**SELECT (Transact-SQL)**

**SQL Server 2012**

[Other Versions](javascript:;)

Description: http://i.technet.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png

* [SQL Server 2008 R2](http://technet.microsoft.com/en-us/library/ms189499(d=printer,v=sql.105).aspx)
* [SQL Server 2008](http://technet.microsoft.com/en-us/library/ms189499(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://technet.microsoft.com/en-us/library/ms189499(d=printer,v=sql.90).aspx)

Retrieves rows from the database and enables the selection of one or many rows or columns from one or many tables in SQL Server 2012. The full syntax of the SELECT statement is complex, but the main clauses can be summarized as:

[ WITH <common\_table\_expression>]

SELECT select\_list [ INTO new\_table ]

[ FROM table\_source ] [ WHERE search\_condition ]

[ GROUP BY group\_by\_expression ]

[ HAVING search\_condition ]

[ ORDER BY order\_expression [ ASC | DESC ] ]

The UNION, EXCEPT and INTERSECT operators can be used between queries to combine or compare their results into one result set.

[Transact-SQL Syntax Conventions](http://technet.microsoft.com/en-us/library/ms177563.aspx)

[Syntax](javascript:void(0))

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_8523594b-1013-46ac-b822-f90d931ac7af');" \o "Copy to clipboard.)

<SELECT statement> ::=

[WITH <common\_table\_expression> [,...n]]

    <query\_expression>

    [ ORDER BY { order\_by\_expression | column\_position [ ASC | DESC ] }

  [ ,...n ] ]

    [ <FOR Clause>]

    [ OPTION ( <query\_hint> [ ,...n ] ) ]

<query\_expression> ::=

    { <query\_specification> | ( <query\_expression> ) }

    [ { UNION [ ALL ] | EXCEPT | INTERSECT }

<query\_specification> | ( <query\_expression> ) [...n ] ]

<query\_specification> ::=

SELECT [ ALL | DISTINCT ]

    [TOP ( expression ) [PERCENT] [ WITH TIES ] ]

    < select\_list >

    [ INTO new\_table ]

    [ FROM { <table\_source> } [ ,...n ] ]

    [ WHERE <search\_condition> ]

    [ <GROUP BY> ]

    [ HAVING < search\_condition > ]

[Remarks](javascript:void(0))

Because of the complexity of the SELECT statement, detailed syntax elements and arguments are shown by clause:

|  |  |
| --- | --- |
| [WITH common\_table\_expression](http://technet.microsoft.com/en-us/library/ms175972.aspx) | [HAVING](http://technet.microsoft.com/en-us/library/ms180199.aspx) |
| [SELECT Clause](http://technet.microsoft.com/en-us/library/ms176104.aspx) | [UNION](http://technet.microsoft.com/en-us/library/ms180026.aspx) |
| [INTO Clause](http://technet.microsoft.com/en-us/library/ms188029.aspx) | [EXCEPT and INTERSECT](http://technet.microsoft.com/en-us/library/ms188055.aspx) |
| [FROM](http://technet.microsoft.com/en-us/library/ms177634.aspx) | [ORDER BY](http://technet.microsoft.com/en-us/library/ms188385.aspx) |
| [WHERE](http://technet.microsoft.com/en-us/library/ms188047.aspx) | [FOR Clause](http://technet.microsoft.com/en-us/library/ms173812.aspx) |
| [GROUP BY](http://technet.microsoft.com/en-us/library/ms177673.aspx) | [OPTION Clause](http://technet.microsoft.com/en-us/library/ms190322.aspx) |

The order of the clauses in the SELECT statement is significant. Any one of the optional clauses can be omitted, but when the optional clauses are used, they must appear in the appropriate order.

SELECT statements are permitted in user-defined functions only if the select lists of these statements contain expressions that assign values to variables that are local to the functions.

A four-part name constructed with the OPENDATASOURCE function as the server-name part can be used as a table source wherever a table name can appear in a SELECT statement.

Some syntax restrictions apply to SELECT statements that involve remote tables.

