

# Data Management Using Microsoft SQL Server

Session: 9

Advanced Queries and Joins

For Aptech Centre Use Only

# Objectives

- Explain grouping and aggregating data
- Describe subqueries
- Describe table expressions
- Explain joins
- Describe various types of joins
- Explain the use of various set operators to combine data
- Describe pivoting and grouping set operations

# Introduction

- SQL Server 2012 includes several powerful query features that help you to retrieve data efficiently and quickly.
- Data can be grouped and/or aggregated together in order to present summarized information.
- Using the concept of subqueries, a resultset of a `SELECT` can be used as criteria for another `SELECT` statement or query.
- Joins help you to combine column data from two or more tables based on a logical relationship between the tables.
- Set operators such as `UNION` and `INTERSECT` help you to combine row data from two or more tables.
- The `PIVOT` and `UNPIVOT` operators are used to transform the orientation of data from column-oriented to row-oriented and vice versa.
- The `GROUPING SET` subclause of the `GROUP BY` clause helps to specify multiple groupings in a single query.

# Grouping Data 1-3

The **GROUP BY** clause partitions the resultset into one or more subsets. Each subset has values and expressions in common.

The **GROUP BY** keyword is followed by a list of columns, known as grouped columns. Every grouped column restricts the number of rows of the resultset.

For every grouped column, there is only one row. The **GROUP BY** clause can have more than one grouped column.

➤ The syntax for **GROUP BY** clause is as follows:

## Syntax:

```
CREATE TYPE [ schema_name. ] type_name { FROM base_type [ ( precision [ ,  
scale ] ) ] [ NULL | NOT NULL ] } [ ; ]
```

where,

**column\_name**: is the name of the column according to which the resultset should be grouped.

## Grouping Data 2-3

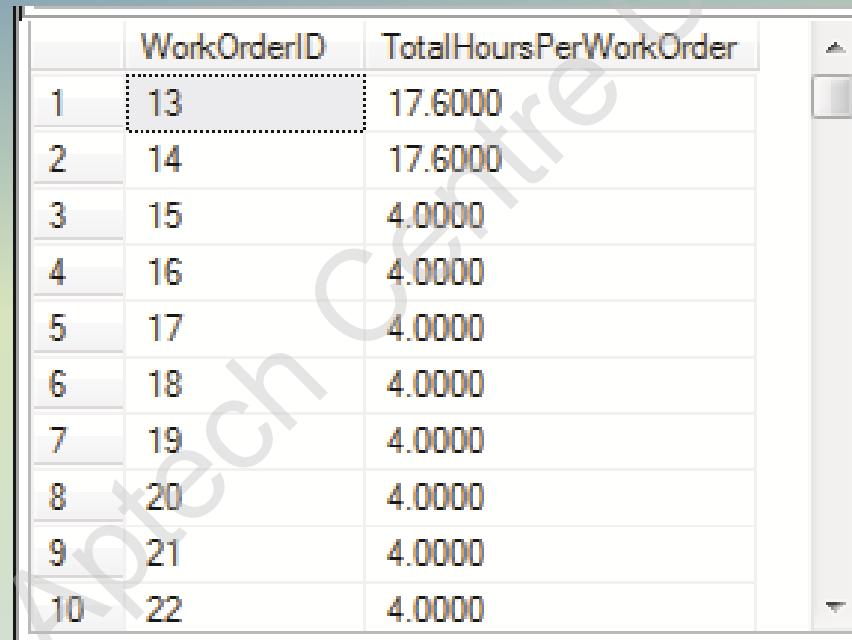
- Consider the `WorkOrderRouting` table in the AdventureWorks2012 database.
- The total resource hours per work order needs to be calculated.
- To achieve this, the records need to be grouped by work order number, that is, `WorkOrderID`.
- Following code snippet retrieves and displays the total resource hours per work order along with the work order number:

```
SELECT WorkOrderID, SUM(ActualResourceHrs) AS TotalHoursPerWorkOrder  
FROM Production.WorkOrderRouting GROUP BY WorkOrderID
```

- In this query, a built-in function named `SUM ( )` is used to calculate the total.
- `SUM ( )` is an aggregate function.
- Aggregate functions will be covered in detail in a later section.

## Grouping Data 3-3

- Executing this query will return all the work order numbers along with the total number of resource hours per work order.
- A part of the output is shown in the following figure:



A screenshot of a SQL query result grid. The grid has two columns: 'WorkOrderID' and 'TotalHoursPerWorkOrder'. The first column contains integers from 1 to 10, and the second column contains decimal values. The first row (1, 13, 17.6000) is highlighted with a dotted border. A vertical scrollbar is visible on the right side of the grid.

	WorkOrderID	TotalHoursPerWorkOrder
1	13	17.6000
2	14	17.6000
3	15	4.0000
4	16	4.0000
5	17	4.0000
6	18	4.0000
7	19	4.0000
8	20	4.0000
9	21	4.0000
10	22	4.0000

# GROUP BY with WHERE 1-2

The WHERE clause can also be used with GROUP BY clause to restrict the rows for grouping.

The rows that satisfy the search condition are considered for grouping.

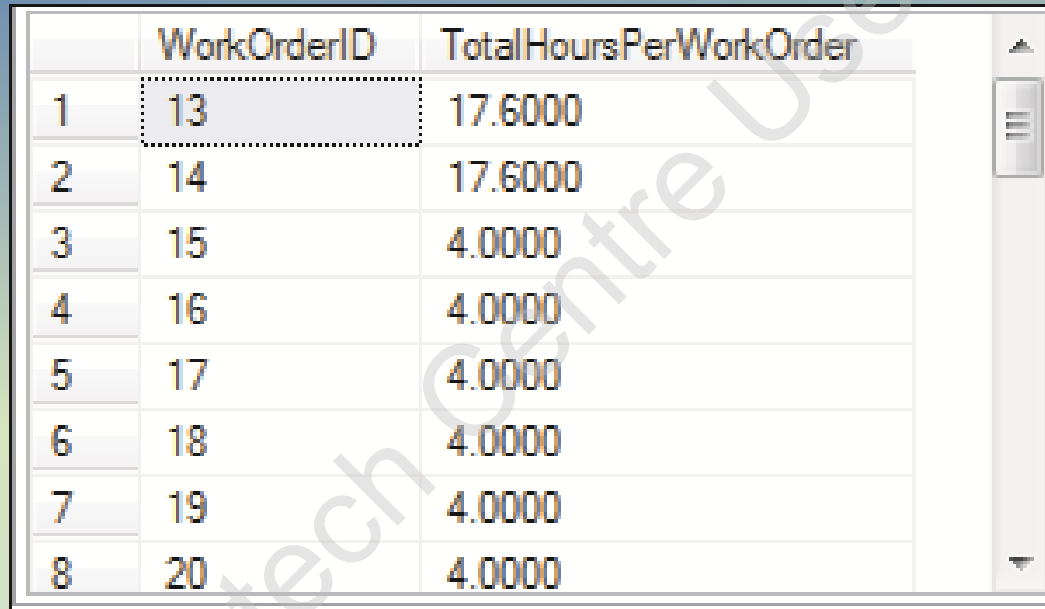
The rows that do not meet the conditions in the WHERE clause are eliminated before any grouping is done.

