – for year.

The format for the DATEADD function is:

DATEADD(datepart, number, datetime)

May refer to the example of the DATEADD Statements:

DECLARE

@datetime2 datetime2 ;

Set @datetime2 = GETDATE();

SELECT 'getdate', GETDATE()

UNION ALL

SELECT 'year', DATEADD(year,1,@datetime2)

UNION ALL

SELECT 'quarter',DATEADD(quarter,1,@datetime2)

UNION ALL

SELECT 'month',DATEADD(month,1,@datetime2)

UNION ALL

SELECT 'dayofyear',DATEADD(dayofyear,1,@datetime2)

UNION ALL

SELECT 'day',DATEADD(day,1,@datetime2)

UNION ALL

SELECT 'week',DATEADD(week,1,@datetime2)

UNION ALL

SELECT 'weekday',DATEADD(weekday,1,@datetime2)

UNION ALL

SELECT 'hour',DATEADD(hour,1,@datetime2)

UNION ALL

SELECT 'minute',DATEADD(minute,1,@datetime2)

UNION ALL

SELECT 'second',DATEADD(second,1,@datetime2)

UNION ALL

SELECT 'millisecond',DATEADD(millisecond,1,@datetime2)

UNION ALL

SELECT 'microsecond',DATEADD(microsecond,1,@datetime2)

UNION ALL

SELECT 'nanosecond',DATEADD(nanosecond,1,@datetime2);

The results may find below:

|  |  |
| --- | --- |
| Command used | Date |
| getdate | 2011-08-03 23:24:30.593 |
| year | **2012**-08-03 23:24:30.593 |
| quarter | 2011-**11**-03 23:24:30.593 |
| month | 2011-**09**-03 23:24:30.593 |
| dayofyear | 2011-08-**04** 23:24:30.593 |
| day | 2011-08-**04** 23:24:30.593 |
| week | 2011-08-**10** 23:24:30.593 |
| weekday | 2011-08-**04** 23:24:30.593 |
| hour | 2011-08-04 **00**:24:30.593 |
| minute | 2011-08-03 23:**25**:30.593 |
| second | 2011-08-03 23:24:**31**.593 |
| millisecond | 2011-08-03 23:24:30.**594** |

### DATEDIFF

The DATEDIFF function returns the difference between two parts of a date. The DATEDIFF function syntax is:

DATEDIFF(datepart, datetime1, datetime2)

For example, to find the difference between the OrderDate and ShipDate, create the next query:

select top 5 a.OrderDate,a.ShipDate, DATEDIFF(DD, a.OrderDate, a.ShipDate)as 'Ship - Order'

from SalesLT.SalesOrderHeader a

Script above will produce the following results:

|  |  |  |
| --- | --- | --- |
| OrderDate | ShipDate | Ship - Order |
| 01/06/2008 | 08/06/2008 | 7 |
| 01/06/2008 | 08/06/2008 | 7 |
| 01/06/2008 | 08/06/2008 | 7 |
| 01/06/2008 | 08/06/2008 | 7 |
| 01/06/2008 | 08/06/2008 | 7 |

### DATEPART

The DATEPART function returns the specified part of the date requested. The DATEPART syntax is:

**DATEPART(datepart, datetime)**

**Example:**

DECLARE

@datetime2 datetime2 ;

Set @datetime2 = GETDATE();

Select GETDATE(), DATEPART(YYYY, @datetime2) as 'YYYY, OrderDate',

DATEPART(MONTH, @datetime2) as 'MONTH, ShipDate',

DATEPART(DD, @datetime2) as 'DD, duedate',

DATEPART(HOUR, @datetime2) as 'HOUR, duedate'

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| OrderDate | YYYY, OrderDate | MONTH, ShipDate | DD, duedate | HOUR, duedate |
| 2012-01-30 06:13:17.310 | 2012 | 1 | 30 | 6 |

### YEAR

The YEAR (column) function will extract the year from a value stored as a DATETIME2 data type.

1. Extract the year from the OrderDate:

select top 5

a.OrderDate , YEAR(a.OrderDate)as 'OrderDate YEAR'

from SalesLT.SalesOrderHeader a

|  |  |
| --- | --- |
| OrderDate | OrderDate YEAR |
| 01/06/2008 00:00 | 2008 |
| 01/06/2008 00:00 | 2008 |
| 01/06/2007 00:00 | 2007 |
| 01/06/2013 00:00 | 2013 |
| 01/06/2008 00:00 | 2008 |

We can also use the YEAR function in date calculations.

1. Find the number of years between SellEndDate and SellStartDate for a product type:

select top 5 p.SellStartDate, p.SellEndDate,

isnull(YEAR(p.SellEndDate),0) - isnull(YEAR(p.SellStartDate),0)as Difference

from SalesLT.Product p where p.SellEndDate is not null

|  |  |  |
| --- | --- | --- |
| SellStartDate | SellEndDate | Difference |
| 01/07/2005 00:00 | 30/06/2006 00:00 | 1 |
| 01/07/2005 00:00 | 30/06/2006 00:00 | 1 |
| 01/07/2005 00:00 | 30/06/2007 00:00 | 2 |
| 01/07/2005 00:00 | 30/06/2007 00:00 | 2 |
| 01/07/2005 00:00 | 30/06/2007 00:00 | 2 |

(Note: Find more on IS NULL operator uses in the following chapter [Working with NULLs](#_Working_with_Nulls_1).)

### MONTH

The MONTH function will extract the month from a date. Using MONTH function, you can also increase month by a desired number. Firstly, extract all months and then add or extract a number, “2” in the example below:

select top 5 p.SellStartDate,

MONTH(p.SellStartDate) as 'Month of Sell Date',

(MONTH(p.SellStartDate)+2)as 'Month of Sell Date + 2'

from SalesLT.Product p

This query will retrieve the following results:

|  |  |  |
| --- | --- | --- |
| SellStartDate | Month of Sell Date | Month of Sell Date + 2 |
| 01/06/2002 00:00 | 6 | 8 |
| 01/06/2002 00:00 | 6 | 8 |
| 01/07/2005 00:00 | 7 | 9 |
| 01/07/2005 00:00 | 7 | 9 |
| 01/07/2005 00:00 | 7 | 9 |

### DAY

The DAY function extracts the day of the month from a date.

select top 5 p.SellEndDate,

DAY(p.SellEndDate)as 'Sell End Day'

from SalesLT.Product p

where p.SellEndDate is not null

|  |  |
| --- | --- |
| SellEndDate | Sell End Day |
| 30/06/2006 00:00 | 30 |
| 30/06/2006 00:00 | 30 |
| 30/06/2007 00:00 | 30 |
| 30/06/2007 00:00 | 30 |
| 30/06/2007 00:00 | 30 |

(Note: Find more on NULL operator uses in the following chapter [Working with NULLs](#_Working_with_Nulls_1).)

### GETDATE

The GETDATE function returns the current system date and time.

1. Get today’s date.

SELECT 'Today ' = GETDATE()

|  |
| --- |
| Today |
| 04/01/2012 01:17 |

1. Find the number of years between today (the current date) and the date of birth:

select DATEDIFF(YYYY,'23 march 1953', GETDATE()) Difference

|  |
| --- |
| Difference |
| 59 |

# Working with Nulls

The NULL value might be confusing at the beginning; but you will quickly get used to it while practicing. Conceptually, NULL means “a missing unknown value”, and it is treated differently from other values. To test for NULL, you cannot use the arithmetic comparison operators such as =, <, or <>.

To help with NULL handling, you can use the IS NULL and IS NOT NULL operators.

In SQL, the NULL value is never true in comparison to any other value, even NULL. An expression that contains NULL always produces a null value. In row-level functions, if a null value is included in calculations, the result is always null.

The IS NULL function has the following form:

IS NULL (check\_expression, replacement\_value)

1. Where *check\_expression* can be of any type and the function returns the same type as *check\_expression.*

DECLARE @a FLOAT, @b FLOAT

SET @a = NULL

SET @b = 2

select @a + @b as 'A + B not applying ISNULL:'

The result is the following:

|  |
| --- |
| A + B not applying ISNULL: |
| NULL |

1. The following select and use of IS NULL function solve problems with NULL. The IS NULL function says that if the expression (or column value) is not null, return the value, but if the value is null, return ValueIfNull:

DECLARE @a FLOAT, @b FLOAT

SET @a = NULL

SET @b = 2

SELECT ISNULL(@a, 0) + ISNULL(@b, 0) AS 'A + B = '

The result is the following:

|  |
| --- |
| A + B = |
| 2 |

1. Aggregate (summary) functions such as COUNT(), MIN(), and SUM() ignore NULL values. The exception to this is COUNT(\*), which counts rows and not individual column values. Follow the examples below:

use AdventureWorksLT\_EZ

select AVG(a.Weight) 'Without NULLs treatment' from SalesLT.Product a

|  |
| --- |
| Without NULLs treatment |
| 5483.705606 |

1. Different result of AVG function is reached by applying ISNULL function:

select AVG(isnull(a.Weight,0)) 'With Null Treatment' from SalesLT.Product a

|  |
| --- |
| With Null Treatment |
| 3680.588847 |

select COUNT(\*) CountAll from SalesLT.Product a

|  |
| --- |
| CountAll |
| 295 |

select COUNT(a.Weight) CountOfSpecifitColumn from SalesLT.Product a

|  |
| --- |
| CountOfSpecifitColumn |
| 198 |

1. The following queries are equivalent and return the same row number:

select COUNT(a.Weight) CountOfSpecifitColumn1 from SalesLT.Product a

union all

select COUNT(\*) CountOfSpecifitColumn2 from SalesLT.Product a

where a.Weight is not null

|  |
| --- |
| CountOfSpecifitColumn1 |
| 198 |
| 198 |

|  |  |  |
| --- | --- | --- |
| **NOTE:** Never use comparison like <, >, !=, = to find values different of NULLS. Use IS NOT NULL operator. The query below returns wrong result:  select COUNT(\*) CountOfSpecifitColumn from SalesLT.Product a  where a.Weight <> null   |  | | --- | | CountOfSpecifitColumn | | 0 | |

# GROUP BY

The SQL GROUP BY statement is used together with the SQL aggregate functions to group the retrieved data by one or more columns. The GROUP BY concept is one of the most complicated concepts for people new to the SQL language and the easiest way to understand it is by examples.

1. Calculate the average for the Unit Price and the sum for the LinetTotal and group by ProductID (to select just distinct ProductID).

We use the following SELECT statement:

SELECT ProductID, AVG(UnitPrice) AS 'Average Price',

SUM(LineTotal) AS Total

FROM [AdventureWorks].[Sales].[SalesOrderDetail]

GROUP BY ProductID

ORDER BY ProductID

The results are:

|  |  |  |
| --- | --- | --- |
| ProductID | Average Price | Total |
| 707 | 30,8865 | 157772.394392 |
| 708 | 30,4495 | 160869.517836 |
| 709 | 5,656 | 6060.388200 |
| 710 | 5,70 | 513.000000 |
| 711 | 30,365 | 165406.617049 |
| 712 | 7,6682 | 51229.445623 |
| 713 | 49,99 | 21445.710000 |
| 714 | 36,7811 | 115249.214976 |
| 715 | 34,901 | 198754.975360 |
| 716 | 37,165 | 95611.197080 |
| 717 | 814,0413 | 394255.572400 |
| 718 | 813,2804 | 395182.699300 |
| 719 | 821,6336 | 89872.173600 |
| 722 | 188,3567 | 177635.904000 |

*...*

*266 rows selected*

1. Calculate the average for the Unit Price and the sum for the LinetTotal and group by ProductID to select just distinct ProductID and we will select and distinct SpecialOfferID (compare this example with example above, pay attention of the ProductID and SpecialOfferID)

We use the following SELECT statement:

SELECT ProductID, SpecialOfferID, AVG(UnitPrice) AS 'Average Price',

SUM(LineTotal) AS SubTotal FROM [AdventureWorks].[Sales].[SalesOrderDetail]

GROUP BY ProductID, SpecialOfferID

ORDER BY ProductID,SpecialOfferID

The results are:

|  |  |  |  |
| --- | --- | --- | --- |
| ProductID | SpecialOfferID | Average Price | SubTotal |
| 707 | **1** | 31,3436 | 141271.252000 |
| 707 | **2** | 20,0556 | 8886.245452 |
| 707 | **3** | 18,9272 | 2191.058910 |
| 707 | **8** | 16,8221 | 2452.662180 |
| 707 | **11** | 15,7455 | 2971.175850 |
| 708 | **1** | 30,9648 | 140403.764500 |
| 708 | **2** | 20,0502 | 11689.730276 |
| 708 | **3** | 18,9753 | 3461.676690 |
| 708 | **8** | 16,8221 | 2316.403170 |
| 708 | **11** | 15,7455 | 2997.943200 |
| 709 | **1** | 5,70 | 4235.100000 |
| 709 | **2** | 5,51 | 723.573200 |
| 709 | **3** | 5,225 | 853.765000 |

*...*

*484 rows selected*

1. Calculate the average for Unit Price and just for distinct ProductID, then order by average unit price.

We use the following SELECT statement:

SELECT ProductID, AVG(UnitPrice) AS 'Average Price'

FROM [AdventureWorks].[Sales].[SalesOrderDetail]

GROUP BY ProductID

ORDER BY AVG(UnitPrice)

The results are:

|  |  |
| --- | --- |
| ProductID | Average Price |
| 873 | 2,2454 |
| 922 | 3,99 |
| 870 | 4,799 |
| 921 | 4,99 |
| 923 | 4,99 |
| 709 | 5,656 |
| 710 | 5,70 |
| 877 | 6,9361 |

*...*

*266 rows selected*

1. Calculate the average for Unit Price where OrderQTY is more that 10 and just for distinct ProductID, then order by average unit price.

