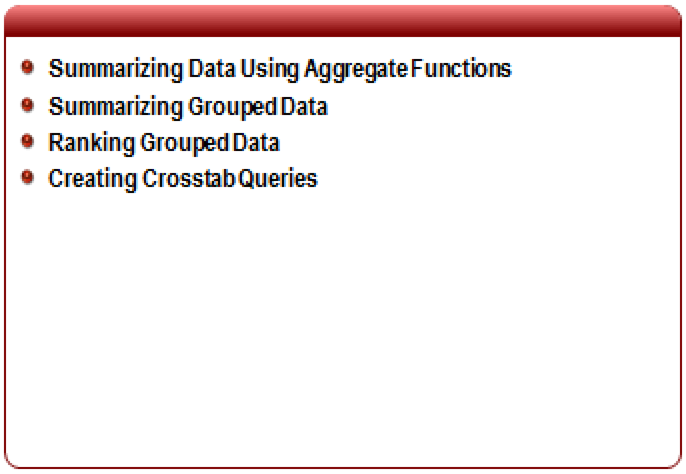
Module 3:
Grouping and Summarizing Data

Contents

Module Overview



- Summarizing Data Using Aggregate Functions
- Summarizing Grouped Data
- Ranking Grouped Data
- Creating Crosstab Queries

Data summarizing is one of the most common scenarios of T-SQL usage in business environments. It is also a very important for developers, report creators and information workers. T-SQL is built on the top of aggregate functions. One the major differences between this and other (regular) functions is that aggregations operate on sets or rows. Non-aggregate works with individual values.

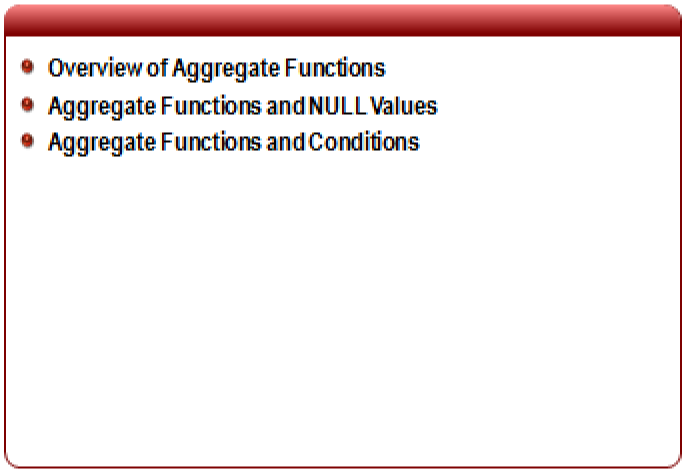
In this module, you will learn how to recognize places in data and scenarios where and when to use aggregations based on AVG, SUM, COUNT and other set based functions. You will understand usage of GROUP BY clause and filtering result sets with HAVING.

Objectives

After completing this module, you will be able to:

- Summarize data using aggregate functions
- Summarize grouped data
- Rank grouped data
- Create crosstab queries

Lesson 1: Summarizing Data Using Aggregate Functions



- Overview of Aggregate Functions
- Aggregate Functions and NULL Values
- Aggregate Functions and Conditions

SQL Server 2012 has several built in aggregate functions such as: AVG, SUM and MIN to perform summarizing data operations. Basically, those operations are taken using multiple values to produce a single (scalar) value. (E.g., average function on a column with ten thousand values will always produce a single output). NULL is a special case of missing data in tables and is treated differently from non-NULL aggregate functions and requires special attention when writing queries.

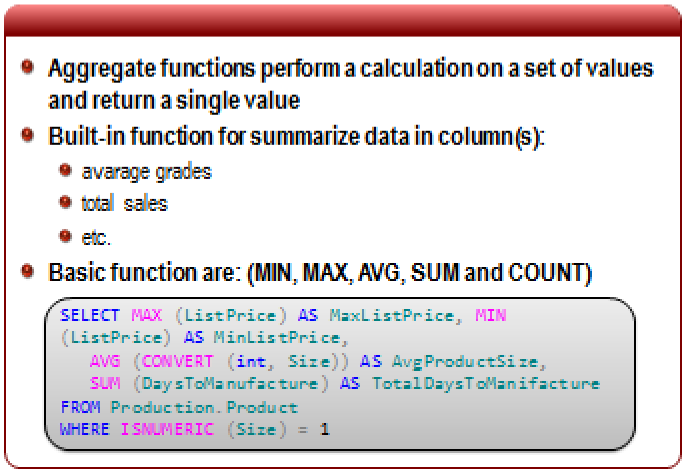
In this lesson you will learn when and how to use built-in functions to aggregate or summarize data in your database environments.

Objectives

After completing this lesson, you will be able to:

- Understand what aggregate functions are
- Understand and manage NULLs when working in aggregations
- Combine aggregate functions with conditions to produce more precise queries

Overview of Aggregate Functions



- Aggregate functions perform a calculation on a set of values and return a single value
- Built-in function for summarize data in column(s):
  - avarage grades
  - total sales
  - etc.
- Basic function are: (MIN, MAX, AVG, SUM and COUNT)

```
SELECT MAX (ListPrice) AS MaxListPrice, MIN
(ListPrice) AS MinListPrice,
   AVG (CONVERT (int, Size)) AS AvgProductSize,
   SUM (DaysToManufacture) AS TotalDaysToManifacture
FROM Production.Product
WHERE ISNUMERIC (Size) = 1
```

Data aggregation means using a specific built-in function to summarize data in column(s). Aggregate functions perform a calculation on a set of values to return a single value. Results will differ based on the functions used (E.g. average grades of students, total sales for specific product, and so on). Aggregate functions are frequently used in combination with the GROUP BY clause.

SQL Server 2012 comes with the following built-in and most commonly used aggregate functions:

Function	Example	Description
MIN	MIN (ListPrice)	Finds the smallest value in the column
MAX	MAX (Grade)	Finds the largest values in the column
SUM	SUM (TotalSales)	Creates a sum of numeric values in the column (non-null).
AVG	AVG (Size)	Creates an average of numeric values in a column (non-null)
COUNT	COUNT (OrderID) COUNT (*)	COUNT with column name counts the number of data and ignores nulls. COUNT (*) counts the number of rows in the table.

The query below uses four out of five aggregate functions to search the Production.Product table to find:

- the largest and smallest list prices,
- average size of all products, and
- the total days spent to manufacture for all products

```
SELECT MAX (ListPrice) AS MaxListPrice,
       MIN (ListPrice) AS MinListPrice,
       AVG (CONVERT (int, Size)) AS AvgProductSize,
       SUM (DaysToManufacture) AS TotalDaysToManufacture
FROM Production.Product
WHERE ISNUMERIC (Size) = 1
```

Result:

```
MaxListPrice      MinListPrice    AvgProductSize TotalDaysToManufacture
----------------- --------------- -------------- ----------------------
3578,27           54,99           48             474

(1 row(s) affected)
```

Notice that before calculating the average value on column Size we need to convert data using CONVERT to numeric data type.

The next example uses MIN and MAX functions on column SalesEndDate (datetime, data type)

```
      SELECT MIN (SellEndDate), MAX (SellEndDate)
      FROM Production.Product
```

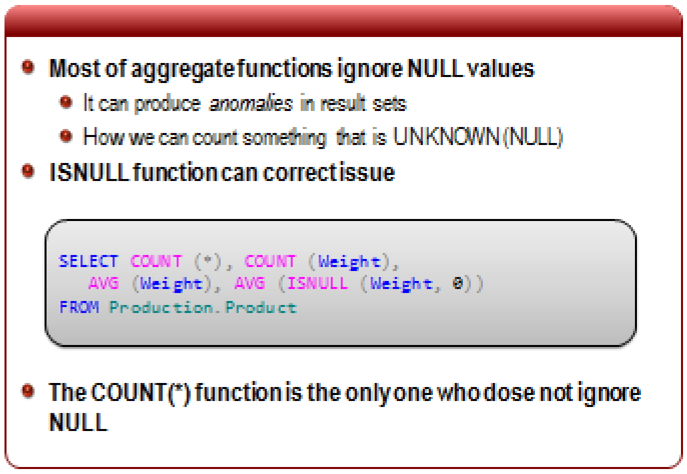
Result:

```
----------------------- -----------------------
2006-06-30 00:00:00.000 2007-06-30 00:00:00.000
Warning: Null value is eliminated by an aggregate or other SET operation.

(1 row(s) affected)
```

Things to remember when working with aggregations and summarizing data:

- An aggregate function always returns scalar (single) values.
- Various aggregate functions act different on different data types. (E.g. SUM and AVG only work with numerical data. MIN and MAX can work with either numeric or character data but cannot work on bit data type),
- Aggregate functions can only be part of SELECT statements, COMPUTE/COMPUTE BY and HAVING clauses.

Aggregate Functions and NULL Values



- Most of aggregate functions ignore NULL values
  - It can produce *anomalies* in result sets
  - How we can count something that is UNKNOWN (NULL)
- ISNULL function can correct issue

```
SELECT COUNT (*), COUNT (Weight),
   AVG (Weight), AVG (ISNULL (Weight, 0))
FROM Production.Product
```

- The COUNT(*) function is the only one who dose not ignore NULL

Most queries with aggregation functions inside ignore NULL values. This can produce *anomalies* in result sets and create confusion. The query below will count the number of rows and column data in the Production.Product table.