- Following code snippet shows a query that limits the rows displayed, by considering only those records with WorkOrderID less than 50:

```
SELECT WorkOrderID, SUM(ActualResourceHrs) AS TotalHoursPerWorkOrder  
FROM Production.WorkOrderRouting WHERE WorkOrderID <50 GROUP BY  
WorkOrderID
```

## GROUP BY with WHERE 2-2

- As the number of records returned is more than 25, a part of the output is shown in the following figure:



	WorkOrderID	TotalHoursPerWorkOrder
1	13	17.6000
2	14	17.6000
3	15	4.0000
4	16	4.0000
5	17	4.0000
6	18	4.0000
7	19	4.0000
8	20	4.0000



# GROUP BY with NULL

If the grouping column contains a NULL value, that row becomes a separate group in the resultset.

If the grouping column contains more than one NULL value, the NULL values are put into a single row.

Consider the `Production.Product` table. There are some rows in it that have NULL values in the `Class` column.

Using a `GROUP BY` on a query for this table will take into consideration the NULL values too.

- Following code snippet retrieves and displays the average of the list price for each class:

```
SELECT Class, AVG (ListPrice) AS 'AverageListPrice' FROM  
Production.Product GROUP BY Class
```

- As shown in the following figure, the NULL values are grouped into a single row in the output:

	Class	AverageListPrice
1	NULL	16.314
2	H	1679.4964
3	L	370.6887
4	M	635.5816

# GROUP BY with ALL 1-2

The ALL keyword can also be used with the GROUP BY clause. It is significant only when the SELECT has a WHERE clause.

When ALL is used, it includes all the groups that the GROUP BY clause produces.

It even includes those groups which do not meet the search conditions.

- The syntax of using GROUP BY with ALL is as follows:

## Syntax:

```
SELECT <column_name> FROM <table_name> WHERE <condition> GROUP BY ALL  
<column _name>
```

# GROUP BY with ALL 2-2

- Consider the Sales.SalesTerritory table.
- This table has a column named Group indicating the geographic area to which the sales territory belongs to.
- Following code snippet calculates and displays the total sales for each group:

```
SELECT [Group],SUM(SalesYTD) AS 'TotalSales' FROM Sales.SalesTerritory  
WHERE [Group] LIKE 'N%' OR [Group] LIKE 'E%' GROUP BY ALL [Group]
```

- The output needs to display all the groups regardless of whether they had any sales or not.
- To achieve this, the code makes use of GROUP BY with ALL.
- Apart from the rows that are displayed, it will also display the group 'Pacific' with null values as shown in the following figure:

	Group	TotalSales
1	Europe	13590506.0212
2	North America	33182889.0168
3	Pacific	NULL

- This is because the Pacific region did not have any sales.

# GROUP BY with HAVING 1-2

HAVING clause is used only with SELECT statement to specify a search condition for a group.

The HAVING clause acts as a WHERE clause in places where the WHERE clause cannot be used against aggregate functions such as SUM ( ) .

Once you have created groups with a GROUP BY clause, you may wish to filter the results further.

The HAVING clause acts as a filter on groups, similar to how the WHERE clause acts as a filter on rows returned by the FROM clause.

# GROUP BY with HAVING 2-2

- The syntax of GROUP BY with HAVING is as follows:

## Syntax:

```
SELECT <column_name> FROM <table_name> GROUP BY <column_name> HAVING  
<search_condition>
```

- Following code snippet displays the row with the group 'Pacific' as it has total sales less than 6000000:

```
SELECT [Group],SUM(SalesYTD) AS 'TotalSales' FROM Sales.SalesTerritory  
WHERE [Group] LIKE 'N%' OR [Group] LIKE 'E%' GROUP BY ALL [Group]
```

- The output of this is only row, with Group name Pacific and total sales, 5977814.9154.

# CUBE 1-2

CUBE is an aggregate operator that produces a super-aggregate row.

In addition to the usual rows provided by the GROUP BY, it also provides the summary of the rows that the GROUP BY clause generates.

The summary row is displayed for every possible combination of groups in the resultset.

The summary row displays NULL in the resultset but at the same time returns all the values for those.

➤ The syntax for CUBE is as follows:

## Syntax:

```
SELECT <column_name> FROM <table_name> GROUP BY <column_name> WITH CUBE
```

## CUBE 2-2

- Following code snippet demonstrates the use of CUBE:

```
SELECT Name, CountryRegionCode, SUM(SalesYTD) AS TotalSales FROM  
Sales.SalesTerritory WHERE Name <> 'Australia' AND Name<> 'Canada '  
GROUP BY Name, CountryRegionCode WITH CUBE
```

- The query retrieves and displays the total sales of each country and also, the total of the sales of all the countries' regions.
- The output is shown in the following figure:

	Name	CountryRegionCode	TotalSales
1	Germany	DE	3805202.3478
2	NULL	DE	3805202.3478
3	France	FR	4772398.3078
4	NULL	FR	4772398.3078
5	United Kingdom	GB	5012905.3656
6	NULL	GB	5012905.3656
7	Central	US	3072175.118
8	Northeast	US	2402176.8476
9	Northwest	US	7887186.7882
10	Southeast	US	2538667.2515

# ROLLUP 1-2

In addition to the usual rows that are generated by the GROUP BY clause, it also introduces summary rows into the resultset.

It is similar to CUBE operator but generates a resultset that shows groups arranged in a hierarchical order.

It arranges the groups from the lowest to the highest.

Group hierarchy in the result is dependent on the order in which the columns that are grouped are specified.

➤ The syntax of ROLLUP is as follows:

## Syntax:

```
SELECT <column_name> FROM <table_name> GROUP BY <column_name> WITH ROLLUP
```



## ROLLUP 2-2

- Following code snippet demonstrates the use of ROLLUP:

```
SELECT Name, CountryRegionCode, SUM(SalesYTD) AS TotalSales
FROM Sales.SalesTerritory
WHERE Name <> 'Australia' AND Name <> 'Canada'
GROUP BY Name, CountryRegionCode
WITH ROLLUP
```

- It retrieves and displays the total sales of each country, the total of the sales of all the countries' regions and arranges them in order.
- The output is shown in the following figure:

	Name	TotalSales
1	Australia	5977814.9154
2	Canada	6771829.1376
3	Central	3072175.118
4	France	4772398.3078
5	Germany	3805202.3478
6	Northeast	2402176.8476
7	Northwest	7887186.7882
8	Southeast	2538667.2515
9	Southwest	10510853.8...
10	United Kingdom	5012905.3656
11	NULL	52751209.9...

# Aggregate Functions 1-5

Aggregate functions help to perform analysis across rows, such as counting rows meeting specific criteria or summarizing total sales for all orders.

Aggregate functions return a single value and can be used in `SELECT` statements with a single expression, such as `SELECT`, `HAVING`, and `ORDER BY` clauses.

Aggregate functions ignore `NULL`s, except when using `COUNT (*)`.

Aggregate functions in a `SELECT` list do not generate a column alias. You may wish to use the `AS` clause to provide one.

Aggregate functions in a `SELECT` clause operate on all rows passed to the `SELECT` phase. If there is no `GROUP BY` clause, all rows will be summarized.