**Logical Processing Order of the SELECT statement**

The following steps show the logical processing order, or binding order, for a SELECT statement. This order determines when the objects defined in one step are made available to the clauses in subsequent steps. For example, if the query processor can bind to (access) the tables or views defined in the FROM clause, these objects and their columns are made available to all subsequent steps. Conversely, because the SELECT clause is step 8, any column aliases or derived columns defined in that clause cannot be referenced by preceding clauses. However, they can be referenced by subsequent clauses such as the ORDER BY clause. Note that the actual physical execution of the statement is determined by the query processor and the order may vary from this list.

1. FROM
2. ON
3. JOIN
4. WHERE
5. GROUP BY
6. WITH CUBE or WITH ROLLUP
7. HAVING
8. SELECT
9. DISTINCT
10. ORDER BY
11. TOP

[Permissions](javascript:void(0))

Selecting data requires SELECT permission on the table or view, which could be inherited from a higher scope such as SELECT permission on the schema or CONTROL permission on the table. Or requires membership in the db\_datareader or db\_owner fixed database roles, or the sysadmin fixed server role. Creating a new table using SELECT INTO also requires both the CREATE TABLE permission, and the ALTER SCHEMA permission on the schema that owns the new table.

**SELECT Examples (Transact-SQL)**

**SQL Server 2012**

[Other Versions](javascript:;)

Description: http://i.technet.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png

* [SQL Server 2008 R2](http://technet.microsoft.com/en-us/library/ms187731(d=printer,v=sql.105).aspx)
* [SQL Server 2008](http://technet.microsoft.com/en-us/library/ms187731(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://technet.microsoft.com/en-us/library/ms187731(d=printer,v=sql.90).aspx)

This topic provides examples of using the [SELECT](http://technet.microsoft.com/en-us/library/ms189499.aspx) statement.

[A. Using SELECT to retrieve rows and columns](javascript:void(0))

The following example shows three code examples. This first code example returns all rows (no WHERE clause is specified) and all columns (using the \*) from the Product table in the AdventureWorks2012 database.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT \*

FROM Production.Product

ORDER BY Name ASC;

-- Alternate way.

USE AdventureWorks2012;

GO

SELECT p.\*

FROM Production.Product AS p

ORDER BY Name ASC;

GO

This example returns all rows (no WHERE clause is specified), and only a subset of the columns (Name, ProductNumber, ListPrice) from the Product table in the AdventureWorks2012 database. Additionally, a column heading is added.

USE AdventureWorks2012;

GO

SELECT Name, ProductNumber, ListPrice AS Price

FROM Production.Product

ORDER BY Name ASC;

GO

This example returns only the rows for Product that have a product line of R and that have days to manufacture that is less than 4.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT Name, ProductNumber, ListPrice AS Price

FROM Production.Product

WHERE ProductLine = 'R'

AND DaysToManufacture < 4

ORDER BY Name ASC;

GO

[B. Using SELECT with column headings and calculations](javascript:void(0))

The following examples return all rows from the Product table. The first example returns total sales and the discounts for each product. In the second example, the total revenue is calculated for each product.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT p.Name AS ProductName,

NonDiscountSales = (OrderQty \* UnitPrice),

Discounts = ((OrderQty \* UnitPrice) \* UnitPriceDiscount)

FROM Production.Product AS p

INNER JOIN Sales.SalesOrderDetail AS sod

ON p.ProductID = sod.ProductID

ORDER BY ProductName DESC;

GO

This is the query that calculates the revenue for each product in each sales order.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT 'Total income is', ((OrderQty \* UnitPrice) \* (1.0 - UnitPriceDiscount)), ' for ',

p.Name AS ProductName

FROM Production.Product AS p

INNER JOIN Sales.SalesOrderDetail AS sod

ON p.ProductID = sod.ProductID

ORDER BY ProductName ASC;

GO

[C. Using DISTINCT with SELECT](javascript:void(0))

The following example uses DISTINCT to prevent the retrieval of duplicate titles.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT DISTINCT JobTitle

FROM HumanResources.Employee

ORDER BY JobTitle;

GO

[D. Creating tables with SELECT INTO](javascript:void(0))

The following first example creates a temporary table named #Bicycles in tempdb.