We use the following SELECT statement:

SELECT ProductID, AVG(UnitPrice) AS 'Average Price'

FROM [AdventureWorks].[Sales].[SalesOrderDetail]

WHERE OrderQty > 10

GROUP BY ProductID

ORDER BY AVG(UnitPrice);

The results are:

|  |  |
| --- | --- |
| ProductID | Average Price |
| 873 | 1,3282 |
| 870 | 2,8325 |
| 877 | 4,5343 |
| 712 | 5,0313 |
| 875 | 5,08 |
| 874 | 5,2142 |
| 709 | 5,3243 |
| 844 | 11,5942 |
| 952 | 11,7392 |
| 859 | 13,948 |
| 858 | 14,1127 |
| 860 | 14,2042 |
| 843 | 14,50 |

*...*

*114 rows selected*

# HAVING

The SQL HAVING clause is used in conjunction with the SELECT clause to specify a search condition for a group or aggregate. The HAVING clause behaves like the WHERE clause, but is applicable to groups - the rows in the result set representing groups. In contrast the WHERE clause is applied to individual rows, not to groups.

1. Calculate average per Product ID and display products having quantity less than 5.

We use the following SELECT statement:

SELECT ProductID

FROM [AdventureWorks].[Sales].[SalesOrderDetail]

GROUP BY ProductID

HAVING AVG(OrderQty) > 5

ORDER BY ProductID

The results are:

|  |
| --- |
| ProductID |
| 862 |
| 863 |
| 864 |

1. Calculate the sum for line total and group by ProductID where sum(LineTotal) more than $2000000.00

We use the following SELECT statement:

SELECT ProductID, SUM(LineTotal) as Total

FROM [AdventureWorks].[Sales].[SalesOrderDetail]

GROUP BY ProductID

HAVING SUM(LineTotal) > $2000000.00

The results are:

|  |  |
| --- | --- |
| ProductID | Total |
| 779 | 3693678.025272 |
| 782 | 4400592.800400 |
| 793 | 2516857.314918 |
| 794 | 2347655.953454 |
| 780 | 3438478.860423 |
| 783 | 4009494.761841 |
| 795 | 2012447.775000 |
| 781 | 3434256.941928 |
| 784 | 3309673.216908 |

1. SelectdistinctSalesOrderID, CarrierTrackingNumber from the [Sales].[SalesOrderDetail] where CarrierTrackingNumber contains '4BD%'.

We use the following SELECT statement:

SELECT SalesOrderID, CarrierTrackingNumber

FROM [AdventureWorks].[Sales].[SalesOrderDetail]

GROUP BY SalesOrderID, CarrierTrackingNumber

HAVING CarrierTrackingNumber LIKE '4BD%'

ORDER BY SalesOrderID

The results are:

|  |  |
| --- | --- |
| SalesOrderID | CarrierTrackingNumber |
| 44511 | 4BD6-4CFA-B8 |
| 47411 | 4BDD-4511-AC |
| 50749 | 4BD9-4D6C-83 |
| 63188 | 4BD6-4AB7-A0 |
| 67348 | 4BD1-4782-A3 |

# Create Table

The CREATE TABLE statement is used to create a table in a database.

CREATE TABLE works under the next syntax:

CREATE TABLE table\_name  
(  
column\_name1 data\_type,  
column\_name2 data\_type,  
column\_name3 data\_type,  
....  
)

The table columns are defined using three basic properties:

* Column name
* [Data type](#_Data_Types)
* [Nullability](#_Working_with_NULLs)

1. Create a table Product\_List containing the next four columns: Product\_ID, Name, Description, and Cost.

CREATE TABLE Product\_List

(

Product\_ID INT NOT NULL,

Name VARCHAR(255) NOT NULL,

Description VARCHAR(1000) NULL,

Cost VARCHAR(255) NULL

)

Now, you can populate the table Product\_List with some data (see more details on INSERT in [Insert Into…Select](#_Insert_into..Select)).

## Data Types

In SQL Server, each column, local variable, expression, and parameter has a related data type. A data type is an attribute that specifies the type of data that the object can hold: integer data, character data, monetary data, date and time data, binary strings, and so on.

There is a set of system data types that define all the types of data that can be used with SQL Server. Developers can also define their own data types.

Data types in SQL Server are organised into the following categories:

|  |  |
| --- | --- |
| Category | Description |
| Exact numerics | Unicode character strings |
| Approximate numerics | Binary strings |
| Date and time | Other data types |
| Character strings |  |

The list of data types and their description, you can find in the [Data Types](#_APPENDIX:_Data_Types) appendix.

## SQL Constraints

Constraints are used to limit the type of data that can go into a table. Constraints can be specified when a table is created (with the [CREATE TABLE](#_Create_Table) statement) or after the table is created (with the [ALTER TABLE](#_Alter_Table) statement).

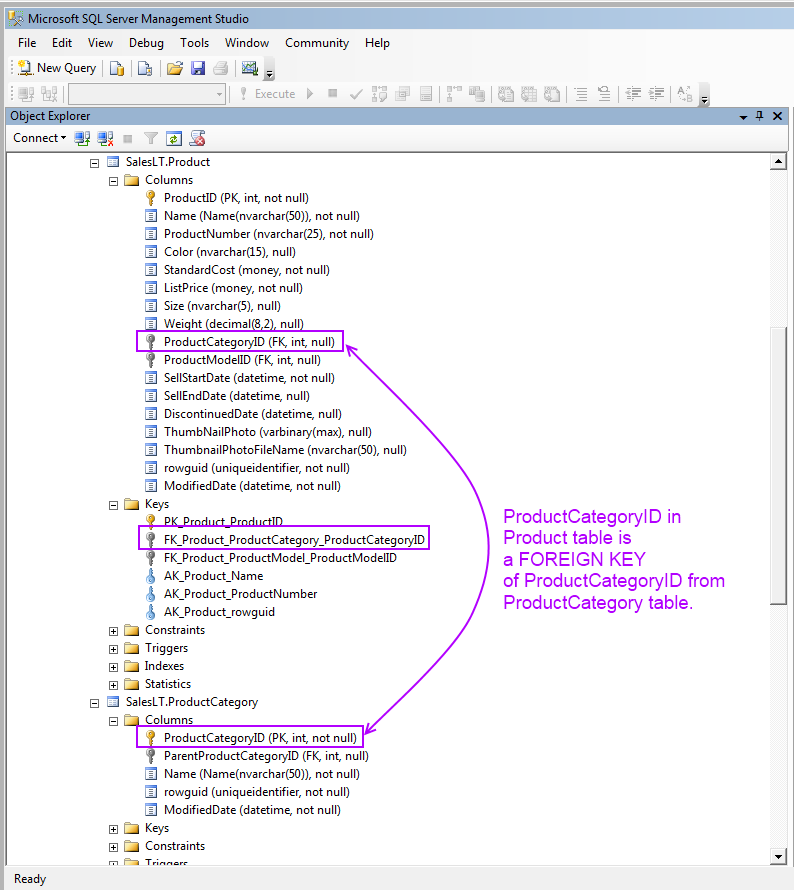
We will focus on the following constraints:

* [NOT NULL or NULL](#_Working_with_NULLs)
* [PRIMARY and FOREIGN KEYs](#_Primary_and_Foreign)
* [UNIQUE](#_Unique)
* [CHECK](#_Check)
* [DEFAULT](#_Default)

### Primary and Foreign Keys

A PRIMARY KEY (PK) is a column or combination of columns with a unique value for each row. Each PRIMARY KEY value must be unique within the table. It binds data together, across tables, without repeating all of the data in every table, and it ensures that no duplicate records are inserted into the table to which it is applied. It must always be NON NULL and will always have a [unique](#_Unique) index. A PRIMARY KEY is used for [joins](#_Join_Tables) with FOREIGN KEYS in other tables.

A FOREIGN KEY (FK) in one table points to a PRIMARY KEY in another table. A FOREIGN KEY column in one table is dependent to a PRIMARY KEY column in the second table. This constraint does not create an index automatically. It is a good practice to create an index on the column or columns specified in a FOREIGN KEY; otherwise the missing index can cause big performance problems for queries. Bellow you can observe how PRIMARY and FOREIGN KEYs appear in the database:



1. FK vs PK in AdventureWorksLT table
2. Create a table Product\_List containing the next four columns: Product\_ID, Name, Description, and Cost. Set Primay Key for the Product\_ID column:

CREATE TABLE Product\_List

(

Product\_ID INT NOT NULL Primary Key,

Name VARCHAR(255) NOT NULL,

Description VARCHAR(1000) NULL,

Cost VARCHAR(255) NULL

Or you may use:

CREATE TABLE Product\_List

(

Product\_ID INT NOT NULL ,

Name VARCHAR(255) NOT NULL,

Description VARCHAR(1000) NULL,

Cost VARCHAR(255) NULL

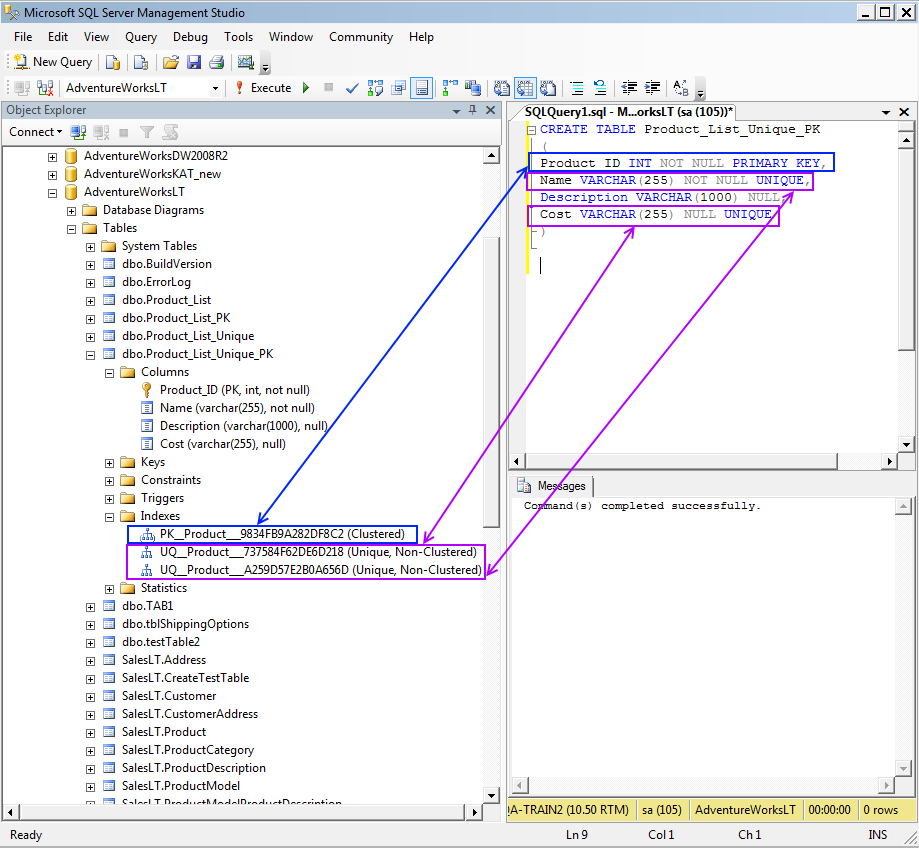
CONSTRAINT pk\_Product\_ID Primary KEY(Product\_ID)

)

### UNIQUE

The UNIQUE constraint is similar to the PRIMARY KEY in that it does not allow duplicate values to be entered into a column or group of columns in the table in which it’s been created. Some systems refer to this constraint as a “Candidate” constraint, because it can replace the PRIMARY KEY in function, should the need arise. Note that A PRIMARY KEY constraint automatically has a UNIQUE constraint defined on it.