# Aggregate Functions 2-5

- SQL Server provides many built-in aggregate functions.
- The following table lists the commonly used functions:

Function Name	Syntax	Description
AVG	AVG(<expression>)	Calculates the average of all the non-NULL numeric values in a column.
COUNT or COUNT_BIG	COUNT(*) or COUNT(<expression>)	<p>When (*) is used, this function counts all rows, including those with NULL. The function returns count of non-NULL rows for the column when a column is specified as &lt;expression&gt;.</p> <p>The return value of COUNT function is an int. The return value of COUNT_BIG is a big_int.</p>
MAX	MAX(<expression>)	Returns the largest number, latest date/time, or last occurring string.
MIN	MIN(<expression>)	Returns the smallest number, earliest date/time, or first occurring string.
SUM	SUM(<expression>)	Calculates the sum of all the non-NULL numeric values in a column.

# Aggregate Functions 3-5

- The following code snippet demonstrates the use of built-in aggregate in a SELECT clause:

```
SELECT AVG([UnitPrice]) AS AvgUnitPrice,  
MIN([OrderQty]) AS MinQty,  
MAX([UnitPriceDiscount]) AS MaxDiscount  
FROM Sales.SalesOrderDetail;
```

- Since the query does not use a GROUP BY clause, all rows in the table will be summarized by the aggregate formulas in the SELECT clause.
- The output is shown in the following figure:

	AvgUnitPrice	MinQty	MaxDiscount
1	465.0934	1	0.40

- When using aggregates in a SELECT clause, all columns referenced in the SELECT list must be used as inputs for an aggregate function or must be referenced in a GROUP BY clause.
- Failing this, there will be an error.

# Aggregate Functions 4-5

- The following code snippet will return an error:

```
SELECT SalesOrderID, AVG(UnitPrice) AS AvgPrice  
FROM Sales.SalesOrderDetail;
```

This returns an error stating that the column `Sales.SalesOrderDetail.SalesOrderID` is invalid in the `SELECT` list because it is not contained in either an aggregate function or the `GROUP BY` clause.

As the query is not using a `GROUP BY` clause, all rows will be treated as a single group. All columns, therefore, must be used as inputs to aggregate functions.

# Aggregate Functions 5-5

- To correct or prevent the error, one needs to remove SalesOrderID from the query.
- Besides using numeric data, aggregate expressions can also include date, time, and character data for summarizing.
- Following code snippet returns the earliest and latest order date, using MIN and MAX function:

```
SELECT MIN(OrderDate) AS Earliest,  
       MAX(OrderDate) AS Latest  
FROM Sales.SalesOrderHeader;
```

- Following figure shows the output:

	Earliest	Latest
1	2005-07-01 00:00:00.000	2008-07-31 00:00:00.000

# Spatial Aggregates 1-4

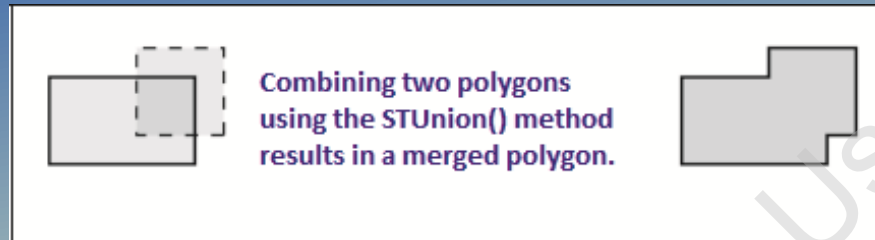
- SQL Server provides several methods that help to aggregate two individual items of geometry or geography data.
- The following table lists the methods:

Method	Description
STUnion	Returns an object that represents the union of a geometry/geography instance with another geometry/geography instance.
STIntersection	Returns an object that represents the points where a geometry/geography instance intersects another geometry/geography instance.
STConvexHull	Returns an object representing the convex hull of a geometry/geography instance.

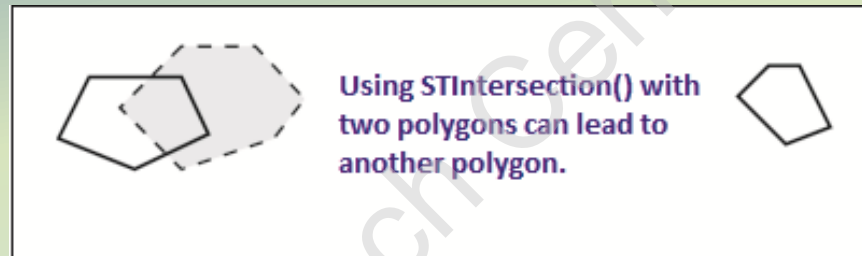
- A set of points is called convex if for any two points, the entire segment is contained in the set.
- The convex hull of a set of points is the smallest convex set containing the set.
- For any given set of points, there is only one convex hull.

# Spatial Aggregates 2-4

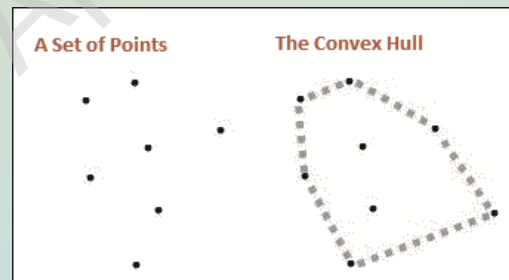
- Following figure depicts an example of `STUnion()`:



- Following figure depicts an example of `STIntersection()`:



- Following figure depicts an example of `STConvexHull()`:



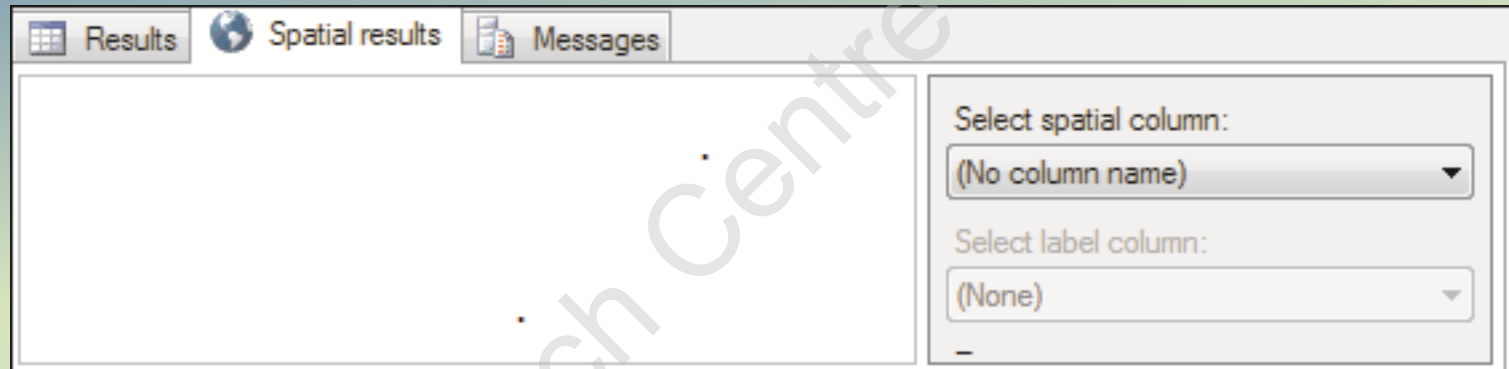


# Spatial Aggregates 3-4

- The following code snippet demonstrates the use of `STUnion()`:

```
SELECT geometry::Point(251, 1,  
4326).STUnion(geometry::Point(252, 2, 4326));
```