Transact-SQL

USE tempdb;

GO

IF OBJECT\_ID (N'#Bicycles',N'U') IS NOT NULL

DROP TABLE #Bicycles;

GO

SELECT \*

INTO #Bicycles

FROM AdventureWorks2012.Production.Product

WHERE ProductNumber LIKE 'BK%';

GO

This second example creates the permanent table NewProducts.

Transact-SQL

USE AdventureWorks2012;

GO

IF OBJECT\_ID('dbo.NewProducts', 'U') IS NOT NULL

DROP TABLE dbo.NewProducts;

GO

ALTER DATABASE AdventureWorks2012 SET RECOVERY BULK\_LOGGED;

GO

SELECT \* INTO dbo.NewProducts

FROM Production.Product

WHERE ListPrice > $25

AND ListPrice < $100;

GO

ALTER DATABASE AdventureWorks2012 SET RECOVERY FULL;

GO

[E. Using correlated subqueries](javascript:void(0))

The following example shows queries that are semantically equivalent and illustrates the difference between using the EXISTS keyword and the IN keyword. Both are examples of a valid subquery that retrieves one instance of each product name for which the product model is a long sleeve logo jersey, and the ProductModelID numbers match between the Product and ProductModel tables.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT DISTINCT Name

FROM Production.Product AS p

WHERE EXISTS

(SELECT \*

FROM Production.ProductModel AS pm

WHERE p.ProductModelID = pm.ProductModelID

AND pm.Name LIKE 'Long-Sleeve Logo Jersey%');

GO

-- OR

USE AdventureWorks2012;

GO

SELECT DISTINCT Name

FROM Production.Product

WHERE ProductModelID IN

(SELECT ProductModelID

FROM Production.ProductModel

WHERE Name LIKE 'Long-Sleeve Logo Jersey%');

GO

The following example uses IN in a correlated, or repeating, subquery. This is a query that depends on the outer query for its values. The query is executed repeatedly, one time for each row that may be selected by the outer query. This query retrieves one instance of the first and last name of each employee for which the bonus in the SalesPerson table is 5000.00 and for which the employee identification numbers match in the Employee and SalesPerson tables.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT DISTINCT p.LastName, p.FirstName

FROM Person.Person AS p

JOIN HumanResources.Employee AS e

ON e.BusinessEntityID = p.BusinessEntityID WHERE 5000.00 IN

(SELECT Bonus

FROM Sales.SalesPerson AS sp

WHERE e.BusinessEntityID = sp.BusinessEntityID);

GO

The previous subquery in this statement cannot be evaluated independently of the outer query. It requires a value for Employee.EmployeeID, but this value changes as the SQL Server Database Engine examines different rows in Employee.

A correlated subquery can also be used in the HAVING clause of an outer query. This example finds the product models for which the maximum list price is more than twice the average for the model.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT p1.ProductModelID

FROM Production.Product AS p1

GROUP BY p1.ProductModelID

HAVING MAX(p1.ListPrice) >= ALL

(SELECT AVG(p2.ListPrice)

FROM Production.Product AS p2

WHERE p1.ProductModelID = p2.ProductModelID);

GO

This example uses two correlated subqueries to find the names of employees who have sold a particular product.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT DISTINCT pp.LastName, pp.FirstName

FROM Person.Person pp JOIN HumanResources.Employee e

ON e.BusinessEntityID = pp.BusinessEntityID WHERE pp.BusinessEntityID IN

(SELECT SalesPersonID

FROM Sales.SalesOrderHeader

WHERE SalesOrderID IN

(SELECT SalesOrderID

FROM Sales.SalesOrderDetail

WHERE ProductID IN

(SELECT ProductID

FROM Production.Product p

WHERE ProductNumber = 'BK-M68B-42')));

GO

[F. Using GROUP BY](javascript:void(0))

The following example finds the total of each sales order in the database.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT SalesOrderID, SUM(LineTotal) AS SubTotal

FROM Sales.SalesOrderDetail

GROUP BY SalesOrderID

ORDER BY SalesOrderID;

GO

Because of the GROUP BY clause, only one row containing the sum of all sales is returned for each sales order.