Things to remember when working with aggregations and NULL:

- Most aggregations ignore NULL values
- Aggregations can produce unexpected results
- Use ISNULL to specify how to treat NULL values

```
SELECT COUNT (*), COUNT (SellEndDate)
FROM Production.Product
```

Result:

```
----------- -----------
504         98
Warning: Null value is eliminated by an aggregate or other SET operation.

(1 row(s) affected)
```

COUNT (*) - counts all records in a table including NULL and duplicate values.

COUNT (SellEndDate) - ignores NULL values. You can’t count something that is UNKNOWN (NULL).

Let’s look at next example:

```
SELECT COUNT (Weight) CountOfWeights, AVG (Weight) AS Average
FROM Production.Product
```

Result:

```
CountOfWeights Average
-------------- ---------------------------------------
205            74.069219
Warning: Null value is eliminated by an aggregate or other SET operation.

(1 row(s) affected)
```

Remember from the previous example that the table has 504 rows. The column count now shows 205 rows. This means that we have NULL values and the AVG calculation may or may not be correct.

The query below includes a clause to manage NULL values. In this case we use the ISNULL set function to check for and replace NULL values with zero and then proceed with the average calculation.

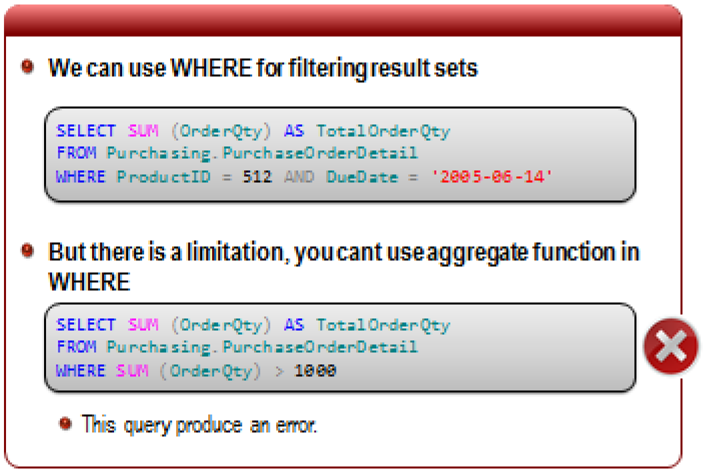
```
      SELECT COUNT (Weight),
             AVG (Weight),
             AVG (ISNULL (Weight, 0))
      FROM Production.Product
```

Result:

```
----------- --------------------------------------- ------------------------------
205         74.069219                               30.127361
Warning: Null value is eliminated by an aggregate or other SET operation.

(1 row(s) affected)
```

.

Aggregate Functions and Conditions


- We can use WHERE for filtering result sets

```
SELECT SUM (OrderQty) AS TotalOrderQty
FROM Purchasing.PurchaseOrderDetail
WHERE ProductID = 512 AND DueDate = '2005-06-14'
```

- But there is a limitation, you cant use aggregate function in WHERE

```
SELECT SUM (OrderQty) AS TotalOrderQty
FROM Purchasing.PurchaseOrderDetail
WHERE SUM (OrderQty) > 1000
```

- This query produce an error.

You can query for more specific results that go deeper than the WHERE clause by including conditional clauses that require data to meet specific requirements and excludes data that does not.

The query below uses the WHERE clause together with logical operators to fetch the sum of all order quantities for a specific product and a specific due date.

```
SELECT SUM (OrderQty) AS TotalOrderQty
FROM Purchasing.PurchaseOrderDetail
WHERE ProductID = 512 AND DueDate = '2005-06-14'
```

Result:

```
TotalOrderQty
-------------
550

(1 row(s) affected)
```

The next query uses an aggregate function in the WHERE clause to fetch the total number of Order Quantities greater than 1,000.

```
SELECT SUM (OrderQty) AS TotalOrderQty
FROM Purchasing.PurchaseOrderDetail
WHERE SUM (OrderQty) > 1000
```

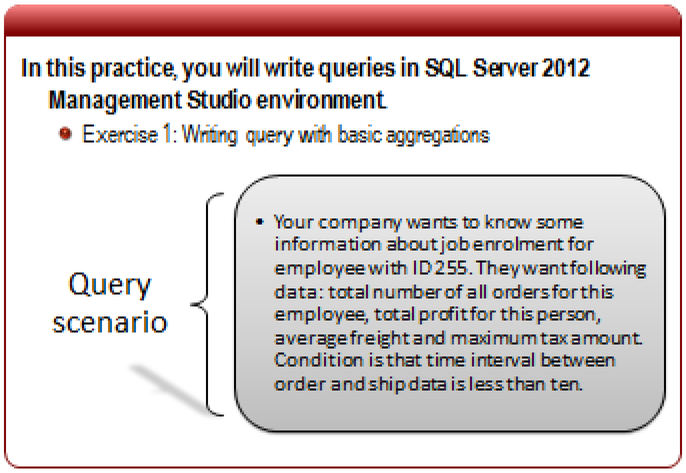
Result:

```
Msg 147, Level 15, State 1, Line 1
```

```
An aggregate may not appear in the WHERE clause unless it is in a subquery
contained in a HAVING clause or a select list, and the column being aggregated is
an outer reference.
```

This error means that we cannot use aggregations in a WHERE clause and should have been in a HAVING clause.

Practice: Basic Summarizing of Data


In this practice, you will write queries in SQL Server 2012 Management Studio environment

- Exercise 1: Writing query with basic aggregations

Query scenario

- Your company wants to know some information about job enrolment for employee with ID 255. They want following data: total number of all orders for this employee, total profit for this person, average freight and maximum tax amount. Condition is that time interval between order and ship data is less than ten.

In this practice you will write queries in SQL Server 2012 Management Studio environment.

To successfully complete the exercise you need following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
  - http://msftdbprodsamples.codeplex.com/

Exercise 1: Writing a query with basic aggregations

1. Click Start→Microsoft SQL Server 2012→SQL Server Management Studio
2. In the **Connect to Server** dialog box, under **Server name** type the name of your local instance.
   a. If the server name is *default* then change to (local)
3. Use Windows Authentication
4. From the **File** menu, click **New** and **Database Engine Query**

Query Scenario

Your company wants to know some information about job enrolment for employee with ID number 255 and require the following data: total number of all orders for this employee, total profit for the employee, average freight and maximum tax amount with a condition that the time interval between order and ship data is less than ten.

5. Type following T-SQL code in query window.

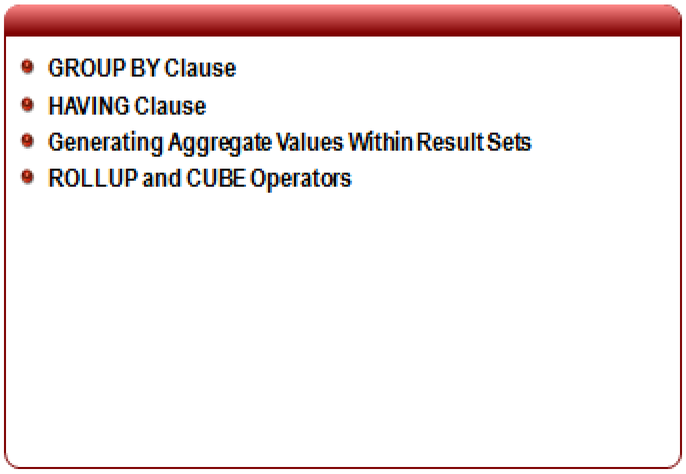
```
SELECT COUNT (EmployeeID) AS NumberOfOrders,
       SUM (SubTotal) AS TotalProfit,
       AVG (Freight) AS AverageFreight,
       MAX (TaxAmt) AS MaxTax
FROM Purchasing.PurchaseOrderHeader
WHERE EmployeeID = 255 AND
       DATEDIFF (DAY, OrderDate,ShipDate) < 10
```

6. Click on Execute or press F5 to run

Result:

```
NumberOfOrders TotalProfit           AverageFreight        MaxTax
-------------- --------------------- --------------------- ---------------------
360            5705987,175           396,2491              7289,436

(1 row(s) affected)
```

Lesson 2: Summarizing Grouped Data



- GROUP BY Clause
- HAVING Clause
- Generating Aggregate Values Within Result Sets
- ROLLUP and CUBE Operators

Often we find it useful to group data by some characteristic of the group, such as department or division, or benefit level, so that summary statistics about the group (totals, averages, etc.) can be calculated. For example, to calculate average student's grades, the user could group the grades of all students. The GROUP BY clause is used to divide the rows of a table into groups that have matching values in one or more columns.