- The following figure displays the output as two points:



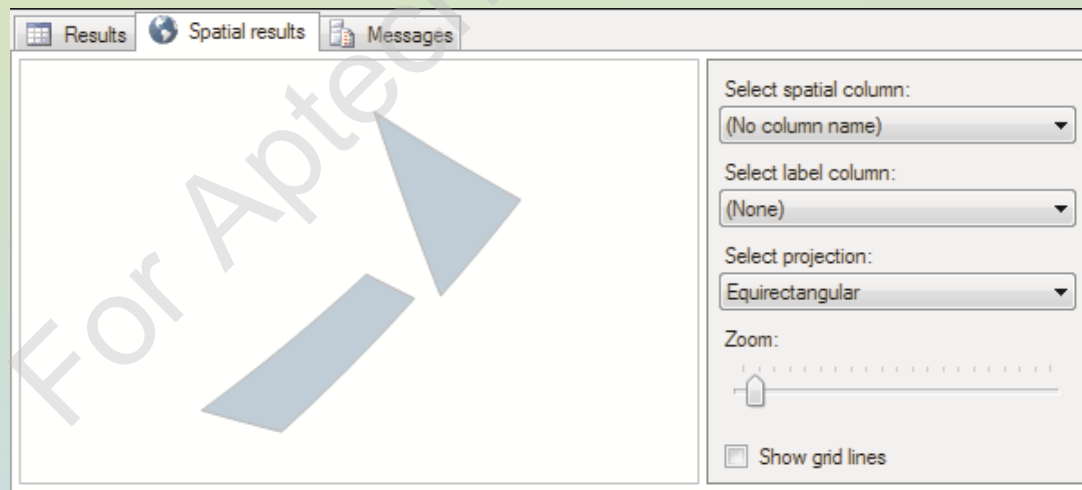
- The following code snippet displays another example:

```
DECLARE @City1 geography  
SET @City1 = geography::STPolyFromText(  
'POLYGON((175.3 -41.5, 178.3 -37.9, 172.8 -34.6, 175.3 -41.5))',  
4326)
```

# Spatial Aggregates 4-4

```
DECLARE @City2 geography
SET @City2 = geography::STPolyFromText(
'POLYGON((169.3 -46.6, 174.3 -41.6, 172.5 -40.7, 166.3 -45.8, 169.3 -
46.6))',
4326)
DECLARE @CombinedCity geography = @City1.STUnion(@City2)
SELECT @CombinedCity
```

- Here, two variables are declared of the geography type and appropriate values are assigned to them.
- Then, they are combined into a third variable of geography type by using the `STUnion()` method. The following figure shows the output of the code:



# New Spatial Aggregates

- SQL Server 2012 has introduced four new aggregates to the suite of spatial operators in SQL Server.

Union Aggregate

Envelope Aggregate

Collection Aggregate

Convex Hull Aggregate

- These aggregates are implemented as static methods, which work for either the `geography` or the `geometry` data types.
- Although aggregates are applicable to all classes of spatial data, they can be best described with polygons.

# Union Aggregate 1-2

It performs a union operation on a set of geometry objects.

It combines multiple spatial objects into a single spatial object, removing interior boundaries, where applicable.

- The syntax of UnionAggregate is as follows:

## Syntax:

```
UnionAggregate ( geometry_operand or geography_operand)
```

where,

**geometry\_operand:** is a geometry type table column comprising the set of geometry objects on which a union operation will be performed.

**geography\_operand:** is a geography type table column comprising the set of geography objects on which a union operation will be performed.

# Union Aggregate 2-2

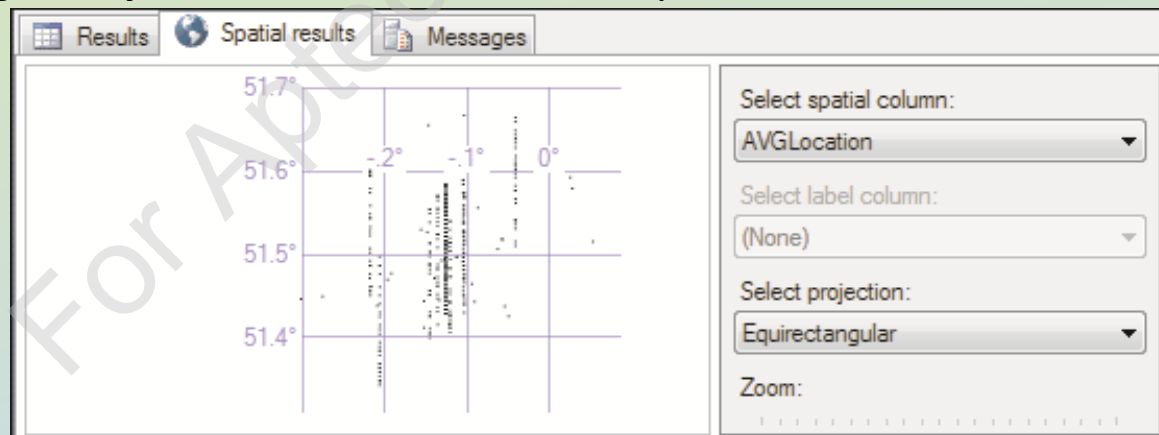
- Following code snippet demonstrates a simple example of using the Union aggregate.
- It uses the Person.Address table in the AdventureWorks2012 database.

```
SELECT Geography::UnionAggregate(SpatialLocation)
AS AVGLocation
FROM Person.Address
WHERE City = 'London';
```

- The following figure displays the output:

	AVGLocation
1	0xE61000000104A00100003DA82EC605C249407D37109CAD...

- The following figure displays a visual representation of the spatial data and is viewed by clicking the **Spatial results** tab in the output window:



# Envelope Aggregate 1-2

It returns a bounding area for a given set of `geometry` or `geography` objects. It exhibits different behaviors for `geography` and `geometry` types.

For the `geometry` type, the result is a 'traditional' rectangular polygon, which closely bounds the selected input objects.

For the `geography` type, the result is a circular object, which loosely bounds the selected input objects.

Furthermore, the circular object is defined using the new `CurvePolygon` feature.

➤ The following is the syntax of `EnvelopeAggregate`:

## Syntax:

```
EnvelopeAggregate (geometry_operand or geography_operand)
```

where,

`geometry_operand`: is a `geometry` type table column comprising the set of `geometry` objects.

## Envelope Aggregate 2-2

geography\_operand: is a geography type table column comprising the set of geography objects.

- Following code snippet returns a bounding box for a set of objects in a table variable column:

```
SELECT Geography::EnvelopeAggregate(SpatialLocation)
AS Location
FROM Person.Address
WHERE City = 'London'
```

- The following figure shows the visual representation of the output:



# Collection Aggregate 1-2

It returns a `GeometryCollection/GeographyCollection` instance with one geometry/geography part for each spatial object(s) in the selection set.