As with the PRIMARY KEY constraint, an index will be created to match the columns specified in the constraint definition. You have the ability to supply index-specific information through either the [CREATE](#_Create_Table) or [ALTER](#_Alter_Table_1) TABLE statements.



1. PRIMARY KEYS vs UNIQUE on creating a table

The UNIQUE constraint helps to improve performance of SQL queries. For example, if most queries are invoked for Name and Cost, they are set to UNIQUE index (see the Image 2). The decision on applying UNIQUE constraint is usually made by a database administrator.

Unlike PRIMARY KEYS, UNIQUE supplies a NULL value for a column, though it is recommend not allowing NULLs because of the behavior of NULLs in sorting and output, and because a NULL value in a column would prevent the use of the UNIQUE constraint as a PRIMARY KEY.

### CHECK

The CHECK constraint is used to limit the value range that can be placed in a column.

If you define a CHECK constraint on a single column it allows only certain values for this column. If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row. When a table is [dropped](#_Drop_Table) from the database, the CHECK constraints are also dropped.

The SQL creates a CHECK constraint on the P\_Id column when the Product\_List table is created. The CHECK constraint specifies that the P\_Id column must only include integers greater than 0.

CREATE TABLE Product\_List

(

P\_Id int NOT NULL,

Name varchar(255) NOT NULL,

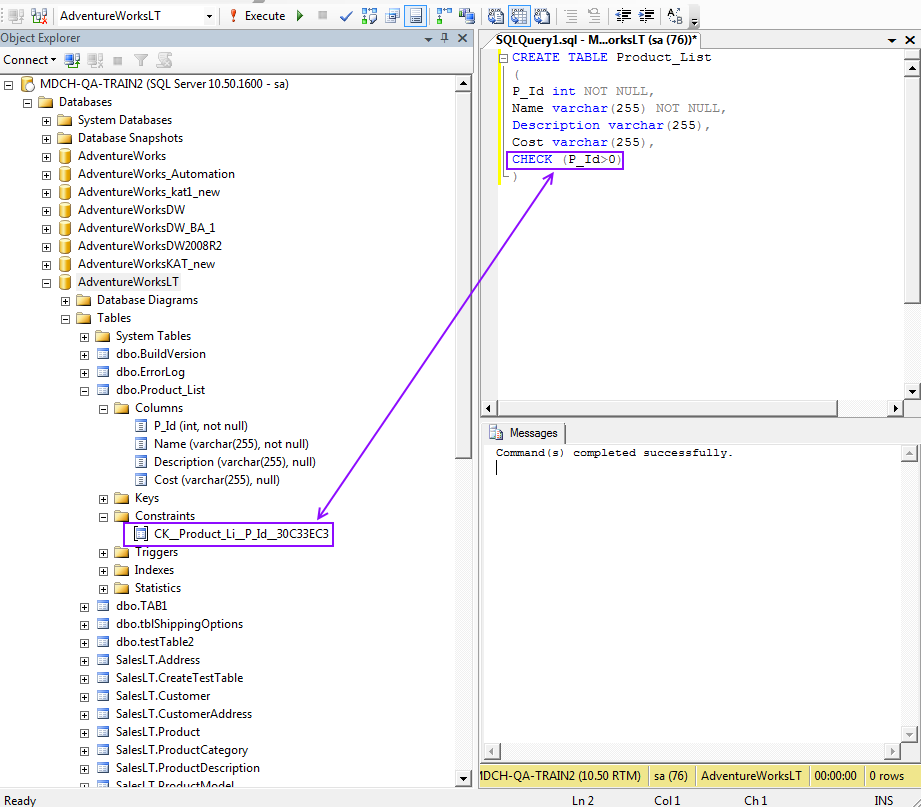
Description varchar(255),

Cost varchar(255),

CHECK (P\_Id>0)

)

The result will look like here:



1. CHECK on creating a table

### DEFAULT

The DEFAULT constraint is used to insert a default value into a column. You can create a constraint that in essence works just like the DEFAULT object except that it is applied to the table and column directly through the [CREATE](#_Create_Table) or [ALTER](#_Alter_Table_1) statements.

The following SQL creates a DEFAULT constraint on the City column when the Product\_List table is created:

CREATE TABLE Product\_List

(

Product\_ID INT NOT NULL,

Name VARCHAR(255) NOT NULL,

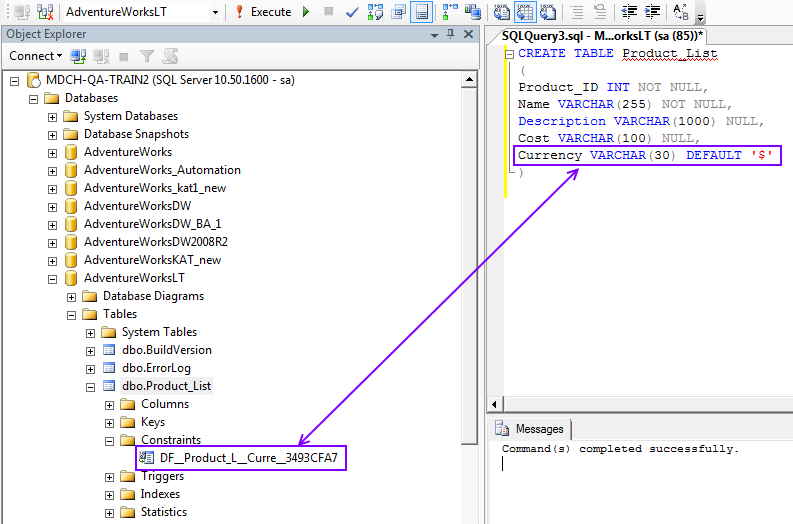
Description VARCHAR(1000) NULL,

Cost VARCHAR(100) NULL,

Currency VARCHAR(30) DEFAULT '$'

)

The table will contain the DEFAULT constraint as below:



1. DEFAULT on creating a table

# INSERT INTO…SELECT

With the INSERT INTO ... VALUES option, you insert only one row at a time into a table (please refer to Basic SQL training plan: *QA-BMS-24.02 - Basic MS SQL Commands*).

With the INSERT INTO ... SELECT option, you may insert many rows into a table at one time.

The general syntax for the INSERT INTO .. SELECT option is:

INSERT INTO target table (column1, column2, column3, . . .)

"SELECT clause"

For the following example, we will use the AdventureWorks database. Follow the next steps for practicing INSERT INTO ... SELECT query:

1. Select all records from the existing table:

SELECT \* from dbo.TCustomer

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| TCustomer\_Id | TCustomer\_Name | TCustomer\_Surname | TCustomer\_Job\_Title | Country\_ID |
| 1 | CN1 | CS1 | Tester | 1 |
| 2 | CN2 | CS2 | Tester | 3 |
| 3 | CN3 | CS3 | Developer | 1 |
| 4 | CN4 | CS4 | Tester | 2 |
| 5 | CN5 | CS5 | Developer | 2 |
| 8 | CN8 | CS8 | Developer | 1 |
| 9 | CN9 | CS9 | Tester | 1 |
| 10 | CN10 | CS10 | Tester | 2 |
| 11 | CN11 | CS11 | Inginer | 2 |
| 12 | CN12 | CS12 | Inginer | 3 |
| 13 | CN13 | CS13 | Developer | 3 |

1. Create new test table with one field:

IF OBJECT\_ID('TestInsert') IS NOT NULL

DROP TABLE TestInsert

CREATE TABLE TestInsert

(jobTitle VARCHAR (50))

1. Select all records from the newly created table. Actual result is nullData set

SELECT \* FROM TestInsert

1. Populate table TestInsert with values from Tcustomer table:

INSERT INTO TestInsert (jobTitle)

SELECT c.TCustomer\_Job\_Title FROM TCustomer c

WHERE c.TCustomer\_Job\_Title = 'Tester'

1. Extract all records from TestInsert table:

SELECT \* FROM TestInsert

The result-set is:

|  |
| --- |
| jobTitle |
| Tester |
| Tester |
| Tester |
| Tester |
| Tester |

|  |
| --- |
| **NOTE:**  The **INSERT INTO .. SELECT** will succeed if the data types of the SELECT match the data types of the columns in the table you are inserting. |

# Alter Table

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.

1. To **add a column** **in a table**, use the following query:

ALTER TABLE dbo.Product\_List

ADD Model VARCHAR(150) NULL

1. To **delete a column in a table**, apply to DROP statement as follows:

ALTER TABLE dbo.Product\_List

DROP COLUMN Model

1. To **change the data type of a column in a table**, for example, modify the Cost column’s data type from VARCHAR into FLOAT, use the following query:

ALTER TABLE dbo.Product\_List

ALTER COLUMN Cost float

# Drop Table

Using the SQL DROP command, you can easily delete columns, indexes, tables, and databases. Note, to remove a column, DROP command is used within [ALTERT](#_Alter_Table_1) statement (see in Example 2 above).

|  |
| --- |
| **IMPORTANT:** Once a column or table has been dropped, the data stored inside of it will be lost forever. **Proceed with caution!** |

The DROP TABLE statement is used to delete a table:

|  |
| --- |
| **NOTE:**  If using the QA MS SQL database for testing scope, please, make sure you have restored the modified data by removing the newly created columns or tables. |

DROP TABLE dbo.Product\_List

***Reference:*** *Removing data from a table, without removing a table, via TRUNCATE TABLE was discussed in QA-BMS-24.02-01.02 - Basic MS SQL Commands.*

# Join Tables

The most frequently used commands during database testing is manipulating with SQL joins. Once there is a need to use data for more than one table, you can apply to the next types of joins:

* **Cross Join** – all rows in the first (left) table are joined to all rows of the (right) second table without any conditions.
* **Inner Join** – all rows from both tables (left and right) are matched.
* **Left (Outer) Join** – the rows from first (left) table are returned even if not matched.
* **Right (Outer) Join** - the rows from second (right) table are returned even if not matched.
* **Full (Outer) Join** – all rows in the first (left) table are joined to all rows of the (right) second table with some conditions.
* **Self-Join** – records from one or several columns are matched in the same table.

Tables in a database are related to each other with keys (see more on primary and foreign keys in [Create Table](#_Create_Table)).

Most queries using a join can be rewritten using a sub-query (a query nested within another query), and most sub-queries can be rewritten as joins. (For more information, see [Sub-Queries](#_Sub-queries).)

Before proceeding with JOIN conditions, it is good to know that there are NULL values, which are unknown missing data in the table. (Find more about different treatment of NULLs in the [Working with NULLs](#_Working_with_NULLs) chapter.)