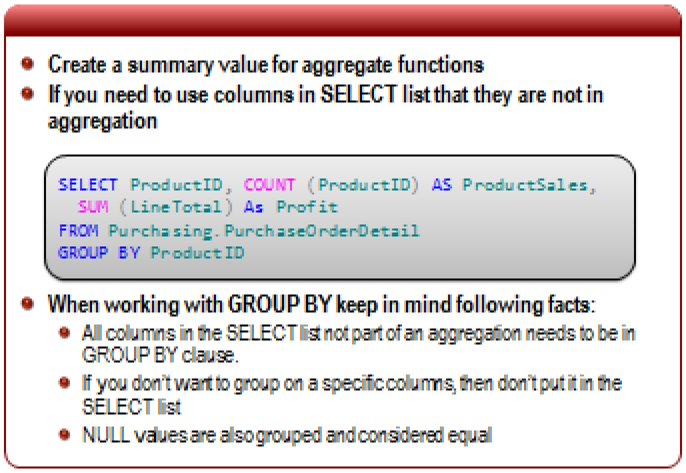
In this lesson you will learn when and how to use the GROUP BY clause and write efficient conditions using HAVING clauses. We will also cover more advanced aggregations using ROLLUP and CUBE operators.

Objectives

After completing this lesson, you will be able to:

- Use the GROUP BY clause
- Extend using conditions with HAVING clause
- Generate aggregate values within result sets
- Recognize scenarios for ROLLUP and CUBE operators
- Use COMPUTE and COMPUTE BY clauses

GROUP BY Clause



- Create a summary value for aggregate functions
- If you need to use columns in SELECT list that they are not in aggregation

```
SELECT ProductID, COUNT (ProductID) AS ProductSales,
  SUM (LineTotal) As Profit
FROM Purchasing.PurchaseOrderDetail
GROUP BY ProductID
```

- When working with GROUP BY keep in mind following facts:
  - All columns in the SELECT list not part of an aggregation needs to be in GROUP BY clause.
  - If you don't want to group on a specific columns, then don't put it in the SELECT list
  - NULL values are also grouped and considered equal

The GROUP BY clause is used to combine records with identical values in a specified field list into a single record. A summary value is created for each record when included in an aggregate function such as SUM or COUNT in the SELECT statement.

Let's examine following example and error that produce:

We want a list of all products, with a count of sales items for specific products and profit information. Query looks just fine. Then what is a problem? Problem is that COUNT and SUM returns a single (scalar) value and first column in SELECT (ProductID) returns all products. So we have conflict in result sets. Solution for this is usage GROUP BY clause.

```
SELECT ProductID, COUNT (ProductID) AS ProductSales,
                  SUM (LineTotal) As Profit
FROM Purchasing.PurchaseOrderDetail
ORDER BY ProductID
```

Result is:

```
Msg 8120, Level 16, State 1, Line 1

Column 'Purchasing.PurchaseOrderDetail.ProductID' is invalid in the select list
because it is not contained in either an aggregate function or the GROUP BY clause.
```

Now let's modified this example and run it again:

```
SELECT ProductID, COUNT (ProductID) AS ProductSales,
                  SUM (LineTotal) As Profit
FROM Purchasing.PurchaseOrderDetail
GROUP BY ProductID
ORDER BY ProductID
```

Result:

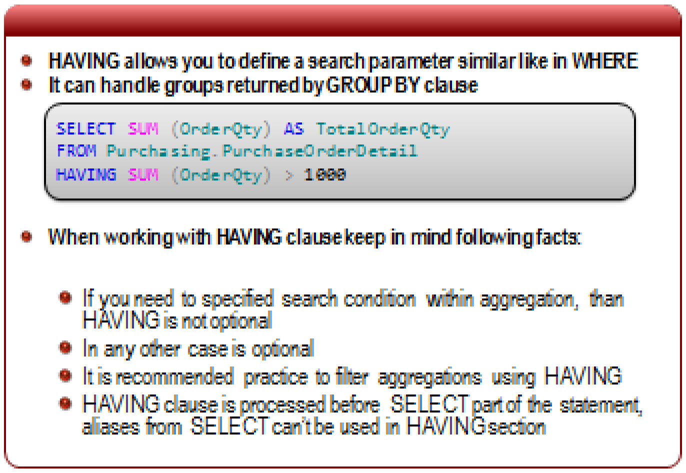
```
ProductID   ProductSales Profit
----------- ------------ ---------------------
1           51           7740,565
2           50           6287,40
4           51           8724,9015
317         80           1246014,00
318         80           1532916,00
319         130          3358797,75
320         125          17804,3985
321         125          15943,095
322         124          199625,58
323         51           7690,3155
325         50           13125,00
326         50           13125,00
332         50           297412,50
341         59           8030,5785
...
941         51           1766855,475
948         50           2277948,75
952         50           47218,50

(265 row(s) affected)
```

Explanation is quite simple. In our case, GROUPY BY ProductID create groups of unique data in result set based on data in column ProductID. (E.g. if finds same ProductID on ten places, GROUP BY take only one – group it). Then for each group makes an aggregation.

When working with GROUP BY keep in mind following facts:

- All columns in the SELECT list not part of an aggregation needs to be in GROUP BY clause.
- If you don't want to group on a specific columns, then don't put it in the SELECT list
- NULL values are also grouped and considered equal

HAVING Clause



- HAVING allows you to define a search parameter similar like in WHERE
- It can handle groups returned by GROUP BY clause

```
SELECT SUM (OrderQty) AS TotalOrderQty
FROM Purchasing.PurchaseOrderDetail
HAVING SUM (OrderQty) > 1000
```

- When working with HAVING clause keep in mind following facts:
  - If you need to specified search condition within aggregation, than HAVING is not optional
  - In any other case is optional
  - It is recommended practice to filter aggregations using HAVING
  - HAVING clause is processed before SELECT part of the statement, aliases from SELECT can't be used in HAVING section

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions and would result in the error seen in the example below.

Before we explain HAVING, let's go back for a moment on example where we were talking about WHERE in aggregations.

```
      SELECT SUM (OrderQty) AS TotalOrderQty
      FROM Purchasing.PurchaseOrderDetail
      WHERE SUM (OrderQty) > 1000
```

Result is:

```
Msg 147, Level 15, State 1, Line 1
An aggregate may not appear in the WHERE clause unless it is in a subquery
contained in a HAVING clause or a select list, and the column being aggregated is
an outer reference.
```

This error means that we cannot use aggregation in a WHERE clause. HAVING allows you to define a search parameter similar to WHERE, but in this case HAVING can handle groups returned by GROUP BY clause as seen in the example below.

```
      SELECT SUM (OrderQty) AS TotalOrderQty
      FROM Purchasing.PurchaseOrderDetail
      HAVING SUM (OrderQty) > 1000
```

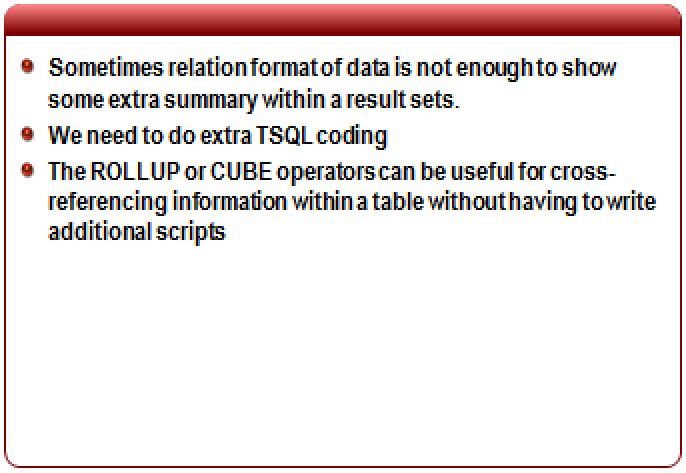
Result is:

```
TotalOrderQty
-------------
2348637

(1 row(s) affected)
```

Things to remember when working with the HAVING clause:

- If you need to specified search condition within aggregation , than HAVING is not optional
- In any other case is optional
- It is recommended practice to filter aggregations using HAVING and not WHERE
- can be used only with SELECT statements
- Because HAVING clause is processed before SELECT part of the statement, aliases from SELECT cannot be used in HAVING section.

Generating Aggregate Values within Result Sets



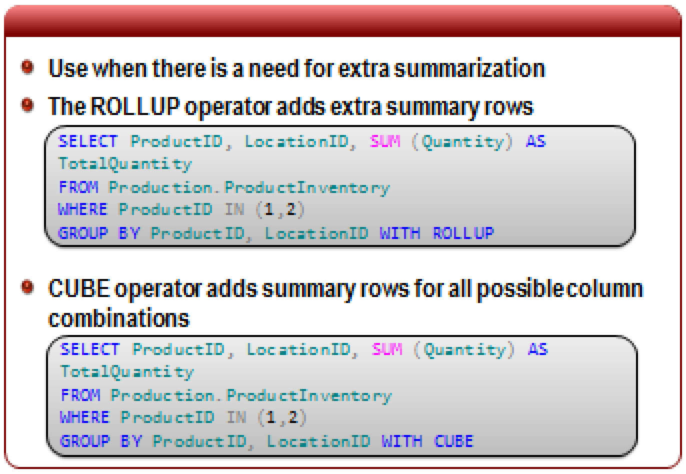
- Sometimes relation format of data is not enough to show some extra summary within a result sets.
- We need to do extra TSQL coding
- The ROLLUP or CUBE operators can be useful for cross-referencing information within a table without having to write additional scripts

Sometimes relation format of data is not enough to show some extra summary within a result sets. So we need to do extra T-SQL coding or we can see if there is another operator to help in these cases.

Use the GROUP BY clause with ROLLUP and CUBE operators to generate summary aggregate values within result sets. ROLLUP or CUBE operators can be useful for cross-referencing information within a table without having to write additional scripts.