[G. Using GROUP BY with multiple groups](javascript:void(0))

The following example finds the average price and the sum of year-to-date sales, grouped by product ID and special offer ID.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT ProductID, SpecialOfferID, AVG(UnitPrice) AS [Average Price],

SUM(LineTotal) AS SubTotal

FROM Sales.SalesOrderDetail

GROUP BY ProductID, SpecialOfferID

ORDER BY ProductID;

GO

[H. Using GROUP BY and WHERE](javascript:void(0))

The following example puts the results into groups after retrieving only the rows with list prices greater than $1000.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT ProductModelID, AVG(ListPrice) AS [Average List Price]

FROM Production.Product

WHERE ListPrice > $1000

GROUP BY ProductModelID

ORDER BY ProductModelID;

GO

[I. Using GROUP BY with an expression](javascript:void(0))

The following example groups by an expression. You can group by an expression if the expression does not include aggregate functions.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT AVG(OrderQty) AS [Average Quantity],

NonDiscountSales = (OrderQty \* UnitPrice)

FROM Sales.SalesOrderDetail

GROUP BY (OrderQty \* UnitPrice)

ORDER BY (OrderQty \* UnitPrice) DESC;

GO

[J. Using GROUP BY with ORDER BY](javascript:void(0))

The following example finds the average price of each type of product and orders the results by average price.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT ProductID, AVG(UnitPrice) AS [Average Price]

FROM Sales.SalesOrderDetail

WHERE OrderQty > 10

GROUP BY ProductID

ORDER BY AVG(UnitPrice);

GO

[K. Using the HAVING clause](javascript:void(0))

The first example that follows shows a HAVING clause with an aggregate function. It groups the rows in the SalesOrderDetail table by product ID and eliminates products whose average order quantities are five or less. The second example shows a HAVING clause without aggregate functions.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT ProductID

FROM Sales.SalesOrderDetail

GROUP BY ProductID

HAVING AVG(OrderQty) > 5

ORDER BY ProductID;

GO

This query uses the LIKE clause in the HAVING clause.

USE AdventureWorks2012 ;

GO

SELECT SalesOrderID, CarrierTrackingNumber

FROM Sales.SalesOrderDetail

GROUP BY SalesOrderID, CarrierTrackingNumber

HAVING CarrierTrackingNumber LIKE '4BD%'

ORDER BY SalesOrderID ;

GO

[L. Using HAVING and GROUP BY](javascript:void(0))

The following example shows using GROUP BY, HAVING, WHERE, and ORDER BY clauses in one SELECT statement. It produces groups and summary values but does so after eliminating the products with prices over $25 and average order quantities under 5. It also organizes the results by ProductID.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT ProductID

FROM Sales.SalesOrderDetail

WHERE UnitPrice < 25.00

GROUP BY ProductID

HAVING AVG(OrderQty) > 5

ORDER BY ProductID;

GO

[M. Using HAVING with SUM and AVG](javascript:void(0))

The following example groups the SalesOrderDetail table by product ID and includes only those groups of products that have orders totaling more than $1000000.00 and whose average order quantities are less than 3.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT ProductID, AVG(OrderQty) AS AverageQuantity, SUM(LineTotal) AS Total

FROM Sales.SalesOrderDetail

GROUP BY ProductID

HAVING SUM(LineTotal) > $1000000.00

AND AVG(OrderQty) < 3;

GO

To see the products that have had total sales greater than $2000000.00, use this query:

Transact-SQL

USE AdventureWorks2012;

GO

SELECT ProductID, Total = SUM(LineTotal)

FROM Sales.SalesOrderDetail

GROUP BY ProductID

HAVING SUM(LineTotal) > $2000000.00;

GO

If you want to make sure there are at least one thousand five hundred items involved in the calculations for each product, use HAVING COUNT(\*) > 1500 to eliminate the products that return totals for fewer than 1500 items sold. The query looks like this:

Transact-SQL

USE AdventureWorks2012;

GO

SELECT ProductID, SUM(LineTotal) AS Total

FROM Sales.SalesOrderDetail

GROUP BY ProductID

HAVING COUNT(\*) > 1500;