## Cross Join

The CROSS JOIN (a.k.a. Cartesian product) is formed when join condition is omitted and all combinations of rows are displayed. The CROSS JOIN generates a large numbers of rows.

|  |
| --- |
| **IMPORTANT:**  You should always include a valid join condition in a WHERE clause, unless you want to combine all rows from all tables on purpose. This type of join might be useful for specific performance tests to simulate a reasonable amount of data. |

The example below returns customer’s First Name and Product Name from SalesLT.Customer and SalesLT.Product tables. Because no WHERE clause specified, the query generated 249865 results by combining 295 rows from SalesLT.Product and 847 rows from SalesLT.Customer tables.

select FirstName, Name as ProductName

from SalesLT.Customer, SalesLT.Product

|  |  |
| --- | --- |
| FirstName | ProductName |
| A. | All-Purpose Bike Stand |
| A. | All-Purpose Bike Stand |
| Abigail | All-Purpose Bike Stand |
| Abigail | All-Purpose Bike Stand |
| Abraham | All-Purpose Bike Stand |

*…*

*249865 rows selected.*

## INNER JOIN

The INNER JOIN (a.k.a. natural join) or, simply, JOIN keyword returns rows when there is at least one match in both tables. If there are no matches in tables, the INNER JOIN skips those data. This type of join, usually, involves primary and foreign key compliments.

1. Display the full name and address type of all the customers with the first name as David and origin from USA.

|  |
| --- |
| **TIP:**  You can use at the beginning decision matrix for simplifying building joins. You can start with the next decision tree:  **Columns to Display Table Condition**  FirstName Customer WHERE c.FirstName = 'david'  LastName CustomerAddress a.CountryRegion = 'United States'  CountryRegion Address |

Bearing in mind the tip above, you can create a query using INNER JOIN.

select c.FirstName, c.LastName, a.CountryRegion

from SalesLT.Customer as c

inner join SalesLT.CustomerAddress as ca

on c.CustomerID = ca.CustomerID

inner join SalesLT.Address as a

on ca.AddressID = a.AddressID

where c.FirstName = 'david'

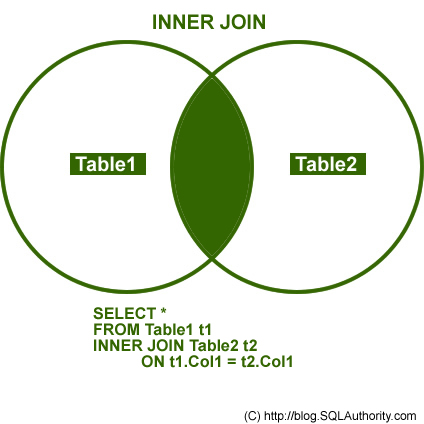
and a.CountryRegion = 'United States'

The results are:

|  |  |  |
| --- | --- | --- |
| FirstName | LastName | Country/Region |
| David | Hodgson | United States |
| David | Johnson | United States |
| David | Liu | United States |

Once skipping the WHERE condition, 416 rows without any NULL values will be returned.

You may consider the next graphical presentation of the INNER JOIN data sets using Venn diagrams:



|  |
| --- |
| **NOTE:** Using „JOIN” will have same behavior as „INNER JOIN” |

## OUTER JOIN

Once INNER JOINS eliminate the rows that do not match with a row from the other table, OUTER JOINS, however, return all rows from at least one of the tables, as long as those rows meet any WHERE or HAVING search conditions. If you will use OUTER JOIN in the join condition, the NULL values will be returned.

All rows are retrieved from the left table referenced with a left outer join, and all rows from the right table referenced in a right outer join. All rows from both tables are returned in a full outer join.

You can use the following keywords for OUTER JOINS specified in a FROM clause:

* LEFT OUTER JOIN or LEFT JOIN
* RIGHT OUTER JOIN or RIGHT JOIN
* FULL OUTER JOIN or FULL JOIN

### LEFT OUTER JOIN

If you need to look for all customers regardless there is office type specified for them or not, we use LEFT JOIN. The LEFT JOIN keyword returns all rows from the left table (Customer), even if there are no matches in the right table (CustomerAddress).

select c.FirstName, c.LastName, ca.AddressType

from SalesLT.Customer as c

left join SalesLT.CustomerAddress as ca

on c.CustomerID = ca.CustomerID

The LEFT JOIN result-set will return NULLs from right table:

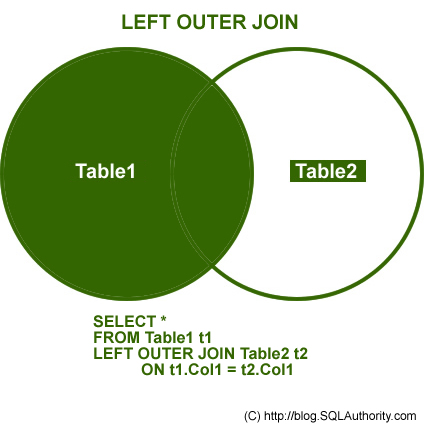
|  |  |  |
| --- | --- | --- |
| FirstName | LastName | AddressType |
| Orlando | Gee | NULL |
| Keith | Harris | NULL |
| Donna | Carreras | NULL |
| Janet | Gates | NULL |
| Lucy | Harrington | NULL |
| Rosmarie | Carroll | NULL |
| Dominic | Gash | NULL |
| Kathleen | Garza | NULL |

*…*

*857 rows selected, including 441 unmatched.*

SQL server returns all the rows from the left table (Customer), even if there are no matches in the right table (CustomerAddress).

Graphically, the LEFT OUTER JOIN relationship can be illustrated as follows:



### RIGHT OUTER JOIN

The RIGHT JOIN keyword returns all the rows from the right table (Address), even if there are no matches in the left table (CustomerAddress). The query bellow retrieves all rows in the CustomerAddress table showing all addresses disregard the address type is specified or not:

select ca.AddressType, a.CountryRegion, a.City, a.AddressLine1

from SalesLT.CustomerAddress as ca

right join SalesLT.Address as a

on ca.AddressID = a.AddressID

order by ca.AddressType

The RIGHT JOIN result-set will return NULLs from left table:

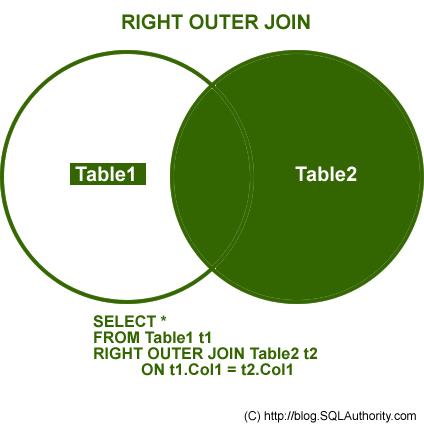
|  |  |  |  |
| --- | --- | --- | --- |
| AddressType | CountryRegion | City | AddressLine1 |
| NULL | Canada | Vancouver | 5250-505 Burning St |
| NULL | Canada | Surrey | 252345 8810th Avenue |
| NULL | Canada | Toronto | 6th Floor Ferguson Block |
| NULL | Canada | Scarborough | 2550 Middlefield Road |
| NULL | Canada | Winnipeg | 6th Floor 5250 Main Street |
| NULL | Canada | Surrey | 255117 - 101 Avenue |
| NULL | Canada | Ottawa | #500-75 O'Connor Street |
| NULL | United States | Wood Dale | 25269 N. Wood Dale Rd |
| NULL | United States | Stafford | Fountains On The Lake |
| NULL | United States | Hillsboro | Southwest Outlet |
| NULL | United States | San Antonio | Fiesta Trail Shopping Center |
| NULL | United Kingdom | London | 67 Vincent Square, Victoria |
| NULL | United Kingdom | York | 7 Pioneer Business Park |

*…*

*450 rows selected, including 33 unmatched records.*

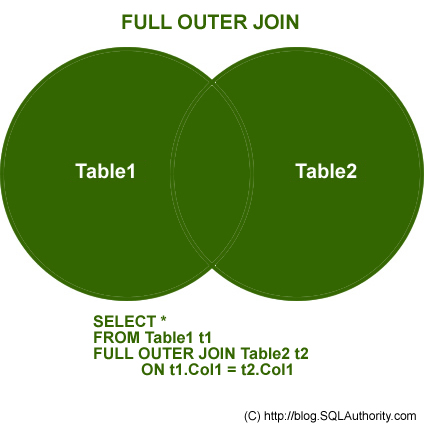
SQL server returns all the rows from the right table (Address), even if there are no matches in the left table (CustomerAddress).

Here it is a graphical representation of the RIGH JOIN using Venn diagrams:



### FULL OUTER JOIN

The FULL (OUTER) JOIN keyword returns rows when there is a match in one of the tables. To list all the (unmatched) records from left and right tables, use the next SQL query:



|  |
| --- |
| **NOTE:** „FULL OUTER JOIN” will have same behavior as „FULL JOIN” |

## Self-Join

Sometimes you need to join table to itself. Use a SELF-JOIN when you want to create a result set that joins records in a table with other records in the same table. To list a table two times in the same query, you must provide a table alias.

Assume you need to find all employees’ managers within the Employees table.

**EMPLOYEES (WORKERS) EMPLOYEES (MANAGERS)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| EmployeeID | FullName | ManagerID |  | EmployeeID | FullName |
| 01 | Jack Smith | 03 | 01 | Jack Smith |
| 02 | Iris Molda |  | 02 | Iris Molda |
| 03 | Leo Somerset |  | 03 | Leo Somerset |
| 04 | Mary Brown | 03 | 04 | Mary Brown |
| 05 | Stephan Great | 02 | 05 | Stephan Great |

The example below joins the Employees table to itself. To simulate two tables in the FROM clause, there are two aliases used, namely workers (w) and manager (m).

select w.FullName as EmployeeName,

'works for',

m.FullName as ManagerName

from Employees w

join Employees m

on w.ManagerID = m.EmployeeID

The result-set will be:

|  |  |  |
| --- | --- | --- |
| EmployeeName |  | ManagerName |
| Mary Brown | works for | Leo Somerset |
| Jack Smith | works for | Leo Somerset |
| Stephan Great | works for | Iris Molda |

In the **PersonsDuplicates**  table below, refer that for same PersonID there are multiple dates and a financial column “Sales”.

|  |  |  |
| --- | --- | --- |
| PersonID | Sales | uDate |
| 1 | 655 | 11.01.2012 07:45 |
| 1 | 658 | 10.01.2012 07:45 |
| 1 | 7888 | 08.01.2012 07:45 |
| 1 | 5667 | 04.01.2012 07:45 |
| 2 | 6568 | 10.01.2012 07:45 |
| 3 | 6582 | 10.01.2012 07:45 |
| 3 | 6581 | 14.01.2012 07:45 |

We need to extract only last updated value for sales and uDate for all PersonIDs:

Please refer a query with SELF JOIN in order to return needed results:

Select

      tab1.PersonID,

      tab1.Sales,

      tab1.uDate

from (

Select

      PersonID ,

      max(udate)as udate  -- get last date for each PersonID

from PersonsDuplicates

Group by PersonID

       ) tab

Join

            (

            Select

                  PersonID ,

                  udate ,

                  Sales

            from PersonsDuplicates

            ) tab1

on  tab.PersonID=tab1.PersonID

   and tab.udate=tab1.uDate

Order by 1

The output is:

|  |  |  |
| --- | --- | --- |
| PersonID | Sales | uDate |
| 1 | 655 | 11.01.2012 07:45 |
| 2 | 6568 | 10.01.2012 07:45 |
| 3 | 6581 | 14.01.2012 07:45 |

## Joining More Tables

Although each join specification joins only two tables, FROM clauses can contain multiple join specifications. This allows many tables to be joined for a single query. An example you might have noticed in the section [Inner Join](#_Inner_Join) when joining three tables: Customer, CustomerAddress and Address.