.

ROLLUP and CUBE Operators



- Use when there is a need for extra summarization
- The ROLLUP operator adds extra summary rows

```
SELECT ProductID, LocationID, SUM (Quantity) AS
TotalQuantity
FROM Production.ProductInventory
WHERE ProductID IN (1,2)
GROUP BY ProductID, LocationID WITH ROLLUP
```

- CUBE operator adds summary rows for all possible column combinations

```
SELECT ProductID, LocationID, SUM (Quantity) AS
TotalQuantity
FROM Production.ProductInventory
WHERE ProductID IN (1,2)
GROUP BY ProductID, LocationID WITH CUBE
```

Use the GROUP BY clause with the ROLLUP operator to summarize group values. The GROUP BY clause with the ROLLUP operator provides data in a standard relational format and is used when you need additional summarizations in result sets than can be achieved the regular way.

Let's examine following example. For each pair, product-location, we want to see total quantity of products.

```
SELECT ProductID, LocationID, SUM (Quantity) AS TotalQuantity
FROM Production.ProductInventory
GROUP BY ProductID, LocationID
ORDER BY ProductID, LocationID DESC
```

Result:

```
ProductID   LocationID TotalQuantity
----------- ---------- -------------
1           50         353
1           6          324
1           1          408
2           50         364
2           6          318
2           1          427
3           50         324
3           6          443
3           1          585
...
998         7          99
999         60         116
999         7          78

(1069 row(s) affected)
```

Now let's extend functionality of last query with option to see, for each product, total sum of products (we are keeping pair product-location total quantity of items). Also we will add total sum for all products. In this case we need ROLLUP operator.

```
SELECT ProductID, LocationID, SUM (Quantity) AS TotalQuantity
FROM Production.ProductInventory
WHERE ProductID IN (1,2)
GROUP BY ProductID, LocationID WITH ROLLUP
ORDER BY ProductID, LocationID DESC
```

Note: WHERE ProductID IN (1, 2) --is only for limiting the result set

Result:

```
ProductID   LocationID TotalQuantity
----------- ---------- -------------
NULL        NULL       2194
1           50         353
1           6          324
1           1          408
1           NULL       1085
2           50         364
2           6          318
2           1          427
2           NULL       1109

(9 row(s) affected)
```

- As you can see we get extra rows in result sets, one for each product-location sums and one for total sum of all items.

When we understand ROLLUP, CUBE is easy as it simply extends functionality. The CUBE operator is similar to the ROLLUP operator but adds summary rows for all possible column combinations in the GROUP BY clause.

Let’s examine following example:

Now we have extra summary rows. In addition to the summary rows from ROLLUP, the CUBE operator extends the output to include summary rows for all possible combinations of groups in the GROUP BY clause. The result set now shows a total sum of items for each location seen in the query below.

```
SELECT ProductID, LocationID, SUM (Quantity) AS TotalQuantity
FROM Production.ProductInventory
WHERE ProductID IN (1,2)
GROUP BY ProductID, LocationID WITH CUBE
ORDER BY ProductID, LocationID
```

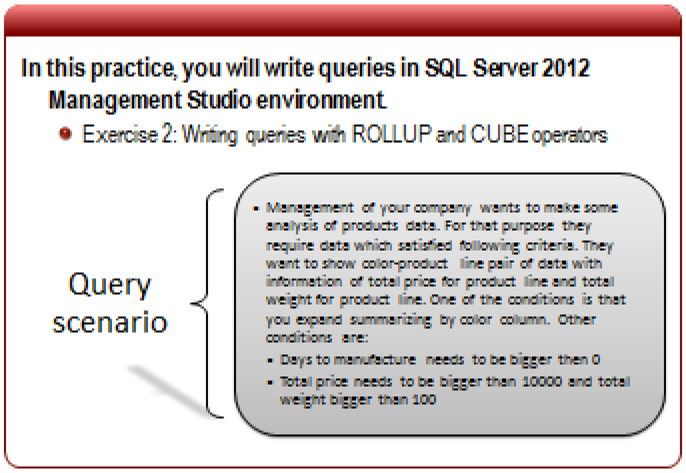
Result:

```
ProductID   LocationID TotalQuantity
----------- ---------- -------------
NULL        NULL       2194
NULL        1          835
NULL        6          642
NULL        50         717
1           NULL       1085
1           1          408
1           6          324
1           50         353
2           NULL       1109
2           1          427
2           6          318
2           50         364

(12 row(s) affected)
```

Keep the following in mind when working with ROLLUP and CUBE operators:

- The ROLLUP operator adds extra summary rows to the result set by grouping rows based on the columns named in the GROUP BY clause.
- If you have *n* columns or expressions in the GROUP BY clause, SQL Server returns $2^n - 1$ possible combination in the result set.
- The aggregate values in extra rows in the result set are subtotals for the groups represented by the leftmost columns in the GROUP BY clause.
- When using the ROLLUP and CUBE operators, ensure that the columns in GROUP BY clause are meaningful in your business environment.

Practice: Summarizing Grouped Data



In this practice, you will write queries in SQL Server 2012 Management Studio environment

- Exercise 2: Writing queries with ROLLUP and CUBE operators

Query scenario

- Management of your company wants to make some analysis of products data. For that purpose they require data which satisfied following criteria. They want to show color-product line pair of data with information of total price for product line and total weight for product line. One of the conditions is that you expand summarizing by color column. Other conditions are:
  - Days to manufacture needs to be bigger then 0
  - Total price needs to be bigger than 10000 and total weight bigger than 100

In this practice you will write queries in SQL Server 2012 Management Studio environment.

To successfully complete exercise you need following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
  - http://msftdbprodsamples.codeplex.com/

Exercise: Writing a query with summarizing within result sets

1. Click **Start→Microsoft SQL Server 2012→SQL Server Management Studio**
2. In the **Connect to Server** dialog box, under **Server name** type or change the name of your local server instance to (*local*)
3. Use Windows Authentication
4. **Open File** menu **→New →Database Engine Query**

Query Scenario

The Management team wants to perform some product data analysis and has asked for a report using the following criteria:

Show color-product line pair of data with information of total price for product line and total weight for product line. One of the conditions is that you expand summarizing by color column. Other conditions are:

5. Days to manufacture needs to be bigger then 0
6. Total price needs to be bigger than 10000 and total weight bigger than 100

7. Type following T-SQL code in query window.

```
        SELECT Color, ProductLine,
                  SUM (ListPrice) AS TotalPrice,
                  SUM (Weight) AS TotalWeight
        FROM Production.Product
        WHERE DaysToManufacture > 0
        GROUP BY Color, ProductLine WITH ROLLUP
        HAVING SUM (Listprice) >10000 and SUM (Weight)>100
        ORDER BY Color DESC, ProductLine
```

8. Click on **Execute** or press F5 to run.

Result:

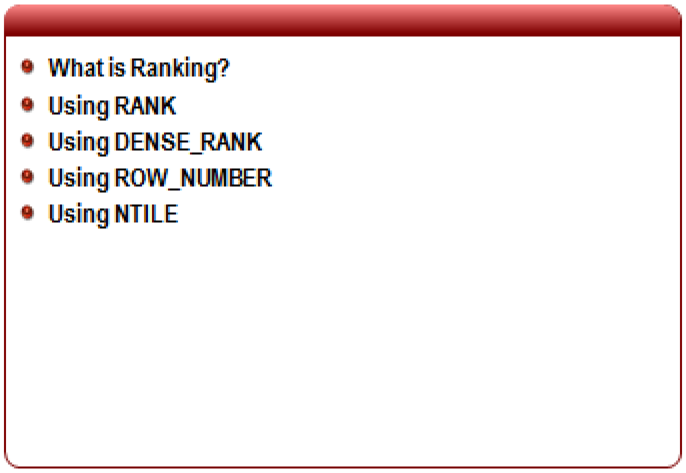
```
Color           ProductLine TotalPrice            TotalWeight
--------------- ----------- --------------------- -----------------------
Yellow          NULL        34311,33              442.26
Yellow          R           15380,56              165.32
Yellow          T           18930,77              276.94
Silver          NULL        36508,14              1379.70
Silver          M           36061,95              442.70
Red             NULL        53239,11              376.43
Red             R           53239,11              376.43
Blue            NULL        23790,17              387.44
Blue            T           23790,17              387.44
Black           NULL        66538,93              7873.36
Black           M           38017,80              438.70
Black           R           27221,14              5624.66
NULL            NULL        218306,97             12144.19
Warning: Null value is eliminated by an aggregate or other SET operation.

(13 row(s) affected)
```

9. Type same query with CUBE operator and run it.

```
        SELECT Color, ProductLine,
                  SUM (ListPrice) AS TotalPrice,
                  SUM (Weight) AS TotalWeight
        FROM Production.Product
        WHERE DaysToManufacture > 0
        GROUP BY Color, ProductLine WITH CUBE
        HAVING SUM (Listprice) >10000 and SUM (Weight)>100
        ORDER BY Color DESC, ProductLine
```

10. Analyze and compare the two result sets.

Lesson 3: Ranking Grouped Data



- What is Ranking?
- Using RANK
- Using DENSE_RANK
- Using ROW_NUMBER
- Using NTILE

The Ranking function returns a ranking value for each row in a result set. Depending on the function that is used, some rows could receive the same value as other rows.