GO

[N. Using the INDEX optimizer hint](javascript:void(0))

The following example shows two ways to use the INDEX optimizer hint. The first example shows how to force the optimizer to use a nonclustered index to retrieve rows from a table, and the second example forces a table scan by using an index of 0.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT pp.FirstName, pp.LastName, e.NationalIDNumber

FROM HumanResources.Employee AS e WITH (INDEX(AK\_Employee\_NationalIDNumber))

JOIN Person.Person AS pp on e.BusinessEntityID = pp.BusinessEntityID

WHERE LastName = 'Johnson';

GO

-- Force a table scan by using INDEX = 0.

USE AdventureWorks2012;

GO

SELECT pp.LastName, pp.FirstName, e.JobTitle

FROM HumanResources.Employee AS e WITH (INDEX = 0) JOIN Person.Person AS pp

ON e.BusinessEntityID = pp.BusinessEntityID

WHERE LastName = 'Johnson';

GO

[M. Using OPTION and the GROUP hints](javascript:void(0))

The following example shows how the OPTION (GROUP) clause is used with a GROUP BY clause.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT ProductID, OrderQty, SUM(LineTotal) AS Total

FROM Sales.SalesOrderDetail

WHERE UnitPrice < $5.00

GROUP BY ProductID, OrderQty

ORDER BY ProductID, OrderQty

OPTION (HASH GROUP, FAST 10);

GO

[O. Using the UNION query hint](javascript:void(0))

The following example uses the MERGE UNION query hint.

Transact-SQL

USE AdventureWorks2012;

GO

SELECT BusinessEntityID, JobTitle, HireDate, VacationHours, SickLeaveHours

FROM HumanResources.Employee AS e1

UNION

SELECT BusinessEntityID, JobTitle, HireDate, VacationHours, SickLeaveHours

FROM HumanResources.Employee AS e2

OPTION (MERGE UNION);

GO

[P. Using a simple UNION](javascript:void(0))

In the following example, the result set includes the contents of the ProductModelID and Name columns of both the ProductModel and Gloves tables.

Transact-SQL

USE AdventureWorks2012;

GO

IF OBJECT\_ID ('dbo.Gloves', 'U') IS NOT NULL

DROP TABLE dbo.Gloves;

GO

-- Create Gloves table.

SELECT ProductModelID, Name

INTO dbo.Gloves

FROM Production.ProductModel

WHERE ProductModelID IN (3, 4);

GO

-- Here is the simple union.

USE AdventureWorks2012;

GO

SELECT ProductModelID, Name

FROM Production.ProductModel

WHERE ProductModelID NOT IN (3, 4)

UNION

SELECT ProductModelID, Name

FROM dbo.Gloves

ORDER BY Name;

GO

[Q. Using SELECT INTO with UNION](javascript:void(0))

In the following example, the INTO clause in the second SELECT statement specifies that the table named ProductResults holds the final result set of the union of the designated columns of the ProductModel and Gloves tables. Note that the Gloves table is created in the first SELECT statement.

Transact-SQL

USE AdventureWorks2012;

GO

IF OBJECT\_ID ('dbo.ProductResults', 'U') IS NOT NULL

DROP TABLE dbo.ProductResults;

GO

IF OBJECT\_ID ('dbo.Gloves', 'U') IS NOT NULL

DROP TABLE dbo.Gloves;

GO

-- Create Gloves table.

SELECT ProductModelID, Name

INTO dbo.Gloves

FROM Production.ProductModel

WHERE ProductModelID IN (3, 4);

GO

USE AdventureWorks2012;

GO

SELECT ProductModelID, Name

INTO dbo.ProductResults

FROM Production.ProductModel

WHERE ProductModelID NOT IN (3, 4)

UNION

SELECT ProductModelID, Name

FROM dbo.Gloves;

GO

SELECT ProductModelID, Name

FROM dbo.ProductResults;

[R. Using UNION of two SELECT statements with ORDER BY](javascript:void(0))

The order of certain parameters used with the UNION clause is important. The following example shows the incorrect and correct use of UNION in two SELECT statements in which a column is to be renamed in the output.