In the following SQL query, to find the names of all customers, their items purchased and some sale details in one, you will need to use a multiple table join by using four tables Customer, SalesOrderHeader, SalesOrderDetail and Product:

select c.LastName, c.FirstName,

p.Name as ProductName,

oh.SalesOrderNumber,

od.OrderQty, od.UnitPrice, od.LineTotal

from SalesLT.Customer as c

join SalesLT.SalesOrderHeader as oh

on c.CustomerID = oh.CustomerID

join SalesLT.SalesOrderDetail as od

on od.SalesOrderID = oh.SalesOrderID

join SalesLT.Product as p

on p.ProductID = od.ProductID

order by c.LastName

The result-set will look as here:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| LastName | FirstName | ProductName | salesOrderNumber | OrderQty | UnitPrice | LineTotal |
| Abel | Catherine | Long-Sleeve Logo Jersey, M | SO71782 | 3 | 29,994 | 89.982.000 |
| Abel | Catherine | Touring-1000 Yellow, 54 | SO71782 | 3 | 1430,442 | 4.291.326.000 |
| Abel | Catherine | Touring-1000 Yellow, 46 | SO71782 | 1 | 1430,442 | 1.430.442.000 |
| Abel | Catherine | AWC Logo Cap | SO71782 | 10 | 5,394 | 53.940.000 |

*…*

*542 rows selected matching records from four tables**.*

# Set Operations

When you need to combine the results from two or more SELECT statements, SQL handles these requirements by using set operations. MS SQL supports the following three main set operations:

* UNION
* UNION ALL
* EXCEPT
* INTERSECT

Here it is an overview of how each operator can be use to return different combination of data:

* To return the data in Set **A** that doesn’t overlap with **B**, use **A** EXCEPT **B**.
* To return only the data that overlaps in the two sets, use **A** INTERSECT **B**.
* To return the data in Set **B** that doesn’t overlap with **A**, use **B** EXCEPT **A**.
* To return the data in all three areas without duplicates, use **A** UNION **B**.
* To return the data in all three areas, including duplicates, use **A** UNION ALL **B**.
* To return the data in the non-overlapping areas of both sets, use (**A** UNION **B**) except (**A** INTERSECT **B**), or perhaps (**A** EXCEPT **B**) UNION (**B** EXCEPT **A**).

## UNION

The UNION operator is used to combine the result-set of two or more SELECT statements. Notice that each SELECT statement within the UNION must have the same number of columns. The columns must also have similar data types. Also, the columns in each SELECT statement must be in the same order.

1. Union the tables [Production].[ProductModel] and [Gloves].

SELECT ProductModelID, Name

FROM [AdventureWorks].[Production].[ProductModel]

UNION

SELECT ProductModelID, Name

FROM dbo.Gloves

ORDER BY ProductModelID

The results are:

|  |  |
| --- | --- |
| ProductModelID | Name |
| 1 | Classic Vest |
| 2 | Cycling Cap |
| 3 | Full-Finger Gloves |
| 4 | Half-Finger Gloves |
| 5 | HL Mountain Frame |
| 6 | HL Road Frame |
| 7 | HL Touring Frame |
| 8 | LL Mountain Frame |

*...*

*128 rows selected*

1. Union the the tables [Production].[ProductModel] and [Gloves]where ProductModelID is not equal with 1 and 2.

|  |
| --- |
| **NOTE:**  In order to specify discrete values in your SQL WHERE search criteria, apply to IN and NOT IN clauses.   1. Select the persons with a product name not equal to "Chain" or "Taillight" from the [AdventureWorks].[Production].[ProductModel]:   SELECT ProductModelID, Name  FROM [AdventureWorks].[Production].[ProductModel]  WHERE ProductModelID NOT IN (3,4)   1. Select the persons with a product name equal to "Chain" or "Taillight" from the [AdventureWorks].[Production].[ProductModel]:   SELECT \* FROM [AdventureWorks].[Production].[ProductModel]  WHERE Name IN ('Chain','Taillight')  (See more examples in *QA-BMS-24.02 - Basic MS SQL Commands*.) |

We use the following SELECT statement:

SELECT ProductModelID,Name

INTO dbo.ProductResult

FROM [AdventureWorks].[Production].[ProductModel]

WHERE ProductModelID NOT IN (1, 2)

UNION

SELECT ProductModelID, Name

FROM dbo.Gloves;

The results are:

|  |  |
| --- | --- |
| ProductModelID | Name |
| 3 | **Full-Finger Gloves** |
| 4 | **Half-Finger Gloves** |
| 5 | HL Mountain Frame |
| 6 | HL Road Frame |
| 7 | HL Touring Frame |

*...*

*126 rows selected*

## UNION ALL

The UNION ALL query allows you to combine the result sets of 2 or more "select" queries. It returns all rows (even if the row exists in more than one of the "select" statements). Each SQL statement within the UNION ALL query must have the same number of fields in the result sets with similar data types. The difference between **UNION ALL** and **UNION** is that, while **UNION** only selects distinct values, **UNION ALL** selects all values.

1. Union the the tables [Production].[ProductModel] and [Gloves].

We use the following SELECT statement:

SELECT ProductModelID, Name

FROM [AdventureWorks].[Production].[ProductModel]

UNION ALL

SELECT ProductModelID, Name

FROM dbo.Gloves

ORDER BY ProductModelID

The results are:

|  |  |
| --- | --- |
| ProductModelID | Name |
| 1 | Classic Vest |
| 2 | Cycling Cap |
| 3 | **Full-Finger Gloves** |
| 3 | **Full-Finger Gloves** |
| 4 | **Half-Finger Gloves** |
| 4 | **Half-Finger Gloves** |
| 5 | HL Mountain Frame |
| 6 | HL Road Frame |

*...*

*130 rows selected*

1. Union the tables [Production].[ProductModel] and [Gloves]and [Product Result]

We use the following SELECT statement:

SELECT ProductModelID, Name

FROM [AdventureWorks].[Production].[ProductModel]

UNION ALL

(SELECT ProductModelID, Name

FROM dbo.Gloves

UNION

SELECT ProductModelID, Name

FROM dbo.ProductResult

)

The results are:

|  |  |
| --- | --- |
| ProductModelID | Name |
| 122 | All-Purpose Bike Stand |
| 119 | Bike Wash |
| 115 | Cable Lock |
| 98 | Chain |

*…*

*256 rows selected*

1. This example shows how UNION ALL can be used together with the HAVING clause.

Table [AdventureWorksLT2008R2].[SalesLT].Address was migrated by an ETL Job to [AdventureWorksLT].[SalesLT].Address.

Let’s write SQL script that checks that the next columns were migrated properly:

[AddressID]

,[AddressLine1]

,[AddressLine2]

,[City]

,[StateProvince]

,[CountryRegion]

,[PostalCode]

,[ModifiedDate]

**1.** First of all, the row count must be verified for both tables:

select COUNT (\*)as "target count" from [AdventureWorksLT2008R2].[SalesLT].Address

select COUNT (\*) as "source count" from [AdventureWorksLT].[SalesLT].Address

|  |
| --- |
| Target Count |
| 450 |
| Source Count |
| 533 |

**2.** To find differences in values for these two tables from different databases, we’ll use UNION ALL, GROUP BY and HAVING COUNT operators:

select max(db)as db,

[AddressID]

,[AddressLine1]

,[AddressLine2]

,[City]

,[StateProvince]

,[CountryRegion]

,[PostalCode]

,[ModifiedDate]

from

(

select 'source' as db,

[AddressID]

,[AddressLine1]

,[AddressLine2]

,[City]

,[StateProvince]

,[CountryRegion]

,[PostalCode]

,[ModifiedDate]

from [AdventureWorksLT2008R2].[SalesLT].Address g

union all

select 'target' as db,

[AddressID]

,[AddressLine1]

,[AddressLine2]

,[City]

,[StateProvince]

,[CountryRegion]

,[PostalCode]

,[ModifiedDate]

from [AdventureWorksLT].[SalesLT].Address t

) m

group by

[AddressID]

,[AddressLine1]

,[AddressLine2]

,[City]

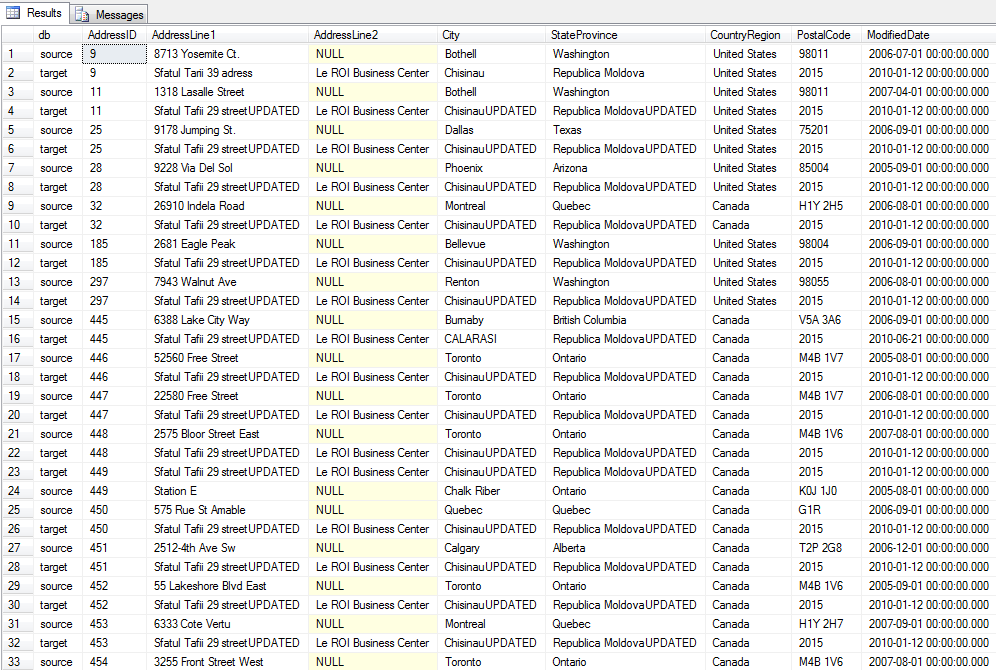
,[StateProvince]

,[CountryRegion]

,[PostalCode]

,[ModifiedDate]

having COUNT(\*) <>2



## EXCEPT

EXCEPT returns any distinct values from the left query that are not also found on the right query.The basic rules for combining the result sets of two queries that use EXCEPT or INTERSECT are the following:

* The number and the order of the columns must be the same in all queries.
* The data types must be compatible.

1. The following query returns any distinct values from the query to the left of the EXCEPT operand that are not also found on the right query. Use the table: [AdventureWorks].[Production].[Product] and [AdventureWorks].[Production].[WorkOrder].

We use the following SELECT statement:

SELECT ProductID

FROM [AdventureWorks].[Production].[Product]

EXCEPT

SELECT ProductID

FROM [AdventureWorks].[Production].[WorkOrder]

The results are:

|  |
| --- |
| ProductID |
| 365 |
| 404 |
| 474 |
| 884 |
| 347 |
| 396 |
| 848 |
| 438 |
| 715 |
| 492 |
| 679 |
| 912 |
| 387 |
| 435 |

*...*

*266 rows selected (products without work orders)*

1. The following query returns any distinct values from the query to the left of the EXCEPT operand that are not also found on the right query. The tables are reversed from the previous example.

We use the following SELECT statement:

SELECT ProductID

FROM [AdventureWorks].[Production].[WorkOrder]

EXCEPT

SELECT ProductID

FROM [AdventureWorks].[Production].[Product]

*Result: 0 Rows (work orders without products)*

## INTERSECT

INTERSECT returns any distinct values that are returned by both the query on the left and right sides of the INTERSECT operand.

1. Union the tables [Production].[ProductModel] and [Gloves]and [Product Result]

The following query returns any distinct values that are returned by both the query on the left and right sides of the INTERSECT operand.

We use the following SELECT statement:

SELECT ProductID

FROM [AdventureWorks].[Production].[Product]

INTERSECT

SELECT ProductID

FROM [AdventureWorks].[Production].[WorkOrder]

The results are:

|  |
| --- |
| ProductID |
| 3 |
| 316 |
| 324 |
| 327 |
| 328 |
| 329 |
| 330 |
| 331 |
| 350 |
| 398 |
| 399 |
| 400 |
| 401 |
| 514 |

*...*

*238 rows selected (products that have work orders)*

# Sub-Queries

Sub-query or Inner query or Nested query is a query in a query. A sub-query is usually added in the WHERE Clause of the SQL statement. Most of the time, a sub-query is used when you know how to search for a value using a SELECT statement, but do not know the exact value. Sub-queries are an alternate way of returning data from multiple tables.

1. Select the Name and the Product Model ID of the product that is present in [Production].[Product] table and where color is black.