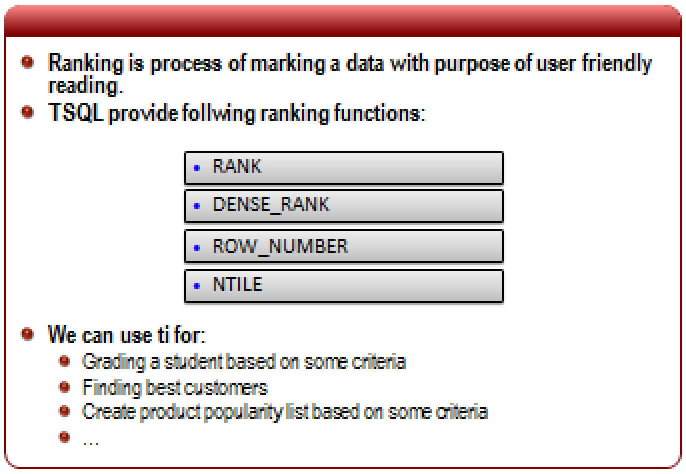
In this lesson you will learn how to use T-SQL ranking function for the purpose of numbering ranking your data in result sets. You will also learn about other types of ranking functions.

Objectives

After completing this lesson, you will be able to:

- Understand and use the RANK functions
- Understand and use the DENSE_RANK function
- Understand and use the ROW_NUMBER function
- Understand and use the NTILE function

What is Ranking?



- Ranking is process of marking a data with purpose of user friendly reading.
- TSQL provide follwing ranking functions:
  - RANK
  - DENSE_RANK
  - ROW_NUMBER
  - NTILE
- We can use ti for:
  - Grading a student based on some criteria
  - Finding best customers
  - Create product popularity list based on some criteria
  - …

Ranking is the process of organizing data for the purpose of user friendly reading. SQL Server 2012 and T-SQL provides the following ranking functions:

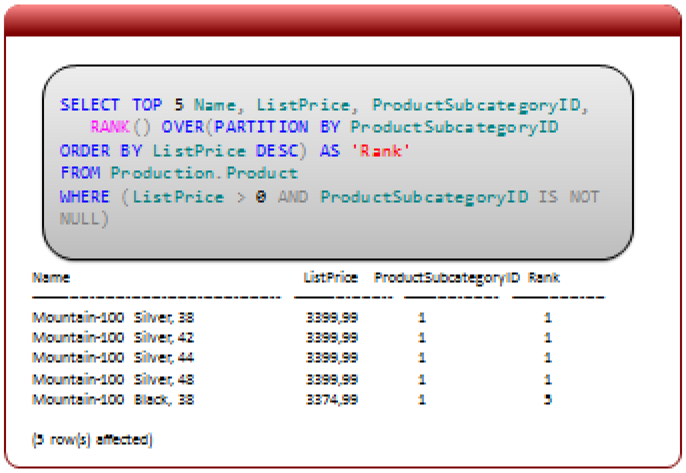
Function
RANK
DENSE_RANK
ROW_NUMBER
NTILE

Some examples of when rank functions are useful are:

- Grading a student based on some criteria
- Finding the top 10 customers
- Create product popularity list based on some criteria
- etc.

The four ranking functions provide different ranking outputs.

Using RANK



```
SELECT TOP 5 Name, ListPrice, ProductSubcategoryID,
   RANK() OVER(PARTITION BY ProductSubcategoryID
ORDER BY ListPrice DESC) AS 'Rank'
FROM Production.Product
WHERE (ListPrice > 0 AND ProductSubcategoryID IS NOT
NULL)
```

```
Name                                ListPrice  ProductSubcategoryID  Rank
----------------------------------  ---------  --------------------  ----------
Mountain-100 Silver, 38             3399,99    1                     1
Mountain-100 Silver, 42             3399,99    1                     1
Mountain-100 Silver, 44             3399,99    1                     1
Mountain-100 Silver, 48             3399,99    1                     1
Mountain-100 Black, 38              3374,99    1                     5

(5 row(s) affected)
```

RANK is the most basic of the Rank functions. It returns a rank of each row within the partition of a result set.

The PARTITION BY clause divides a group into sub groups result set based on returned data in the column. (E.g. `PARTITION BY ProductSubcategoryID.`

```
RANK ( ) OVER ( [ partition_by_clause ] order_by_clause )
```

In this case, partitions are formed based on ProductSubcategoryID. If PARTITION BY is not specified, the function will return all rows as a single group and treats them as such. The ORDER BY clause creates an order before PARTITION BY is applied on result sets.

The query below instructs the database to fetch and rank the top 10 products partitioned by ProductSubcategoryID and List price columns used to rank the result set.

```
SELECT TOP 10 Name, ListPrice,
       ProductSubcategoryID,
RANK() OVER(PARTITION BY ProductSubcategoryID
ORDER BY ListPrice DESC) AS 'Rank'
FROM Production.Product
WHERE (ListPrice > 0 AND ProductSubcategoryID IS NOT NULL)
```

Note: WHERE (ListPrice > 0 AND ProductSubcategoryID IS NOT NULL) – excludes NULLs

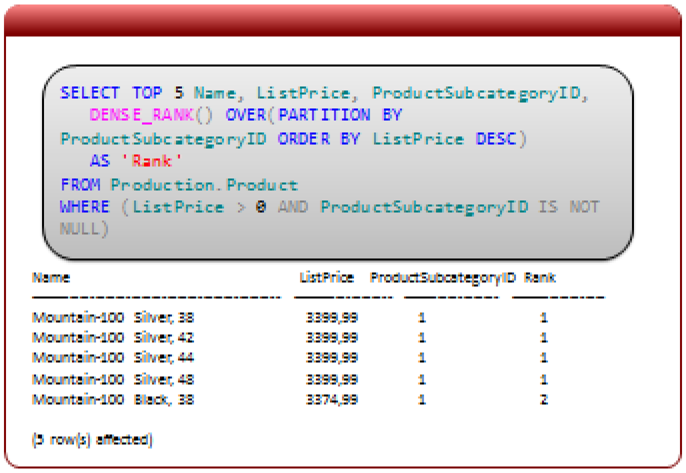
Result

```
Name                             ListPrice      ProductSubcategoryID Rank
------------------------------ ------------- -------------------- -------
Mountain-100 Silver, 38         3399,99         1                    1
Mountain-100 Silver, 42         3399,99         1                    1
Mountain-100 Silver, 44         3399,99         1                    1
Mountain-100 Silver, 48         3399,99         1                    1
Mountain-100 Black,  38         3374,99         1                    5
Mountain-100 Black,  42         3374,99         1                    5
Mountain-100 Black,  44         3374,99         1                    5
Mountain-100 Black,  48         3374,99         1                    5
Mountain-200 Silver, 38         2319,99         1                    9
Mountain-200 Silver, 42         2319,99         1                    9

(10 row(s) affected)
```

Things to remember when working with the RANK function:

- Use ORDER BY in the RANK clause to order the RANK similar to TOP and ORDER BY clauses.
- If two or more rows tie for in rank, each tied row receives the same rank.
- Rank function by itself can result in non-consecutive rankings.
- RANK function creates gaps in ranking

Using DENSE_RANK



```
SELECT TOP 5 Name, ListPrice, ProductSubcategoryID,
   DENSE_RANK() OVER(PARTITION BY
ProductSubcategoryID ORDER BY ListPrice DESC)
   AS 'Rank'
FROM Production.Product
WHERE (ListPrice > 0 AND ProductSubcategoryID IS NOT
NULL)
```

```
Name                                    ListPrice    ProductSubcategoryID Rank
--------------------------------------- ------------ -------------------- -------------

Mountain-100 Silver, 38                 3399,99      1                    1
Mountain-100 Silver, 42                 3399,99      1                    1
Mountain-100 Silver, 44                 3399,99      1                    1
Mountain-100 Silver, 48                 3399,99      1                    1
Mountain-100 Black, 38                  3374,99      1                    2

(5 row(s) affected)
```

DENSE_RANK returns the rank of rows within the partition of a result set, without any gaps in the ranking.