Transact-SQL

USE AdventureWorks2012;

GO

IF OBJECT\_ID ('dbo.Gloves', 'U') IS NOT NULL

DROP TABLE dbo.Gloves;

GO

-- Create Gloves table.

SELECT ProductModelID, Name

INTO dbo.Gloves

FROM Production.ProductModel

WHERE ProductModelID IN (3, 4);

GO

/\* INCORRECT \*/

USE AdventureWorks2012;

GO

SELECT ProductModelID, Name

FROM Production.ProductModel

WHERE ProductModelID NOT IN (3, 4)

ORDER BY Name

UNION

SELECT ProductModelID, Name

FROM dbo.Gloves;

GO

/\* CORRECT \*/

USE AdventureWorks2012;

GO

SELECT ProductModelID, Name

FROM Production.ProductModel

WHERE ProductModelID NOT IN (3, 4)

UNION

SELECT ProductModelID, Name

FROM dbo.Gloves

ORDER BY Name;

GO

[S. Using UNION of three SELECT statements to show the effects of ALL and parentheses](javascript:void(0))

The following examples use UNION to combine the results of three tables that all have the same 5 rows of data. The first example uses UNION ALL to show the duplicated records, and returns all 15 rows. The second example uses UNION without ALL to eliminate the duplicate rows from the combined results of the three SELECT statements, and returns 5 rows.

The third example uses ALL with the first UNION and parentheses enclose the second UNION that is not using ALL. The second UNION is processed first because it is in parentheses, and returns 5 rows because the ALL option is not used and the duplicates are removed. These 5 rows are combined with the results of the first SELECT by using the UNION ALL keywords. This does not remove the duplicates between the two sets of 5 rows. The final result has 10 rows.

Transact-SQL

USE AdventureWorks2012;

GO

IF OBJECT\_ID ('dbo.EmployeeOne', 'U') IS NOT NULL

DROP TABLE dbo.EmployeeOne;

GO

IF OBJECT\_ID ('dbo.EmployeeTwo', 'U') IS NOT NULL

DROP TABLE dbo.EmployeeTwo;

GO

IF OBJECT\_ID ('dbo.EmployeeThree', 'U') IS NOT NULL

DROP TABLE dbo.EmployeeThree;

GO

SELECT pp.LastName, pp.FirstName, e.JobTitle

INTO dbo.EmployeeOne

FROM Person.Person AS pp JOIN HumanResources.Employee AS e

ON e.BusinessEntityID = pp.BusinessEntityID

WHERE LastName = 'Johnson';

GO

SELECT pp.LastName, pp.FirstName, e.JobTitle

INTO dbo.EmployeeTwo

FROM Person.Person AS pp JOIN HumanResources.Employee AS e

ON e.BusinessEntityID = pp.BusinessEntityID

WHERE LastName = 'Johnson';

GO

SELECT pp.LastName, pp.FirstName, e.JobTitle

INTO dbo.EmployeeThree

FROM Person.Person AS pp JOIN HumanResources.Employee AS e

ON e.BusinessEntityID = pp.BusinessEntityID

WHERE LastName = 'Johnson';

GO

-- Union ALL

SELECT LastName, FirstName, JobTitle

FROM dbo.EmployeeOne

UNION ALL

SELECT LastName, FirstName ,JobTitle

FROM dbo.EmployeeTwo

UNION ALL

SELECT LastName, FirstName,JobTitle

FROM dbo.EmployeeThree;

GO

SELECT LastName, FirstName,JobTitle

FROM dbo.EmployeeOne

UNION

SELECT LastName, FirstName, JobTitle

FROM dbo.EmployeeTwo

UNION

SELECT LastName, FirstName, JobTitle

FROM dbo.EmployeeThree;

GO

SELECT LastName, FirstName,JobTitle

FROM dbo.EmployeeOne

UNION ALL

(

SELECT LastName, FirstName, JobTitle

FROM dbo.EmployeeTwo

UNION

SELECT LastName, FirstName, JobTitle

FROM dbo.EmployeeThree

);

GO