We use the following SELECT statement:

SELECT ProductModelID, Name

FROM [AdventureWorks].[Production].[ProductModel]

WHERE ProductModelID IN (SELECT ProductModelID

FROM [AdventureWorks].[Production].[Product]

WHERE color= 'black'

)

The results are:

|  |  |
| --- | --- |
| ProductModelID | Name |
| 3 | Full-Finger Gloves |
| 4 | Half-Finger Gloves |
| 101 | HL Crankset |
| 5 | HL Mountain Frame |
| 46 | HL Mountain Front Wheel |
| 125 | HL Mountain Rear Wheel |
| 6 | HL Road Frame |
| 51 | HL Road Front Wheel |
| 78 | HL Road Rear Wheel |

*...*

*36 rows selected*

1. Select theSalesOrderID where the sum of UnitPrice is more than 25000.00.

We use the following SELECT statement:

SELECT soh.SalesOrderID

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

WHERE soh.SalesOrderID IN

(SELECT sod.SalesOrderID

FROM [AdventureWorks].[Sales].[SalesOrderDetail] AS sod

GROUP BY sod.SalesOrderID

HAVING sum(UnitPrice)> 25000.00)

The same query with exists:

SELECT soh.SalesOrderID

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

WHERE exists

(SELECT sod.SalesOrderID

FROM [adventureworks].[sales].[salesorderdetail] AS sod

WHERE sod.SalesOrderID = soh.SalesOrderID

GROUP BY sod.SalesOrderID

HAVING SUM(UnitPrice)> 25000.00

)

The results are:

|  |
| --- |
| SalesOrderID |
| 46959 |
| 47455 |
| 46666 |

SELECT soh.SalesOrderID,

soh.SalesOrderNumber

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

join (SELECT sod.SalesOrderID,

sod.UnitPrice,

sum(UnitPrice)AS TotalNR

FROM [AdventureWorks].[Sales].[SalesOrderDetail] AS sod

GROUP BY sod.SalesOrderID,sod.UnitPrice

HAVING sum(UnitPrice) > 11000.00

) AS il

ON soh.SalesOrderID=il.SalesOrderID

The results are:

|  |  |
| --- | --- |
| SalesOrderID | SalesOrderNumber |
| 59044 | SO59044 |
| 65201 | SO65201 |
| 71918 | SO71918 |
| 61235 | SO61235 |
| 63284 | SO63284 |
| 65191 | SO65191 |
| 71835 | SO71835 |
| 59008 | SO59008 |
| 69406 | SO69406 |
| 71894 | SO71894 |
| 57150 | SO57150 |
| 65257 | SO65257 |
| 58961 | SO58961 |

*...*

*24 rows selected*

# Derived Structures

## Views

A SQL VIEW is a virtual table, which is based on SQL SELECT query. Essentially a view is very close to a real database table (it has columns and rows just like a regular table), except for the fact that the real tables store data, while the views don’t. The view’s data is generated dynamically when the view is referenced. A view references one or more existing database tables or other views. In effect every view is a filter of the table data referenced in it and this filter can restrict both the columns and the rows of the referenced tables.

Views offer the following advantages:

* **Ease of use**: A view hides the complexity of the database tables from end users. Essentially we can think of views as a layer of abstraction on top of the database tables.
* **Space savings**: Views takes very little space to store, since they do not store actual data.
* **Additional data security**: Views can include only certain columns in the table so that only the non-sensitive columns are included and exposed to the end user. In addition, some databases allow views to have different security settings, thus hiding sensitive data from prying eyes.

CREATE VIEW owner. view\_name [ **(**column [ **,**n ] **)** ]

select\_statement

[ WITH CHECK OPTION ]

### With Check Option

If a view has been defined with the WITH CHECK OPTION, any modification to the underlying data can result in that data no longer being accessible through the view. This force all data modification statements executed against the view to adhere to the criteria set within select\_statement. When a row is modified through a view, the WITH CHECK OPTION guarantees that the data remains visible through the view after the modification has been committed.

|  |
| --- |
| * **IMPORTANT :** * Create view statement cannot contain the INTO keyword or the COMPUTE/COMPUTE BY clauses. * ORDER BY clause can only be specified in conjunction with the TOP keyword. * Views cannot be combined in Batch T-SQL statements . * The maximum number of columns a view can reference is 1024. * A view cannot reference any temporary tables . * Views can only be created in the current database. |

|  |
| --- |
|  |

### Create Views

Consider the example bellow to get more familiar with View creation.

You can create a view by selecting the ProductID, Name, ListPrice columns:

CREATE VIEW TEST AS SELECT ProductID,Name,ListPrice

FROM [AdventureWorks].[Production].[Product]

WHERE ListPrice > 2000.00

The results are:

|  |  |  |
| --- | --- | --- |
| ProductID | Name | ListPrice |
| 749 | Road-150 Red, 62 | 3578,27 |
| 750 | Road-150 Red, 44 | 3578,27 |
| 751 | Road-150 Red, 48 | 3578,27 |
| 752 | Road-150 Red, 52 | 3578,27 |
| 753 | Road-150 Red, 56 | 3578,27 |
| 771 | Mountain-100 Silver, 38 | 3399,99 |
| 772 | Mountain-100 Silver, 42 | 3399,99 |
| 773 | Mountain-100 Silver, 44 | 3399,99 |
| 774 | Mountain-100 Silver, 48 | 3399,99 |

*...*

*35 rows selected*

1. We use the following statement:

CREATE VIEW SalesOrderview

AS

SELECT ProductID,UnitPrice

FROM [AdventureWorks\_kat1\_new].[Sales].[SalesOrderDetail] AS so

JOIN [AdventureWorks\_kat1\_new].[Sales].[SalesOrderHeader] AS soh ON (so.SalesOrderID = soh.SalesOrderID)

The results are:

|  |  |
| --- | --- |
| ProductID | UnitPrice |
| 776 | 2024,994 |
| 777 | 2024,994 |
| 778 | 2024,994 |
| 771 | 2039,994 |
| 772 | 2039,994 |
| 773 | 2039,994 |
| 774 | 2039,994 |
| 714 | 28,8404 |
| 716 | 28,8404 |

1. We use the following statement:

CREATE VIEW locationsview AS

SELECT d.Department\_ID, d.Name, l.Location\_ID, l.Regional\_Group

FROM [For Test].[dbo].[Department] d, [For Test].[dbo].[Location] l

WHERE d.location\_id = l.location\_id;

The results are:

|  |  |  |  |
| --- | --- | --- | --- |
| Department\_ID | Name | Location\_ID | Regional\_Group |
| 1 | Sales | 1 | France |
| 2 | Operation | 3 | Moldova |
| 3 | Accounting | 2 | German |

1. If the view already exists, drop it and create another view as in the following statement:

IF EXISTS (SELECT TABLE\_NAME FROM INFORMATION\_SCHEMA.VIEWS

WHERE TABLE\_NAME = 'titles\_view')

DROP VIEW titles\_view

GO

CREATE VIEW titles\_view

AS

SELECT ProductID,UnitPrice

FROM [AdventureWorks\_kat1\_new].[Sales].[SalesOrderDetail] AS so

JOIN [AdventureWorks\_kat1\_new].[Sales].[SalesOrderHeader] AS soh ON (so.SalesOrderID = soh.SalesOrderID)

|  |  |
| --- | --- |
| ProductID | UnitPrice |
| 776 | 2024,994 |
| 777 | 2024,994 |
| 778 | 2024,994 |
| 771 | 2039,994 |
| ProductID | UnitPrice |

For more VIEW examples, scroll down to [Create View Examples](#_APPENDIX:_Create_View) appendix.

### Update Views

Modify a previously created view. This includes an indexed view. ALTER VIEW does not affect dependent stored procedures or triggers and does not change permissions. If you want to include into the view just two columns, ProductID and Name, use the ALTER VIEW statement:

ALTER VIEW TEST AS SELECT ProductID,Name

FROM [AdventureWorks].[Production].[Product]

The results are:

|  |  |
| --- | --- |
| ProductID | Name |
| 1 | Adjustable Race |
| 879 | All-Purpose Bike Stand |
| 712 | AWC Logo Cap |
| 3 | BB Ball Bearing |
| 2 | Bearing Ball |
| 877 | Bike Wash - Dissolver |
| 316 | Blade |
| 843 | Cable Lock |
| 952 | Chain |

*...*

*35 rows selected*

### Delete Views

You can delete a view with the DROP VIEW command:

DROP VIEW TEST

## Inline Views

An inline view is a statement in the FROM-clause of another SELECT statement. In-line views are commonly used simplify complex queries by removing join operations and condensing several separate queries into a single query. An example of inline view is given below. An inline view is basically a query inside another query, which makes it a sub-query. However, an inline view is a sub query with a twist. It only exists inside of the FROM clause as a run-time result set.

As explained in [the description of a basic query](http://michaelmclaughlin.info/db1/lesson-5-querying-data/single-table-query/), the FROM clause is resolved first in a query. Tables in a FROM clause are resolved through parsing the query, which means looking the table name up in the data dictionary. An inline view is resolved slightly differently. The parsing process takes any nested query found in the FROM clause and parses them separately, like a sub-process. If it finds that all is in order in the nested query, it marks it to be run before any join activity. An example of an inline view is shown below.

select \* from

(select

c.AccountNumber,

c.CustomerType

from

[AdventureWorks].[Sales].[Customer] as c

join (select

a.AddressID,

a.AddressTypeID,

t.Name,

a.CustomerID

from

[AdventureWorks].[Sales].[CustomerAddress] as a

join [AdventureWorks].[Person].[AddressType] as t on a.AddressTypeID = t.AddressTypeID) as il

on il.CustomerID=c.CustomerID) as beta

The results are:

|  |  |
| --- | --- |
| AccountNumber | CustomerType |
| AW00000001 | S |
| AW00000002 | S |
| AW00000002 | S |
| AW00000003 | S |
| AW00000004 | S |
| AW00000004 | S |
| AW00000005 | S |
| AW00000006 | S |
| AW00000007 | S |
| AW00000008 | S |
| AW00000009 | S |
| AW00000010 | S |
| AW00000011 | S |
| AW00000012 | S |

*...*

*19220 rows selected*

## 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. For more information, see [*Recursive Queries Using Common Table Expressions*](http://msdn.microsoft.com/en-us/library/ms186243.aspx).
* Substitute for a view when the general use of a view is not required; that is, you do not have to store the definition in metadata.
* Enable grouping by a column that is derived from a scalar subselect, or a function that is either not deterministic or has external access.
* Reference the resulting table multiple times in the same statement.

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:**

-- 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;

The results are:

|  |  |
| --- | --- |
| SalesPersonID | TotalSales |
| 268 | 4 |
| 268 | 20 |
| 268 | 14 |
| 268 | 10 |
| 275 | 56 |
| 275 | 139 |
| 275 | 169 |
| 275 | 86 |
| 276 | 43 |
| 276 | 127 |
| 276 | 166 |
| 276 | 82 |
| 277 | 55 |
| 277 | 140 |
| 277 | 181 |

*...*

*58 rows selected*

# Programming Logic

## IF…ELSE

SQL IF...ELSE Statement used to test a condition. IF...ELSE statement using in execution of a Transact-SQL statement (Store Procedure or T-SQL) and Trigger. IF tests can be nested after another IF or following an ELSE. There is no limit to the number of nested levels.

* **IF** condition is satisfied and the Boolean expression returns **TRUE**, it will executed **IF** Block SQL statement.
* **IF** condition is not satisfied and the Boolean expression returns **FALSE**, it will executed **ELSE** Block SQL Statement. query.

Syntax for IF...ELSE:

IF ( Boolean\_expression )   
BEGIN  
SQL Statement Block  
END  
ELSE   
BEGIN  
SQL Statement Block  
END

1. Below is simple example of IF...ELSE Statement with one IF... ELSE BlockFor Boolean\_expression part, you can replace with your condition to match with your SQL query. You can also use EXISTS in the Condition to check the existence of a SQL Statement or Row.

IF

(SELECT COUNT(\*) FROM [AdventureWorks].[Production].[Product] WHERE Name LIKE 'Touring-3000%' ) > 5

PRINT 'There are more than 5 Touring-3000 bicycles.'

ELSE PRINT 'There are 5 or less Touring-3000 bicycles.'

*The result is more than 5 Touring-3000 bicycles.*

1. Select the records from the[AdventureWorks].[Production].[Product] table where name contains „Touring-3000” and the records are more that 5. In case the number is more that 100, then the number is large, if the number is less that 10 the number is small, else the number is medium.