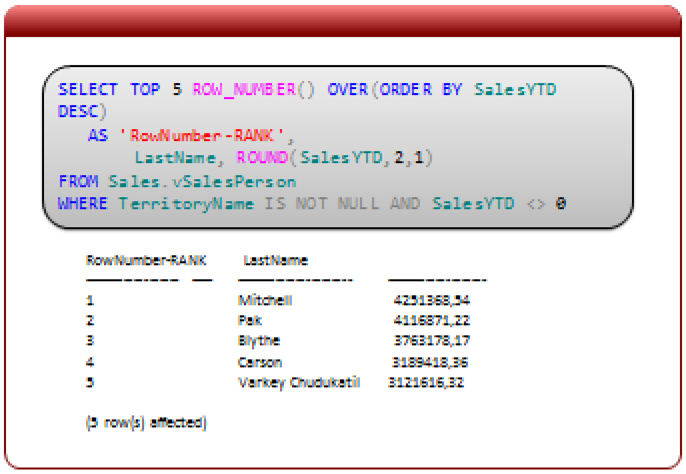
In the example below the Rank column is now ordered by rank and then by numeric order.

```
      SELECT TOP 10 Name, ListPrice,
             ProductSubcategoryID,
      DENSE_RANK() OVER(PARTITION BY ProductSubcategoryID
      ORDER BY ListPrice DESC) AS 'Rank'
      FROM Production.Product
      WHERE (ListPrice > 0 AND ProductSubcategoryID IS NOT NULL)
```

Result

```
Name                           ListPrice     ProductSubcategoryID Rank
------------------------------ ------------- -------------------- -------
Mountain-100 Silver, 38        3399,99         1                    1
Mountain-100 Silver, 42        3399,99         1                    1
Mountain-100 Silver, 44        3399,99         1                    1
Mountain-100 Silver, 48        3399,99         1                    1
Mountain-100 Black,  38        3374,99         1                    2
Mountain-100 Black,  42        3374,99         1                    2
Mountain-100 Black,  44        3374,99         1                    2
Mountain-100 Black,  48        3374,99         1                    2
Mountain-200 Silver, 38        2319,99         1                    3
Mountain-200 Silver, 42        2319,99         1                    3

(10 row(s) affected)
```

Using ROW_NUMBER



```
SELECT TOP 5 ROW_NUMBER() OVER(ORDER BY SalesYTD
DESC)
   AS 'RowNumber-RANK',
       LastName, ROUND(SalesYTD,2,1)
FROM Sales.vSalesPerson
WHERE TerritoryName IS NOT NULL AND SalesYTD <> 0
```

```
RowNumber-RANK      LastName
---------- --       -------------    ----------
1                   Mitchell          4251368,54
2                   Pak               4116871,22
3                   Blythe            3763178,17
4                   Carson            3189418,36
5                   Varkey Chudukatil 3121616,32

(5 row(s) affected)
```

The ROW_NUMBER function can also be used for ranking. Starting at 1, it returns sequential numbering of rows for each partition of a result set.

The example below calculates a row number for the sales people based on their year-to-date sales ranking in reverse order.

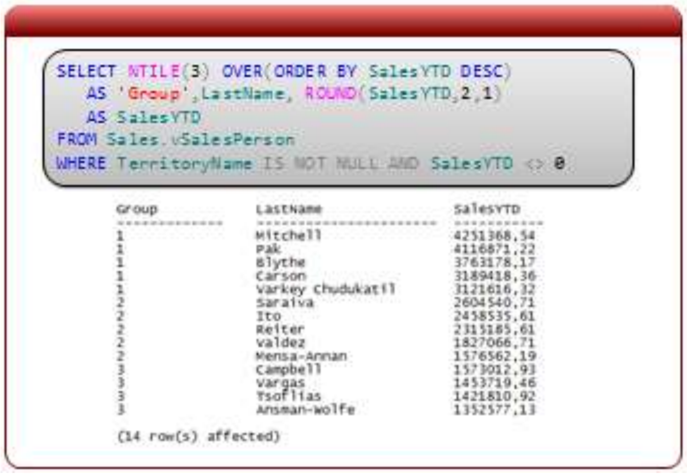
```
SELECT TOP 10 ROW_NUMBER() OVER(ORDER BY SalesYTD DESC)
         AS 'RowNumber-RANK',
       FirstName, LastName, ROUND(SalesYTD,2,1)
FROM Sales.vSalesPerson
WHERE TerritoryName IS NOT NULL AND SalesYTD <> 0
```

Result:

```
RowNumber-RANK   FirstName      LastName             SalesYTD
---------------- -------------- -------------------- --------------
1                    Linda      Mitchell              4251368,54
2                    Jae        Pak                   4116871,22
3                    Michael    Blythe                3763178,17
4                    Jillian    Carson                3189418,36
5                    Ranjit     Varkey Chudukatil     3121616,32
6                    José       Saraiva               2604540,71
7                    Shu        Ito                   2458535,61
8                    Tsvi       Reiter                2315185,61
9                    Rachel     Valdez                1827066,71
10                   Tete       Mensa-Annan           1576562,19

(10 row(s) affected)
```

This function can be used with or without PARTITION BY clause.

Using NTILE



```
SELECT NTILE(3) OVER(ORDER BY SalesYTD DESC)
   AS 'Group',LastName, ROUND(SalesYTD,2,1)
   AS SalesYTD
FROM Sales.vSalesPerson
WHERE TerritoryName IS NOT NULL AND SalesYTD <> 0
```

```
Group          LastName                  SalesYTD
-------------- ------------------------- -----------
1              Mitchell                  4251368,54
1              Pak                       4116871,22
1              Blythe                    3763178,17
1              Carson                    3189418,36
1              Varkey Chudukatil         3121616,32
2              Saraiva                   2604540,71
2              Ito                       2458535,61
2              Reiter                    2315185,61
2              Valdez                    1827066,71
2              Mensa-Annan               1576562,19
3              Campbell                  1573012,93
3              Vargas                    1453719,46
3              Tsoflias                  1421810,92
3              Ansman-Wolfe              1352577,13

(14 row(s) affected)
```

The NTILE function distributes rows in an ordered partition into a specified number of groups. The groups are numbered, starting at one. For each row, NTILE returns the number of the group to which the row belongs.

Ranking data can be sorted into specified groups and the numbers of groups are dependent on NTILE parameters and numbers of rows returned in result set. NTILE is most commonly used in data warehouse environments and less in classic OLTP systems.

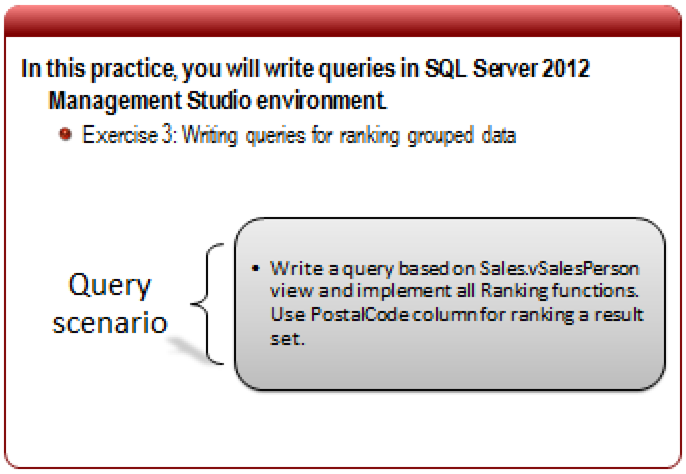
The query below orders sales people into numerical ascending groups and then orders by SalesYTD within the groups in descending order.

```
SELECT NTILE(3) OVER(ORDER BY SalesYTD DESC)
                  AS 'Group',LastName, ROUND(SalesYTD,2,1)
                  AS SalesYTD
      FROM Sales.vSalesPerson
      WHERE TerritoryName IS NOT NULL AND SalesYTD <> 0
```

Result:

```
Group                LastName                                             SalesYTD
-------------------- ---------------------------------------------------- -----------
1                    Mitchell                                             4251368,54
1                    Pak                                                  4116871,22
1                    Blythe                                               3763178,17
1                    Carson                                               3189418,36
1                    Varkey Chudukatil                                    3121616,32
2                    Saraiva                                              2604540,71
2                    Ito                                                  2458535,61
2                    Reiter                                               2315185,61
2                    Valdez                                               1827066,71
2                    Mensa-Annan                                          1576562,19
3                    Campbell                                             1573012,93
3                    Vargas                                               1453719,46
3                    Tsoflias                                             1421810,92
3                    Ansman-Wolfe                                         1352577,13

(14 row(s) affected)
```

Practice: Ranking Grouped Data



In this practice, you will write queries in SQL Server 2012 Management Studio environment

- Exercise 3: Writing queries for ranking grouped data

Query scenario

- Write a query based on Sales.vSalesPerson view and implement all Ranking functions. Use PostalCode column for ranking a result set.

In this practice you will write queries in SQL Server 2012 Management Studio environment. Practice is based on Lesson 3.

To successfully complete exercise you need following resources:

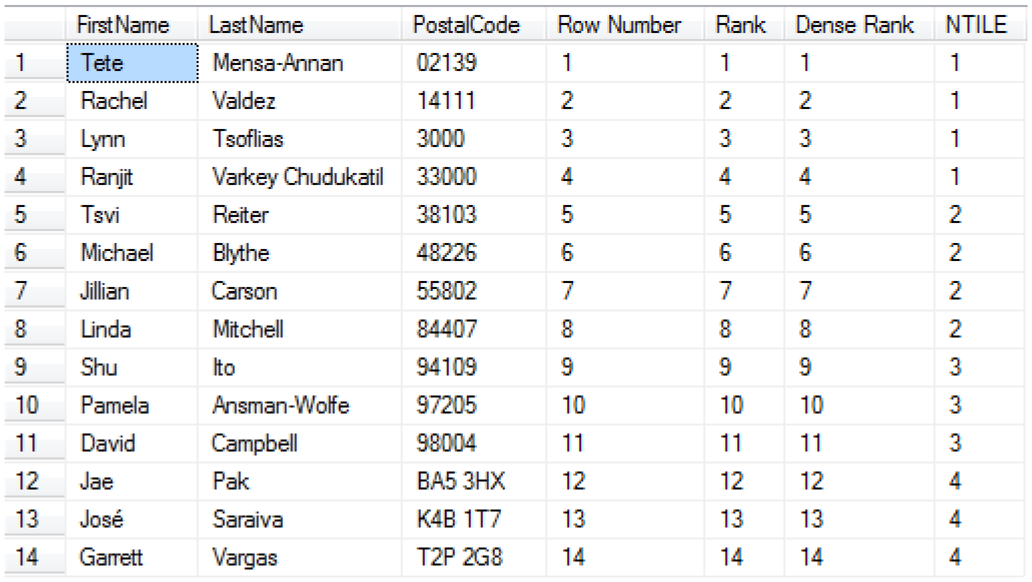
- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
  - http://msftdbprodsamples.codeplex.com/

Exercise: Writing Queries with Ranking Grouped Data

1. Click **Start→Microsoft SQL Server 2012→SQL Server Management Studio**
2. In the **Connect to Server** dialog box, under **Server name** type or change the name of your local server instance to (*local*).
3. Use Windows Authentication.
4. On **File** menu **→New →Database Engine Query**
5. Write a query based on Sales.vSalesPerson view and implement all Ranking functions. Use PostalCode column for ranking the result set.