- The syntax of `CollectionAggregate` is as follows:

## Syntax:

```
CollectionAggregate (geometry_operand or geography_operand)
```

where,

`geometry_operand`: is a geometry type table column comprising the set of geometry objects.

`geography_operand`: is a geography type table column comprising the set of geography objects.

- Following code snippet returns a `GeometryCollection` instance that contains a `CurvePolygon` and a `Polygon`:

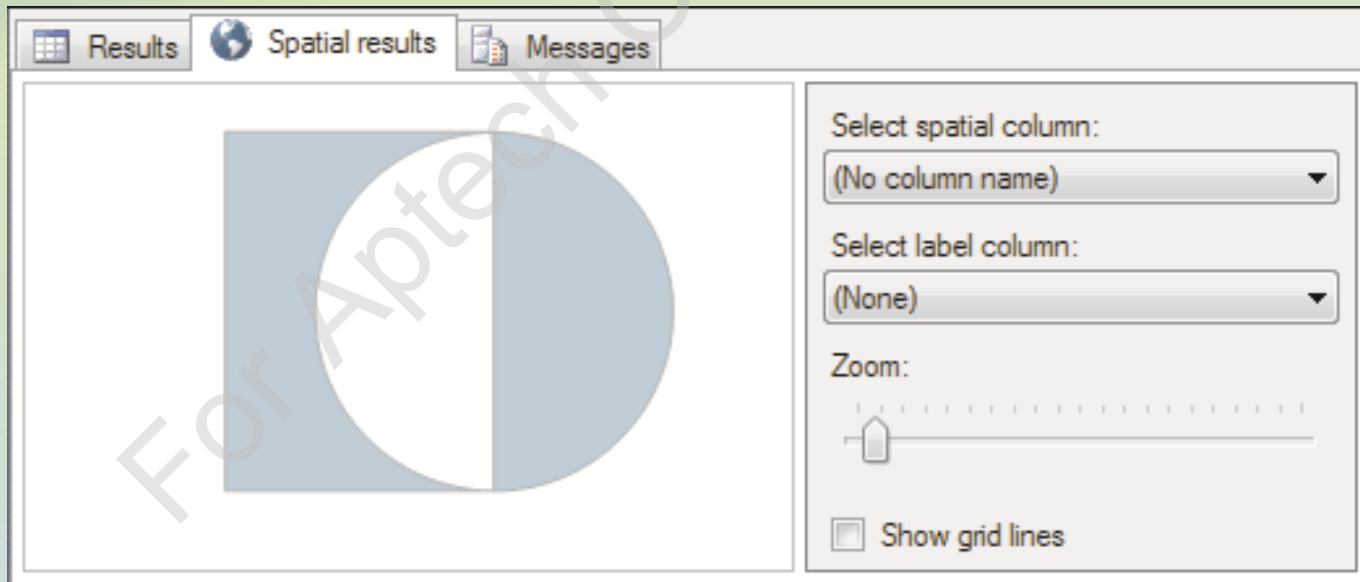
```
DECLARE @CollectionDemo TABLE
(
  shape geometry,
```



# Collection Aggregate 2-2

```
shapeType nvarchar(50)
)
INSERT INTO @CollectionDemo(shape,shapeType)
VALUES('CURVEPOLYGON(CIRCULARSTRING(2 3, 4 1, 6 3, 4 5, 2 3))',
'Circle'),
('POLYGON((1 1, 4 1, 4 5, 1 5, 1 1))', 'Rectangle');
SELECT geometry::CollectionAggregate(shape)
FROM @CollectionDemo;
```

- The following figure shows the output of the code:



# Convex Hull Aggregate 1-2

- It returns a convex hull polygon, which encloses one or more spatial objects for a given set of geometry/geography objects.
- The following is the syntax of ConvexHullAggregate:

## Syntax:

```
ConvexHullAggregate (geometry_operand or geography_operand)
```

where,

geometry\_operand: is a geometry type table column comprising the set of geometry objects.

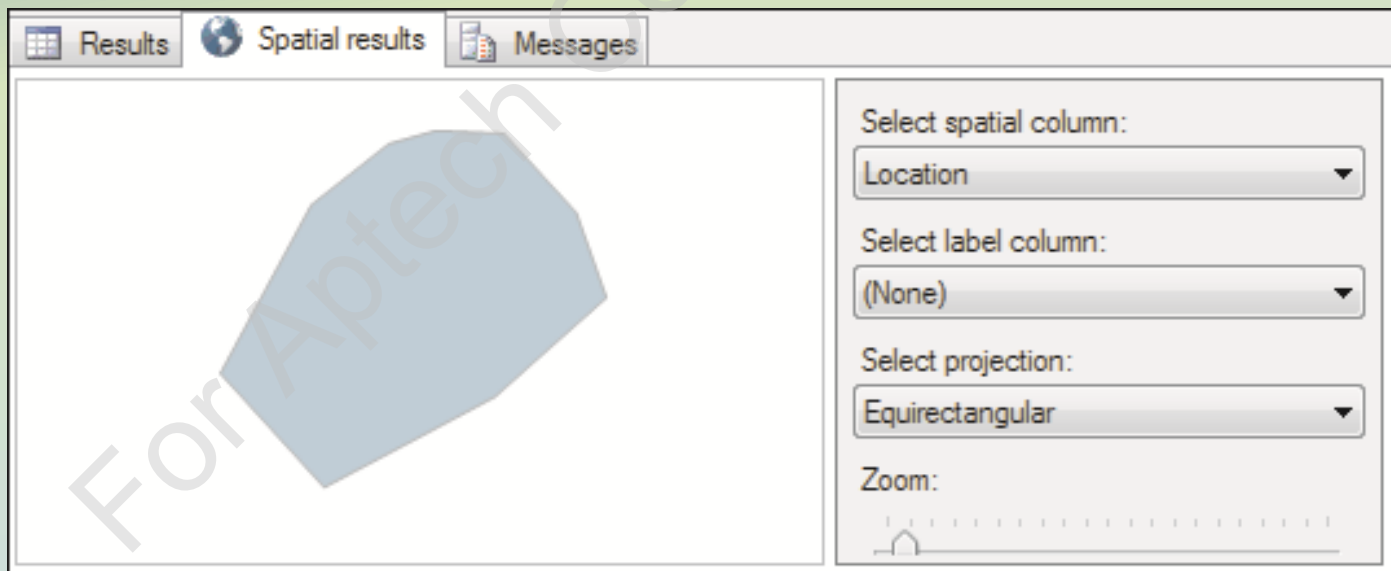
geography\_operand: is a geography type table column comprising the set of geography objects.

# Convex Hull Aggregate 2-2

- Following code snippet demonstrates the use of ConvexHullAggregate:

```
SELECT Geography::ConvexHullAggregate(SpatialLocation)
AS Location
FROM Person.Address
WHERE City = 'London'
```

- The output is shown in the following figure:



# Subqueries 1-3

You can use a `SELECT` statement or a query to return records that will be used as criteria for another `SELECT` statement or query.

The outer query is called parent query and the inner query is called a subquery. The purpose of a subquery is to return results to the outer query.

In other words, the inner query statement should return the column or columns used in the criteria of the outer query statement.

The simplest form of a subquery is one that returns just one column. The parent query can use the results of this subquery using an `=` sign.