DECLARE @Number int;

SET @Number = 50;

IF @Number > 100

PRINT 'The number is large.';

ELSE

BEGIN

IF @Number < 10

PRINT 'The number is small.';

ELSE

PRINT 'The number is medium.';

END

*The result is: “The number is medium.”*

1. Check that the product name Crown Race is present in the [AdventureWorks].[Production].[Product] table. If the product is present print 'Record exits' else 'Record doesn't exist'.

if (select count(\*)

from [AdventureWorks].[Production].[Product]

where Name='Crown Race')>0

Print 'Record exits'

ELSE

Print 'Record doesn't exist'

*The result is: “Record exits”*

## IF…EXISTS

IF…EXISTS imposes conditions on the execution of a statement. The statement that follows an IF keyword and its condition is executed, unless the condition is satisfied: the Boolean expression returns TRUE. The optional ELSE keyword introduces another statement that is executed when the IF condition is not satisfied: the Boolean expression returns FALSE.

**Syntax:**

IF Boolean\_expression

     { sql\_statement | statement\_block }

[ ELSE

     { sql\_statement | statement\_block } ]

1. If exists, product name equal with ‚Crown’ in the [AdventureWorks].[Production].[Product] table, then print that record exist else that doesn’t exist.

We use the following IF…EXISTS statement:

if EXISTS (select \*

from [AdventureWorks].[Production].[Product]

where Name = 'Crown')

Print 'Record exits - Update'

ELSE

Print 'Record doesn't exist - Insert'

*The result is: „Record doesn't exist – Insert”*

1. If the name „Testing” does not exist in the [AdventureWorks].[HumanResources].[Department]table, then insert into the table the following record: Name=HeadTesting,GroupName=Testing, ModifiedDate=1998-06-01.

IF NOT EXISTS (SELECT \* FROM [AdventureWorks].[HumanResources].[Department] WHERE Name='Testing')

BEGIN

INSERT INTO [AdventureWorks].[HumanResources].[Department] (Name,GroupName,ModifiedDate)

Values('Testing','HeadofTesting','1998-06-01');

END

The results are:

|  |  |  |  |
| --- | --- | --- | --- |
| DepartmentID | Name | GroupName | ModifiedDate |
| 1 | Engineering | Research and Development | 1998-06-01 00:00:00.000 |
| 2 | Tool Design | Research and Development | 1998-06-01 00:00:00.000 |
| 3 | Sales | Sales and Marketing | 1998-06-01 00:00:00.000 |
| 4 | Marketing | Sales and Marketing | 1998-06-01 00:00:00.000 |
| 5 | Purchasing | Inventory Management | 1998-06-01 00:00:00.000 |
| 6 | Research and Development | Research and Development | 1998-06-01 00:00:00.000 |
| 7 | Production | Manufacturing | 1998-06-01 00:00:00.000 |
| 8 | Production Control | Manufacturing | 1998-06-01 00:00:00.000 |
| 9 | Human Resources | Executive General and Administration | 1998-06-01 00:00:00.000 |
| 10 | Finance | Executive General and Administration | 1998-06-01 00:00:00.000 |
| 11 | Information Services | Executive General and Administration | 1998-06-01 00:00:00.000 |
| 12 | Document Control | Quality Assurance | 1998-06-01 00:00:00.000 |
| 13 | Quality Assurance | Quality Assurance | 1998-06-01 00:00:00.000 |
| 14 | Facilities and Maintenance | Executive General and Administration | 1998-06-01 00:00:00.000 |
| 15 | Shipping and Receiving | Inventory Management | 1998-06-01 00:00:00.000 |
| 16 | Executive | Executive General and Administration | 1998-06-01 00:00:00.000 |
| 17 | **Testing** | **HeadofTesting** | **1998-06-01 00:00:00.000** |

## CASE

CASE statment 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.

1. When product line is R then category should be ‘Road’; if M then ‘Mountain’; if T then ‘Touring’; if S then ‘Other sale items’, else not for sale.

We use the following SELECT statement:

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 [AdventureWorks].[Production].[Product]

ORDER BY ProductNumber

The results are:

|  |  |  |
| --- | --- | --- |
| ProductNumber | Category | Name |
| AR-5381 | Not for sale | Adjustable Race |
| BA-8327 | Not for sale | Bearing Ball |
| BB-7421 | Not for sale | LL Bottom Bracket |
| BB-8107 | Not for sale | ML Bottom Bracket |
| BB-9108 | Not for sale | HL Bottom Bracket |
| BC-M005 | Mountain | Mountain Bottle Cage |
| BC-R205 | Road | Road Bottle Cage |
| BE-2349 | Not for sale | BB Ball Bearing |
| BE-2908 | Not for sale | Headset Ball Bearings |
| BK-M18B-40 | Mountain | Mountain-500 Black, 40 |
| BK-M18B-42 | Mountain | Mountain-500 Black, 42 |
| BK-M18B-44 | Mountain | Mountain-500 Black, 44 |
| BK-M18B-48 | Mountain | Mountain-500 Black, 48 |

*...*

*504 rows selected*

1. When colour is black and FinishedFoodsFlag is 1 then 1, else should be 0.

We use the following SELECT statement:

SELECT CAST(

CASE

WHEN Color='black' and FinishedGoodsFlag = '1'

THEN 1

ELSE 0

END AS bit) as Salable, Color, FinishedGoodsFlag

FROM [AdventureWorks].[Production].[Product]

The results are:

|  |  |  |
| --- | --- | --- |
| Salable | Color | FinishedGoodsFlag |
| 0 | Multi | 1 |
| 0 | Multi | 1 |
| 0 | Red | 1 |
| 0 | Red | 1 |
| 0 | Red | 1 |
| 0 | Red | 1 |
| 0 | Red | 1 |
| 1 | **Black** | **1** |
| 1 | **Black** | **1** |
| 1 | Black | 1 |
| 0 | Red | 1 |
| 0 | Red | 1 |
| 0 | Red | 1 |

*...*

*504 rows selected*

## BEGIN TRAN…END TRAN

BEGIN TRANSACTION represents a point at which the data referenced by a connection is logically and physically consistent. If errors are encountered, all data modifications made after the BEGIN TRANSACTION can be rolled back to return the data to this known state of consistency. Each transaction lasts until either it completes without errors and COMMIT TRANSACTION is issued to make the modifications a permanent part of the database, or errors are encountered and all modifications are erased with a ROLLBACK TRANSACTION statement.

BEGIN TRANSACTION starts a local transaction for the connection issuing the statement. Depending on the current transaction isolation level settings, many resources acquired to support the Transact-SQL statements issued by the connection are locked by the transaction until it is completed with either a COMMIT TRANSACTION or ROLLBACK TRANSACTION statement. Transactions left outstanding for long periods of time can prevent other users from accessing these locked resources, and also can prevent log truncation.

Although BEGIN TRANSACTION starts a local transaction, it is not recorded in the transaction log until the application subsequently performs an action that must be recorded in the log, such as executing an INSERT, UPDATE, or DELETE statement. An application can perform actions such as acquiring locks to protect the transaction isolation level of SELECT statements, but nothing is recorded in the log until the application performs a modification action.

Naming multiple transactions in a series of nested transactions with a transaction name has little effect on the transaction. Only the first (outermost) transaction name is registered with the system. A rollback to any other name (other than a valid savepoint name) generates an error. None of the statements executed before the rollback is, in fact, rolled back at the time this error occurs. The statements are rolled back only when the outer transaction is rolled back.

The local transaction started by the BEGIN TRANSACTION statement is escalated to a distributed transaction if the following actions are performed before the statement is committed or rolled back:

* An INSERT, DELETE, or UPDATE statement that references a remote table on a linked server is executed. The INSERT, UPDATE, or DELETE statement fails if the OLE DB provider used to access the linked server does not support the **ITransactionJoin** interface.
* A call is made to a remote stored procedure when the REMOTE\_PROC\_TRANSACTIONS option is set to ON.

The BEGIN TRANSACTION syntax is:

BEGIN { TRAN | TRANSACTION }

    [ { transaction\_name | @tran\_name\_variable }

      [ WITH MARK [ 'description' ] ]

    ]

[ ; ]

where:

* **transaction\_name** – is the name assigned to the transaction. **transaction\_name** must conform to the rules for identifiers, but identifiers longer than 32 characters are not allowed. Use transaction names only on the outermost pair of nested BEGIN...COMMIT or BEGIN...ROLLBACK statements.
* **@tran\_name\_variable –** is the name of a user-defined variable containing a valid transaction name. The variable must be declared with a char, varchar, nchar, or nvarchar data type. If more than 32 characters are passed to the variable, only the first 32 characters will be used; the remaining characters will be truncated.
* **WITH MARK [ 'description' ] –** Specifies that the transaction is marked in the log. **description** is a string that describes the mark. A description value longer than 128 characters is truncated to 128 characters before being stored in the msdb.dbo.logmarkhistory table.

If WITH MARK is used, a transaction name must be specified. WITH MARK allows for restoring a transaction log to a named mark.

For example,from the[AdventureWorks].[HumanResources].[JobCandidate] delete the record where JobCandidateID is 13, then commit the transaction.

BEGIN TRANSACTION CandidateDelete

WITH MARK N'Deleting a Job Candidate';

GO

USE [AdventureWorks].[HumanResources].[JobCandidate];

GO

DELETE FROM [AdventureWorks].[HumanResources].[JobCandidate]

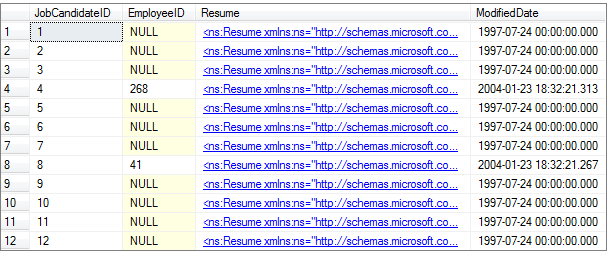
WHERE JobCandidateID = 13;

GO

COMMIT TRANSACTION CandidateDelete;

GO

The results are:

****

1. BEGIN… COMMIT TRANSACTION result-set

# Bibliography

1. 3WSchools SQL Tutorial (<http://www.w3schools.com/sql>);
2. MSDN Library SQL Online Help Guide (<http://msdn.microsoft.com/en-us/library/bb545450.aspx>);
3. Microsoft SQL Server Black Book (<http://portal.aauj.edu/portal_resources/downloads/database/microsoft_sqlL_server_blackbook.pdf>);
4. *The EXCEPT and INTERSECT Operators in SQL Server* on Simple Talk (<http://www.simple-talk.com/sql/performance>).
5. <http://www.codeproject.com/KB/database/sqlviews.aspx>
6. <http://www.techonthenet.com/sql/views.php>
7. <http://www.1keydata.com/sql/sqlunionall.html>
8. <http://www.tizag.com/sqlTutorial/sqlunion.php>

# APPENDIX: Numeric Functions

The following numeric functions are used similar to the ROUND and ABS ones:

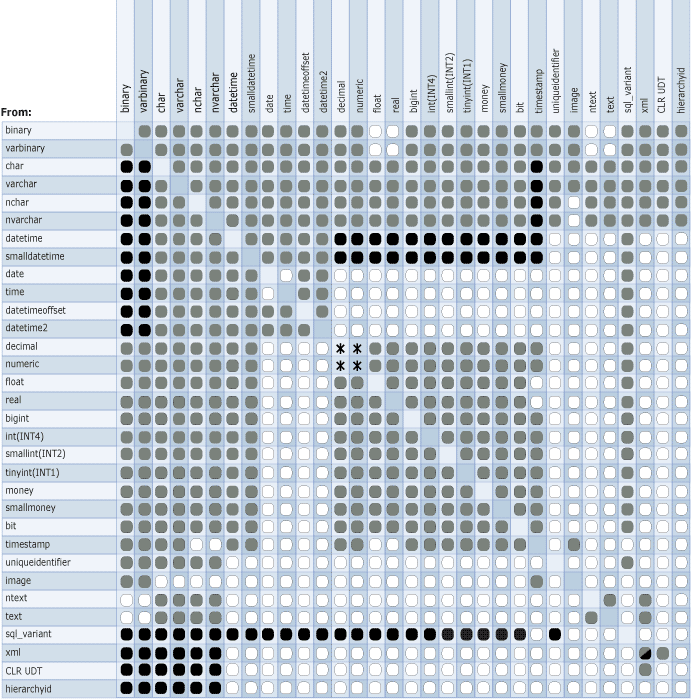
|  |  |
| --- | --- |
| Function | Description |
| CEILING(numeric value) | returns the next larger integer value – when a number contains decimal places. |
| FLOOR(numeric value) | returns the next lower integer value – when a number contains decimal places. |
| SQRT(numeric value) | returns the square root of positive numeric values. |
| SQUARE(numeric value) | returns a number squared. |