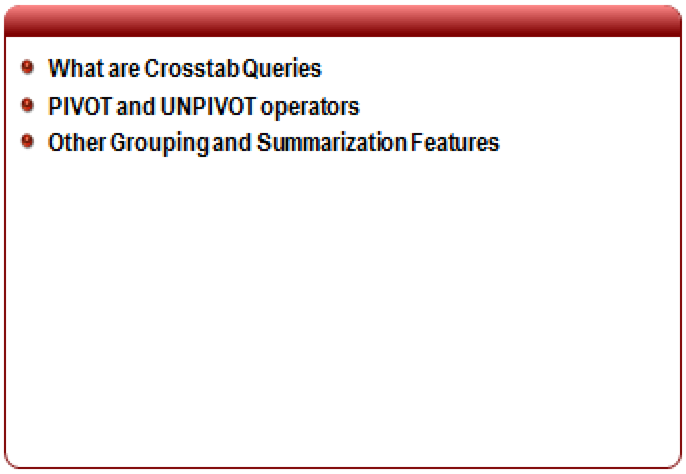
```
SELECT FirstName, LastName,PostalCode
      ,ROW_NUMBER() OVER (ORDER BY PostalCode) AS 'Row Number'
      ,RANK() OVER (ORDER BY PostalCode) AS 'Rank'
      ,DENSE_RANK() OVER (ORDER BY PostalCode) AS 'Dense Rank'
      ,NTILE(4) OVER (ORDER BY PostalCode) AS 'NTILE'
FROM Sales.vSalesPerson
WHERE TerritoryName IS NOT NULL AND SalesYTD <> 0
```

Result:



	FirstName	LastName	PostalCode	Row Number	Rank	Dense Rank	NTILE
1	Tete	Mensa-Annan	02139	1	1	1	1
2	Rachel	Valdez	14111	2	2	2	1
3	Lynn	Tsoflias	3000	3	3	3	1
4	Ranjit	Varkey Chudukatil	33000	4	4	4	1
5	Tsvi	Reiter	38103	5	5	5	2
6	Michael	Blythe	48226	6	6	6	2
7	Jillian	Carson	55802	7	7	7	2
8	Linda	Mitchell	84407	8	8	8	2
9	Shu	Ito	94109	9	9	9	3
10	Pamela	Ansman-Wolfe	97205	10	10	10	3
11	David	Campbell	98004	11	11	11	3
12	Jae	Pak	BA5 3HX	12	12	12	4
13	José	Saraiva	K4B 1T7	13	13	13	4
14	Garrett	Vargas	T2P 2G8	14	14	14	4

Lesson 4: Creating Crosstab Queries



- What are Crosstab Queries
- PIVOT and UNPIVOT operators
- Other Grouping and Summarization Features

Crosstab queries restructure data to make it easier to read and understand. It uses two sets of values, one down the side of the data sheet and one across the top and calculates a sum, average or other aggregate function and then groups the results.

In this lesson you will learn how to write crosstab operator queries. You can find the same functionality in Microsoft Excel product. In this case you can do exactly the same thing but in database environment.

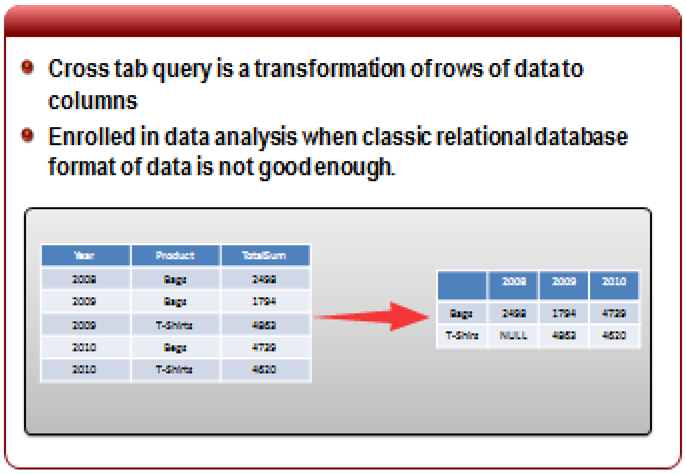
Crosstab is usually used for data analysis when the classic relational database format of data is not good enough.

Objectives

After completing this lesson, you will be able to:

- Understand crosstab queries
- Use PIVOT and UNPIVOT operators
- Explore other Grouping and Summarization Features

What are Crosstab Queries?



- Cross tab query is a transformation of rows of data to columns
- Enrolled in data analysis when classic relational database format of data is not good enough.

Year	Product	TotalSum
2008	Bags	2498
2009	Bags	1794
2009	T-Shirts	4863
2010	Bags	4739
2010	T-Shirts	4620

	2008	2009	2010
Bags	2498	1794	4739
T-Shirts	NULL	4863	4620

Crosstab queries calculate a sum, average, count, or other type of total for data that is grouped by two types of information – one down the left side of the datasheet and another across the top.

Crosstab queries are useful for summarizing information, calculating statistics, spotting bad data and looking for trends. The results of a crosstab query are read-only — data cannot be added, edited or deleted in a crosstab result. An aggregate function, such as sum or count, is used to help summarize the data.

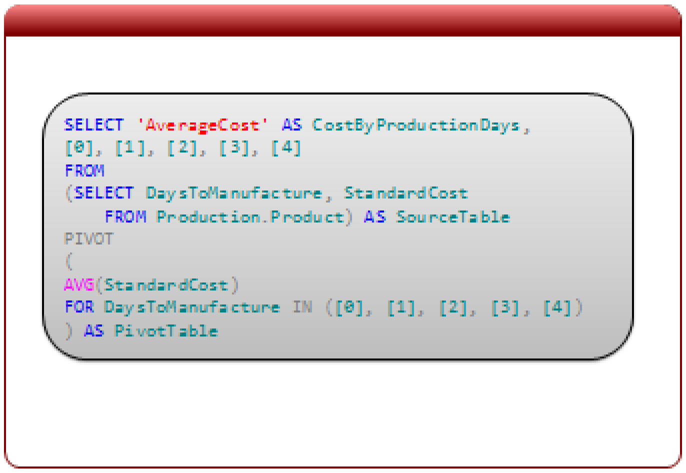
SQL Server 2005 and later comes with PIVOT and UNPIVOT predicates that introduce a new way of writing crosstab queries.

The tables below are examples a crosstab result sets.

Year	Product	TotalSum
2008	Bags	2498
2009	Bags	1794
2009	T-Shirts	4863
2010	Bags	4739
2010	T-Shirts	4620

	2008	2009	2010
Bags	2498	1794	4739
T-Shirts	NULL	4863	4620

PIVOT and UNPIVOT operators



```
SELECT 'AverageCost' AS CostByProductionDays,
[0], [1], [2], [3], [4]
FROM
(SELECT DaysToManufacture, StandardCost
    FROM Production.Product) AS SourceTable
PIVOT
(
AVG(StandardCost)
FOR DaysToManufacture IN ([0], [1], [2], [3], [4])
) AS PivotTable
```

The PIVOT operator rotates a table-valued expression by turning the unique values from one column in the expression into multiple columns in the output. You can also perform aggregations where they are required on any remaining columns. UNPIVOT performs the opposite operation to PIVOT by rotating columns of a table-valued expression into column values.