- The syntax for the most basic form of a subquery using just one column with an `=` sign is as follows:

## Syntax:

```
SELECT <ColumnName> FROM <table>  
WHERE <ColumnName> = ( SELECT <ColumnName> FROM <Table> WHERE  
<ColumnName> = <Condition> )
```

## Subqueries 2-3

- In a subquery, the innermost `SELECT` statement is executed first and its result is passed as criteria to the outer `SELECT` statement.
- Consider a scenario where it is required to determine the due date and ship date of the most recent orders.
- Following code snippet shows the code to achieve this:

```
SELECT DueDate, ShipDate
FROM Sales.SalesOrderHeader
WHERE Sales.SalesOrderHeader.OrderDate =
      (SELECT MAX(OrderDate)
       FROM Sales.SalesOrderHeader)
```

- Here, a subquery has been used to achieve the desired output.
- The inner query or subquery retrieves the most recent order date.
- This is then passed to the outer query, which displays due date and ship date for all the orders that were made on that particular date.

# Subqueries 3-3

- The following figure displays a part of the output of the code:

	DueDate	ShipDate
1	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
2	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
3	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
4	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
5	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
6	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
7	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
8	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
9	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
10	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000

- Based on the results returned by the inner query, a subquery can be classified as follows:

## Scalar subqueries

- returns a single value. Here, the outer query needs to be written to process a single result.

## Multi-valued subqueries

- returns a result similar to a single-column table. Here, the outer query needs to be written to handle multiple possible results.

# Working with Multi-valued Queries 1-4

If an `=` operator is used with the subquery, the subquery must return a single scalar value.

If more than one value is returned, there will be an error and the query will not be processed.

The keywords `ANY`, `ALL`, `IN`, and `EXISTS` can be used with the `WHERE` clause of a `SELECT` statement when the query returns one column but one or more rows.

The simplest form of a subquery is one that returns just one column. The parent query can use the results of this subquery using an `=` sign.

These keywords, also called predicates, are used with multi-valued queries.

For example, consider that all the first names and last names of employees whose job title is 'Research and Development Manager' need to be displayed.

Here, the inner query may return more than one row as there may be more than one employee with that job title.

To ensure that the outer query can use the results of the inner query, the `IN` keyword will have to be used.

# Working with Multi-valued Queries 2-4

- Following code snippet demonstrates this:

```
SELECT FirstName, LastName FROM Person.Person
WHERE Person.Person.BusinessEntityID IN (SELECT BusinessEntityID
FROM HumanResources.Employee WHERE JobTitle ='Research and Development
Manager');
```

- Here, the inner query retrieves the BusinessEntityID from the HumanResources.Employee table for those records having job title 'Research and Development Manager'.
- These results are then passed to the outer query, which matches the BusinessEntityID with that in the Person.Person table.
- Finally, from the records that are matching, the first and last names are extracted and displayed.
- The following figure displays the output:

	FirstName	LastName
1	Dylan	Miller
2	Michael	Raheem



# Working with Multi-valued Queries 3-4

- You should remember the following points when using subqueries:

The `ntext`, `text`, and `image` data types cannot be used in the `SELECT` list of subqueries.

The `SELECT` list of a subquery introduced with a comparison operator can have only one expression or column name.

Subqueries that are introduced by a comparison operator not followed by the keyword `ANY` or `ALL` cannot include `GROUP BY` and `HAVING` clauses.

You cannot use `DISTINCT` keyword with subqueries that include `GROUP BY`.

You can specify `ORDER BY` only when `TOP` is also specified.

# Working with Multi-valued Queries 4-4

- Besides scalar and multi-valued subqueries, you can also choose between self-contained subqueries and correlated subqueries. These are defined as follows:

## Self-contained subqueries

- These queries are written as standalone queries, without any dependencies on the outer query.
- A self-contained subquery is processed once when the outer query runs and passes its results to the outer query.

## Correlated subqueries

- These queries reference one or more columns from the outer query and therefore, depend on the outer query.
- Correlated subqueries cannot be run separately from the outer query.

# EXISTS Keyword 1-2

- The EXISTS keyword is used with a subquery to check the existence of rows returned by the subquery.
- The subquery does not actually return any data; it returns a value of TRUE or FALSE.
- The syntax of a subquery containing the word EXISTS is as follows:

## Syntax:

```
SELECT <ColumnName> FROM <table>  
WHERE [NOT] EXISTS  
(  
<Subquery_Statement>  
)
```

where,

Subquery\_Statement: specifies the subquery.

## EXISTS Keyword 2-2

- Following code snippet shows the use of the EXISTS keyword to yield the same output:

```
SELECT FirstName, LastName FROM Person.Person AS A
WHERE EXISTS (SELECT *
FROM HumanResources.Employee As B WHERE JobTitle ='Research and
Development Manager' AND A.BusinessEntityID=B.BusinessEntityID);
```

- Here, the inner subquery retrieves all those records that match job title as 'Research and Development Manager' and whose BusinessEntityId matches with that in the Person table.
- If there are no records matching both these conditions, the inner subquery will not return any rows.
- Thus, in that case, the EXISTS will return false and the outer query will also not return any rows.
- However, the code will return two rows because the given conditions are satisfied.
- Similarly, one can use the NOT EXISTS keyword.
- The WHERE clause in which it is used is satisfied if there are no rows returned by the subquery.

# Nested Subqueries

- A subquery that is defined inside another subquery is called a nested subquery.
- Consider that you wanted to retrieve and display the names of persons from Canada.
- There is no direct way to retrieve this information since the `Sales.SalesTerritory` table is not related to `Person.Person` table.
- Hence, a nested subquery is used here as shown in the following code snippet:

```
SELECT LastName, FirstName
FROM Person.Person
WHERE BusinessEntityID IN
    (SELECT BusinessEntityID
     FROM Sales.SalesPerson
     WHERE TerritoryID IN
        (SELECT TerritoryID
         FROM Sales.SalesTerritory
         WHERE Name='Canada'))
)
```

- The following figure displays the output:

	LastName	FirstName
1	Vargas	Garrett
2	Saraiva	José

# Correlated Queries 1-2

When a subquery takes parameters from its parent query, it is known as Correlated subquery.

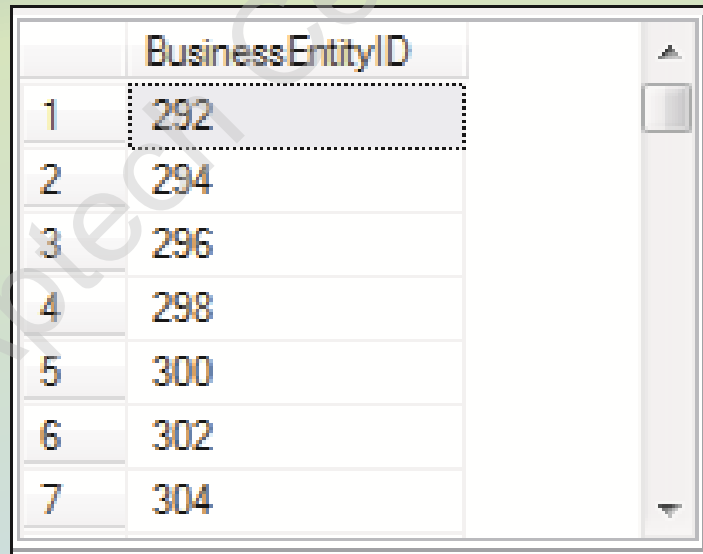
Consider that you want to retrieve all the business entity ids of persons whose contact information was last modified not earlier than 2012.