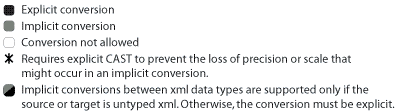
# APPENDIX: Data Types

The next SQL Server data types are most commonly used:

|  |  |
| --- | --- |
| Data Type | Description |
| Character Strings | |
| char(n) | Fixed-length character string. Maximum 8,000 characters. |
| varchar(n) | Variable-length character string. Maximum 8,000 characters. |
| varchar(max) | Variable-length character string. Maximum 1,073,741,824 characters. |
| Unicode Strings | |
| nchar(n) | Fixed-length Unicode data. Maximum 4,000 characters. |
| nvarchar(n) | Variable-length Unicode data. Maximum 4,000 characters. |
| nvarchar(max) | Variable-length Unicode data. Maximum 536,870,912 characters. |
| ntext | Variable-length Unicode data. Maximum 2GB of text data. |
| Binary Types | |
| bit | Allows 0, 1, or NULL |
| binary(n) | Fixed-length binary data; maximum 8,000 bytes. |
| varbinary(n) | Variable-length binary data; maximum 8,000 bytes. |
| varbinary(max) | Variable-length binary data; maximum 2GB. |
| image | Variable-length binary data; maximum 2GB. |
| Number Types | |
| tinyint | Allows whole numbers from 0 to 255. |
| smallint | Allows whole numbers between -32,768 and 32,767. |
| int | Allows whole numbers between -2,147,483,648 and 2,147,483,647. |
| bigint | Allows whole numbers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807. |
| decimal(p,s) | Fixed precision and scale numbers (maximum total number of digits that can be stored).  Allows numbers from -10^38 +1 to 10^38 –1.  p must be a value from 1 to 38. Default is 18.  s must be a value from 0 to p. Default value is 0. |
| numeric(p,s) | “numeric”,  functionally, is equivalent to “decimal”. SQL server treats them the same. |
| float(n) | Floating precision number data from -1.79E + 308 to 1.79E + 308. The n parameter indicates whether the field should hold 4 or 8 bytes. float(24) holds a 4-byte field and float(53) holds an 8-byte field. Default value of n is 53. |
| real | Floating precision number data from -3.40E + 38 to 3.40E + 38. |
| smallmoney | Monetary data from -214,748.3648 to 214,748.3647. |
| Date Types | |
| datetime | From January 1, 1753 to December 31, 9999 with an accuracy of 3.33 milliseconds. |
| datetime2 | From January 1, 0001 to December 31, 9999 with an accuracy of 100 nanoseconds. |
| smalldatetime | From January 1, 1900 to June 6, 2079 with an accuracy of 1 minute. |
| date | Store a date only. From January 1, 0001 to December 31, 9999. |
| time | Store a time only to an accuracy of 100 nanoseconds. |
| datetimeoffset | The same as datetime2 with the addition of a time zone offset. |
| timestamp | Stores a unique number that gets updated every time a row gets created or modified. The timestamp value is based upon an internal clock and does not correspond to real time. Each table may have only one timestamp variable. |
| Other Data Types | |
| sql\_variant | Stores up to 8,000 bytes of data of various data types, except text, ntext, and timestamp. |
| uniqueidentifier | Stores a globally unique identifier (GUID). |
| xml | Stores XML formatted data. Maximum 2GB. |
| cursor | Stores a reference to a cursor used for database operations. |
| table | Stores a result-set for later processing. |

# C. APPENDIX: Implicit Conversions





# D. APPENDIX: Create View Examples

Here are more CREATE VIEW examples for your thorough consideration:

|  |  |
| --- | --- |
| No. | Example |
| 1. | USE [dbIMC]  GO  IF EXISTS  (  SELECT \* FROM sys.views  WHERE object\_id = OBJECT\_ID(N'[dbo].[viewRenewalExpiringTokenReplacementCreditSummary]')  )  DROP VIEW [dbo].[viewRenewalExpiringTokenReplacementCreditSummary]  GO  USE [dbIMC]  GO  CREATE VIEW [dbo].[viewRenewalExpiringTokenReplacementCreditSummary]  AS  SELECT  nServiceProfileID,  cReplacementPartcode,  COUNT(1) AS nQuantity  FROM viewRenewalExpiringTokenReplacementDetails  GROUP BY  nServiceProfileID,  cReplacementPartcode  GO |
| 2. | USE [dbIMC]  GO  IF EXISTS  (  SELECT \* FROM sys.views  WHERE object\_id = OBJECT\_ID(N'[dbo].[viewRenewalExpiringTokenCreditSummary]')  )  DROP VIEW [dbo].[viewRenewalExpiringTokenCreditSummary]  GO  USE [dbIMC]  GO  CREATE VIEW [dbo].[viewRenewalExpiringTokenCreditSummary]  AS  SELECT  nServiceProfileID,  cReplacementPartcode AS cPartcode,  COUNT(1) AS nQuantity  FROM viewRenewalExpiringTokenReplacementDetails  GROUP BY  nServiceProfileID,  cReplacementPartcode  UNION ALL  SELECT  nServiceProfileID,  cProvisioningPartcode AS cPartcode,  CEILING(COUNT(1)/ (CASE  WHEN cProvisioningPartcode LIKE ('%25%') THEN 25.0  WHEN cProvisioningPartcode LIKE ('%60%') THEN 60.0  ELSE 1.0  END)) AS nQuantity  FROM viewRenewalExpiringTokenProviosioningCreditSummary  GROUP BY  cProvisioningPartcode, nServiceProfileID  GO |
| 3. | USE [dbIMC]  GO  IF EXISTS (SELECT \* FROM sys.views WHERE object\_id = OBJECT\_ID(N'[dbo].[viewRenewalExpiringTokenProviosioningCreditSummary]'))  DROP VIEW [dbo].[viewRenewalExpiringTokenProviosioningCreditSummary]  GO  USE [dbIMC]  GO  CREATE VIEW [dbo].[viewRenewalExpiringTokenProviosioningCreditSummary]  AS  SELECT  proviosioningPartcodes.nServiceProfileID,  proviosioningPartcodes.nTokenID,  proviosioningPartcodes.cProvisioningPartcode  FROM  (  SELECT  v.nServiceProfileID,  v.nTokenID,  CASE  WHEN v.nShippingMethod = 2 THEN  CASE c.nShippingRegion  WHEN 1 THEN 'FF-UK-BAT-60'  WHEN 2 THEN 'FF-BAT25-EU-COU'  WHEN 3 THEN 'FF-BAT25-US-COU'  ELSE 'FF-BAT25-ROW-COU'  END  WHEN v.nShippingMethod = 1 THEN  CASE c.nShippingRegion  WHEN 1 THEN 'FF-USR-UK-SD-PREPAY'  WHEN 2 THEN 'FF-USR-EU-COU-PREPAY'  WHEN 3 THEN 'FF-USR-US-COU-PREPAY'  ELSE 'FF-USR-US-COU-PREPAY'  END  END cProvisioningPartcode  FROM  viewRenewalExpiringTokenReplacementDetails v  JOIN tblCountries c ON c.cDescription = v.cShippingDestinationCountry  WHERE  nShippingMethod IS NOT NULL  UNION ALL  SELECT  v.nServiceProfileID,  v.nTokenID,  'FF-USR-UK-SD-PREPAY ' cPartCode  FROM  viewRenewalExpiringTokenReplacementDetails v  JOIN tblTokens t On t.nTokenID = v.nTokenID  WHERE  ISNULL(v.nShippingMethod,0) = 2  AND  t.nPersonID IS NOT NULL  ) AS proviosioningPartcodes  GO |
| 4. | USE [dbIMC]  GO  IF EXISTS  (  SELECT \* FROM sys.views  WHERE object\_id = OBJECT\_ID(N'[dbo].[viewRenewalExpiringTokenReplacementDetails]')  )  DROP VIEW [dbo].[viewRenewalExpiringTokenReplacementDetails]  GO  USE [dbIMC]  GO  CREATE VIEW [dbo].[viewRenewalExpiringTokenReplacementDetails]  AS  SELECT  eti.nServiceProfileID,  t.nTokenID,  t.dExpiryDate,  CASE  WHEN  --decision not made, replace with same as expire, hardware expiring token  (trpd.nReplaceWithDevice IS NULL AND  etip.nReplaceWith IS NULL AND  tt.nTokenClass = 1)  OR  --decision not made, replace with hardware preference  (trpd.nReplaceWithDevice IS NULL AND  pr\_tt.nTokenClass = 1)  OR  --decision is made, replace with hardware  (trpd.nReplaceWithDevice IS NOT NULL AND  r\_tt.nTokenClass = 1)  THEN 'CRED-SID' + CAST(etip.nRSASecurIDTokenLifeTime AS VARCHAR(1)) + 'KF-REPL-EX-PREPAY'  WHEN  --decision not made, replace with same as expire, software expiring token  (trpd.nReplaceWithDevice IS NULL AND  etip.nReplaceWith IS NULL AND  tt.nTokenClass = 2)  OR  --decision not made, replace with software preference  (trpd.nReplaceWithDevice IS NULL AND  pr\_tt.nTokenClass = 2)  OR  --decision is made, replace with software  (trpd.nReplaceWithDevice IS NOT NULL AND  r\_tt.nTokenClass = 2)  THEN 'CRED-SID' + CAST(etip.nRSASoftwareTokenLifeTime AS VARCHAR(1)) + 'SW-REPL-EX-PREPAY'  END AS cReplacementPartcode,  CASE  WHEN (trpd.nShippingOptionID = 2)  THEN 2 --to shipping contact  WHEN t.nPersonID IS NOT NULL AND  trpd.nShippingOptionID = 1 /\*individually\*/ AND  ISNULL(trpd.bAddressConfirmed, 0) = 1  THEN 1 --individually  WHEN t.nPersonID IS NOT NULL AND  trpd.nShippingOptionID = 1 /\*individually\*/ AND  ISNULL(trpd.bAddressConfirmed, 0) = 0  THEN etip.nUnconfirmedAssignedTokenOption  END AS nShippingMethod,    sha.cCountry AS cShippingDestinationCountry  FROM tblExpiringTokenInstances eti  JOIN tblExpiringTokenProcesses etp ON eti.nExpiringTokenInstanceID = etp.nExpiringTokenInstanceID  JOIN tblTokenReplacementProcessDetails trpd ON etp.nTokenReplacementProcessDetailID = trpd.nTokenReplacementProcessDetailID  JOIN tblTokens t ON trpd.nTokenID = t.nTokenID  JOIN tblTokenTypes tt ON t.nTokenTypeID = tt.nTokenTypeID  JOIN tblExpiringTokenInstancePreferences etip ON eti.nExpiringTokenInstancePreferenceID = etip.nExpiringTokenInstancePreferenceID  JOIN tblServiceProfiles sp ON eti.nServiceProfileID = sp.nServiceProfileID  LEFT JOIN tblTokenTypes r\_tt ON trpd.nReplaceWithDevice = r\_tt.nTokenTypeID  LEFT JOIN tblTokenTypes pr\_tt ON etip.nReplaceWith = pr\_tt.nTokenTypeID  LEFT JOIN tblShippingContacts shc ON trpd.nShippingContactID = shc.nShippingContactID  LEFT JOIN tblAddresses sha ON ISNULL(shc.nAddressID, trpd.nReplacementAddressID) = sha.nAddressID  WHERE  trpd.bReplace = 1 AND  sp.dServiceEndDate IS NOT NULL AND  sp.nBillingPeriodMonths IS NOT NULL AND  eti.dExpiryDate BETWEEN sp.dServiceEndDate AND  DATEADD(MM, sp.nBillingPeriodMonths, sp.dServiceEndDate)  GO |