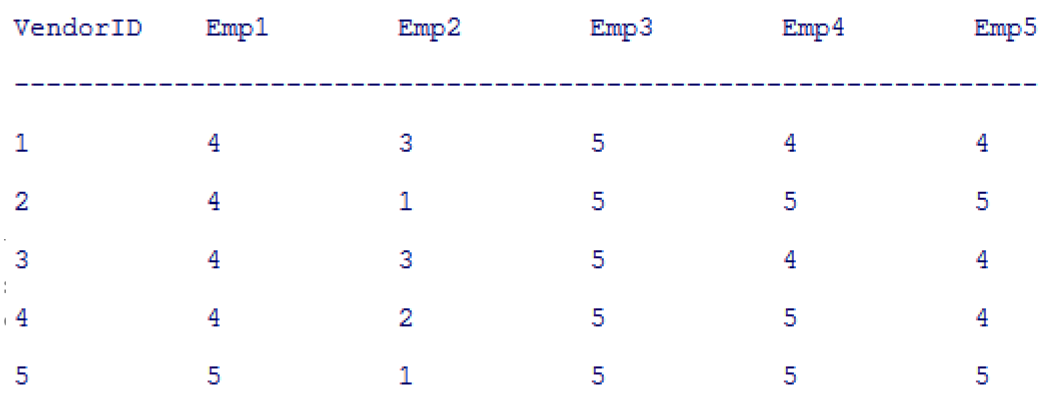
The query below uses a PIVOT operator so that the DaysToManufacture values become the column headings. A column is provided for three days, even though the results are NULL.

```
SELECT 'AvgCost' AS CostByProductionDays,
[0], [1], [2], [3], [4]
FROM
(SELECT DaysToManufacture, StandardCost
    FROM Production.Product) AS SourceTable
PIVOT
(
AVG(StandardCost)
FOR DaysToManufacture IN ([0], [1], [2], [3], [4])
) AS PivotTable;
```

Result:

```
CostByProductionDays 0          1            2           3        4
-------------------- ---------- ------------ ----------- -------- ---------
AverageCost          5,0885     223,88       359,1082    NULL     949,4105

(1 row(s) affected)
```

Let’s assume that we have a table in following format and data and now want to use UNPIVOT reverse what PIVOT operator created.

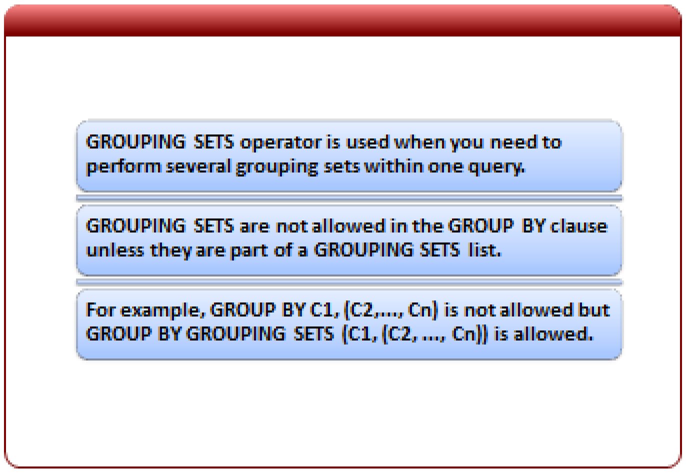


```
VendorID    Emp1        Emp2        Emp3        Emp4        Emp5
----------------------------------------------------------------
1           4           3           5           4           4
2           4           1           5           5           5
3           4           3           5           4           4
4           4           2           5           5           4
5           5           1           5           5           5
```

```
      SELECT VendorID, Employee, Orders
        FROM
            (SELECT VendorID, Emp1, Emp2, Emp3, Emp4, Emp5
            FROM dbo.pvt) AS p
        UNPIVOT
            (Orders FOR Employee IN
                (Emp1, Emp2, Emp3, Emp4, Emp5)
        )AS unpvt
```

UNPIVOT is not a 100% reverse operation of PIVOT. PIVOT performs an aggregation and therefore, merges possible multiple rows into a single row in the output. UNPIVOT does not reproduce the original table-valued expression result because rows have been merged.

Other Grouping and Summarization Features



GROUPING SETS operator is used when you need to perform several grouping sets within one query.

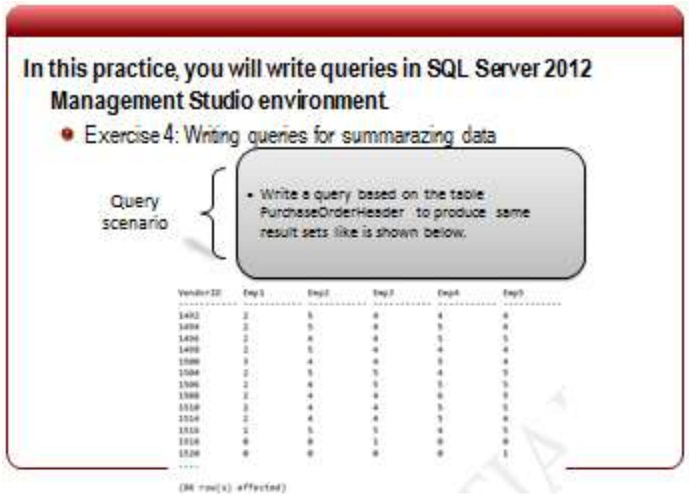
GROUPING SETS are not allowed in the GROUP BY clause unless they are part of a GROUPING SETS list.

For example, GROUP BY C1, (C2,..., Cn) is not allowed but GROUP BY GROUPING SETS (C1, (C2, ..., Cn)) is allowed.

The GROUPING SETS operator is used when you need to perform several grouping sets within a single query. So far we have achieved this by using GROUP BY in combination with UNION ALL.

GROUPING SETS are not allowed in a GROUP BY clause unless they are part of a GROUPING SETS list. For example, GROUP BY C1, (C2,..., Cn) is not allowed but GROUP BY GROUPING SETS (C1, (C2, ..., Cn)) is allowed.

Practice: Creating Crosstab Queries



In this practice, you will write queries in SQL Server 2012 Management Studio environment.

- Exercise 4: Writing queries for summarazing data

- Write a query based on the table PurchaseOrderHeader to produce same result sets like is shown below.

```
VendorID  Emp1  Emp2  Emp3  Emp4  Emp5
--------  ----  ----  ----  ----  ----
1492      2     5     6     4     6
1494      2     5     6     5     6
1496      2     4     6     5     5
1498      2     5     4     4     4
1500      3     4     6     5     4
1504      2     5     5     4     5
1506      2     4     5     5     5
1508      2     4     6     6     5
1510      2     4     4     5     5
1514      2     4     4     5     6
1516      1     5     5     4     5
1518      0     0     1     0     0
1520      0     0     0     0     1
----

(86 row(s) affected)
```

In this practice you will write queries in SQL Server 2012 Management Studio environment. Practice is based on Lesson 4.

To successfully complete exercise you need following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
  - http://msftdbprodsamples.codeplex.com/

Exercise: Writing a query with ranking grouped data

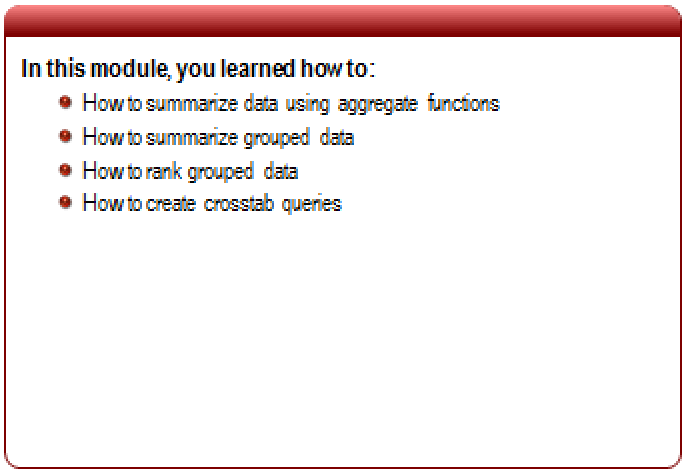
1. Click **Start→Microsoft SQL Server 2012→SQL Server Management Studio**
2. In the **Connect to Server** dialog box, under **Server name** type or change the name of your local server instance to (*local)*.
3. Use Windows Authentication
4. From the **File** menu **→New →Database Engine Query**
5. Query Scenario
6. Write a query based on the table **PurchaseOrderHeader** . The query is based on employees with ID numbers: 250,251,256,257 and 260.

```
      SELECT VendorID, [250] AS Emp1, [251] AS Emp2, [256] AS Emp3,
[257] AS Emp4, [260] AS Emp5
      FROM
      (SELECT PurchaseOrderID, EmployeeID, VendorID
      FROM Purchasing.PurchaseOrderHeader) p
      PIVOT
      (
      COUNT (PurchaseOrderID)
      FOR EmployeeID IN
      ( [250], [251], [256], [257], [260] )
      ) AS pvt
      ORDER BY pvt.VendorID;
```

Result:

```
VendorID    Emp1        Emp2        Emp3        Emp4        Emp5
----------- ----------- ----------- ----------- ----------- -----------
1492        2           5           4           4           4
1494        2           5           4           5           4
1496        2           4           4           5           5
1498        2           5           4           4           4
1500        3           4           4           5           4
1504        2           5           5           4           5
1506        2           4           5           5           5
1508        2           4           4           6           5
1510        2           4           4           5           5
...

1694        2           5           4           5           4
1696        2           5           4           4           4
1698        1           5           5           4           5

(86 row(s) affected)
```

Summary


In this module, you learned how to:

- How to summarize data using aggregate functions
- How to summarize grouped data
- How to rank grouped data
- How to create crosstab queries

In this module you have learned that aggregation is a very important aspect of T-SQL querying. Through the four lessons you learned how to write basic and more complex queries for grouping and summarizing data.

You have enough skills to recognize places in data. You have also learned where and when to use aggregations based on AVG, SUM, COUNT and other set based functions. You have an understanding of how to use the GROUP BY clause and filtering result sets using the HAVING clause. Lastly, you learned how to use data ranking and crosstab techniques.

Objectives

After completing this module, you learned:

- How to summarize data using aggregate functions
- How to summarize grouped data
- How to rank grouped data
- How to create crosstab queries