- The following code snippet displays the use of a correlated subquery:

```
SELECT e.BusinessEntityID
FROM Person.BusinessEntityContact e
WHERE e.ContactTypeID IN
(
  SELECT c.ContactTypeID
  FROM Person.ContactType c
  WHERE YEAR(e.ModifiedDate) >=2012
)
```

## Correlated Queries 2-2

- In the code, the inner query retrieves contact type ids for all those persons whose contact information was modified on or before 2012.
- These results are then passed to the outer query, which matches these contact type ids with those in the `Person.BusinessEntityContact` table and displays the business entity IDs of those records.
- Following figure shows part of the output:



	BusinessEntityID
1	292
2	294
3	296
4	298
5	300
6	302
7	304

# Joins 1-3

- Joins are used to retrieve data from two or more tables based on a logical relationship between tables.
- It defines the manner in which two tables are related in a query by:

Specifying the column from each table to be used for the join. A typical join specifies a foreign key from one table and its associated key in the other table.

Specifying a logical operator such as =, <> to be used in comparing values from the columns.

- Joins can be specified in either the FROM or WHERE clauses.
- The syntax of the JOIN statement is as follows:

## Syntax:

```
SELECT <ColumnName1>, <ColumnName2>...<ColumnNameN>
FROM Table_A AS Table_Alias_A
JOIN
Table_B AS Table_Alias_B
ON
Table_Alias_A.<CommonColumn> = Table_Alias_B.<CommonColumn>
```

where,

<ColumnName1>, <ColumnName2>: Is a list of columns that need to be displayed.



## Joins 2-3

Table\_A: Is the name of the table on the left of the JOIN keyword.

Table\_B: Is the name of the table on the right of the JOIN keyword.

AS Table\_Alias: Is a way of giving an alias name to the table. An alias defined for the table in a query can be used to denote a table so that the full name of the table need not be used.

<CommonColumn>: Is a column that is common to both the tables. In this case, the join succeeds only if the columns have matching values.

- Consider that you want to list employee first names, last names, and their job titles from the `HumanResources.Employee` and `Person.Person`.
- To extract this information from the two tables, you need to join them based on `BusinessEntityID` as shown in the following code snippet:

```
SELECT A.FirstName, A.LastName, B.JobTitle
FROM Person.Person A
JOIN
HumanResources.Employee B
ON
A.BusinessEntityID = B.BusinessEntityID;
```

## Joins 3-3

- Here, the tables `HumanResources.Employee` and `Person.Person` are given aliases `A` and `B`. They are joined together on the basis of their business entity ids.
- The `SELECT` statement then retrieves the desired columns through the aliases.
- Following figure shows the output:

	FirstName	LastName	JobTitle
1	Ken	Sánchez	Chief Executive Officer
2	Terri	Duffy	Vice President of Engineering
3	Roberto	Tamburello	Engineering Manager
4	Rob	Walters	Senior Tool Designer
5	Gail	Erickson	Design Engineer
6	Jossef	Goldberg	Design Engineer
7	Dylan	Miller	Research and Development Manager

# Inner Join

An inner join is formed when records from two tables are combined only if the rows from both the tables are matched based on a common column.

- The syntax of an inner join is as follows:

## Syntax:

```
SELECT <ColumnName1>, <ColumnName2>...<ColumnNameN> FROM  
Table_A AS Table_Alias_A  
INNER JOIN  
Table_B AS Table_Alias_B  
ON  
Table_Alias_A.<CommonColumn> = Table_Alias_B.<CommonColumn>
```

- Following code snippet demonstrates the use of inner join:

```
SELECT A.FirstName, A.LastName, B.JobTitle  
FROM Person.Person A  
INNER JOIN HumanResources.Employee B  
ON  
A.BusinessEntityID = B.BusinessEntityID;
```

- Here, an inner join is constructed between `Person.Person` and `HumanResources.Employee` based on common business entity ids.

# Outer Join

Outer joins are join statements that return all rows from at least one of the tables specified in the `FROM` clause, as long as those rows meet any `WHERE` or `HAVING` conditions of the `SELECT` statement.

- The two types of commonly used outer joins are as follows:

Left Outer Join

Right Outer Join

# Left Outer Join 1-3

Left outer join returns all the records from the left table and only matching records from the right table.

- The syntax of an outer join is as follows:

## Syntax:

```
SELECT <ColumnList> FROM  
Table_A AS Table_Alias_A  
LEFT OUTER JOIN  
Table_B AS Table_Alias_B  
ON  
Table_Alias_A.<CommonColumn> = Table_Alias_B.<CommonColumn>
```

- Consider that you want to retrieve all the customer ids from the `Sales.Customers` table and order information such as ship dates and due dates, even if the customers have not placed any orders.
- Since the record count would be very huge, it is to be restricted to only those orders that are placed before 2012.

## Left Outer Join 2-3

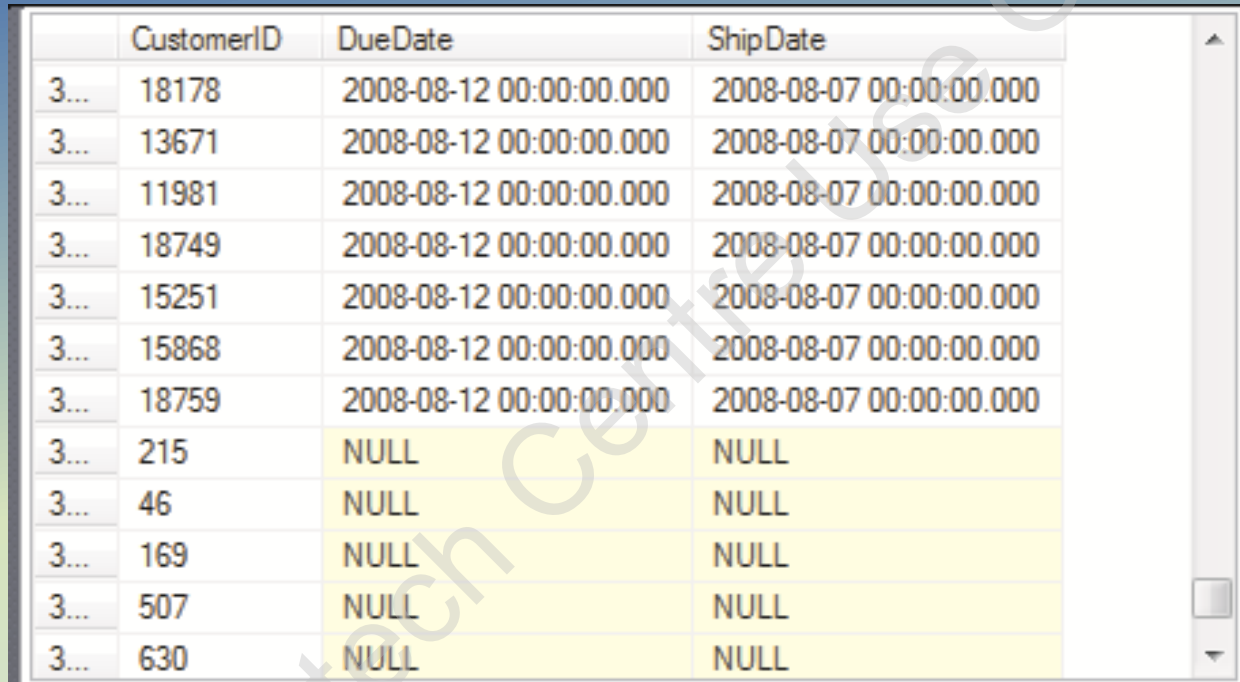
- The following code snippet achieves this by performing a left outer join:

```
SELECT A.CustomerID, B.DueDate, B.ShipDate
FROM Sales.Customer A LEFT OUTER JOIN
Sales.SalesOrderHeader B
ON
A.CustomerID = B.CustomerID AND YEAR(B.DueDate)<2012;
```

- In the query, the left outer join is constructed between the tables `Sales.Customer` and `Sales.SalesOrderHeader`.
- The tables are joined on the basis of customer ids.
- In this case, all records from the left table, `Sales.Customer` and only matching records from the right table, `Sales.SalesOrderHeader`, are returned.

## Left Outer Join 3-3

- Following figure shows the output:



	CustomerID	DueDate	ShipDate
3...	18178	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
3...	13671	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
3...	11981	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
3...	18749	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
3...	15251	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
3...	15868	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
3...	18759	2008-08-12 00:00:00.000	2008-08-07 00:00:00.000
3...	215	NULL	NULL
3...	46	NULL	NULL
3...	169	NULL	NULL
3...	507	NULL	NULL
3...	630	NULL	NULL

- As shown in the output, some records show the due dates and ship dates as NULL.
- This is because for some customers, no order is placed, hence, their records will show the dates as NULL.

# Right Outer Join 1-2

- The right outer join retrieves all the records from the second table in the join regardless of whether there is matching data in the first table or not.
- The syntax of a right outer join is as follows:

## Syntax:

```
SELECT <ColumnList>
FROM Left_Table_Name
AS
Table_A AS Table_Alias_A
RIGHT OUTER JOIN
Table_B AS Table_Alias_B
ON
Table_Alias_A.<CommonColumn> = Table_Alias_B.<CommonColumn>
```



## Right Outer Join 2-2

- Consider that you want to retrieve all the product names from `Product` table and all the corresponding sales order ids from the `SalesOrderDetail` table even if there is no matching record for the products in the `SalesOrderDetail` table.
- The following code snippet achieve this by using a right outer join:

```
SELECT P.Name, S.SalesOrderID
FROM Sales.SalesOrderDetail S
RIGHT OUTER JOIN
Production.Product P
ON P.ProductID = S.ProductID;
```

- In the code, all the records from `Product` table are shown regardless of whether they have been sold or not.

# Self-Join 1-2

- A self-join is used to find records in a table that are related to other records in the same table. A table is joined to itself in a self-join.
- Consider that an **Employee** table in a database named Sterling has a column named **mgr\_id** to denote information for managers whom employees are reporting to.
- Assume that the table has appropriate records inserted in it.
- A manager is also an employee. This means that the **mgr\_id** in the table is the **emp\_id** of an employee.
- For example, **Anabela** with **emp\_id** as **ARD36773F** is an employee but **Anabela** is also a manager for Victoria, Palle, Karla, and other employees as shown in the following figure:

	emp_id	fname	minit	lname	job_id	job_lvl	pub_id	hire_date	mgr_id
1	PMA42628M	Paolo	M	Accorti	13	35	0877	1992-08-27 00:00:00.000	POK93028M
2	PSA89086M	Pedro	S	Afonso	14	89	1389	1990-12-24 00:00:00.000	POK93028M
3	VPA30890F	Victoria	P	Ashworth	6	140	0877	1990-09-13 00:00:00.000	ARD36773F
4	H-B39728F	Helen		Bennett	12	35	0877	1989-09-21 00:00:00.000	POK93028M
5	L-B31947F	Lesley		Brown	7	120	0877	1991-02-13 00:00:00.000	ARD36773F
6	F-C16315M	Francisco		Chang	4	227	9952	1990-11-03 00:00:00.000	MAS70474F
7	PTC11962M	Philip	T	Cramer	2	215	9952	1989-11-11 00:00:00.000	MAS70474F
8	A-C71970F	Aria		Cruz	10	87	1389	1991-10-26 00:00:00.000	POK93028M
9	AMD15433F	Ann	M	Devon	3	200	9952	1991-07-16 00:00:00.000	MAS70474F
10	ARD36773F	Anabela	R	Doming...	8	100	0877	1993-01-27 00:00:00.000	NULL
11	PHF38899M	Peter	H	Franken	10	75	0877	1992-05-17 00:00:00.000	POK93028M
12	PXH22250M	Paul	X	Henriot	5	159	0877	1993-08-19 00:00:00.000	MAS70474F

## Self-Join 2-2

- To get a list of the manager names along with other details, you can use a self-join to join the employee table with itself and then, extract the desired records.
- Following code snippet demonstrates how to use a self-join:

```
SELECT TOP 7 A.fname + ' ' + A.lname AS 'Employee Name', B.fname + ' ' + B.lname AS 'Manager'
FROM
Employee AS A
INNER JOIN
Employee AS B
ON A.mgr_id = B.emp_id
```

- In the code, the Employee table is joined to itself based on the mgr\_id and emp\_id columns.
- The following figure displays the output of the code:

	Employee Name	Manager
1	Paolo Accorti	Pirkko Koskitalo
2	Pedro Afonso	Pirkko Koskitalo
3	Victoria Ashworth	Anabela Domingues
4	Helen Bennett	Pirkko Koskitalo
5	Lesley Brown	Anabela Domingues
6	Francisco Chang	Margaret Smith
7	Philip Cramer	Margaret Smith

# MERGE Statement 1-6

- The `MERGE` statement allows you to maintain a target table based on certain join conditions on a source table using a single statement.
- You can now perform the following actions in one `MERGE` statement:

Insert a new row from the source if the row is missing in the target

Update a target row if a record already exists in the source table

Delete a target row if the row is missing in the source table

- For example, assume you have a `Products` table that maintains records of all products.
- A **`NewProducts`** table maintains records of new products.
- You want to update the `Products` table with records from the **`NewProducts`** table.

# MERGE Statement 2-6

- Here, **NewProducts** table is the source table and **Products** is the target table.
- The **Products** table contains records of existing products with updated data and new products.
- Following figure shows the two tables:

Products					
	ProductID	Name	Type	PurchaseDate	
1	101	Rivets	Hardware	2012-12-01	
2	102	Nuts	Hardware	2012-12-01	
3	103	Washers	Hardware	2011-01-01	
4	104	Rings	Hardware	2013-01-15	
5	105	Paper Clips	Stationery	2013-01-01	

NewProducts					
	ProductID	Name	Type	PurchaseDate	
1	102	Nuts	Hardware	2012-12-01	
2	103	Washers	Hardware	2011-01-01	
3	107	Rings	Hardware	2013-01-15	
4	108	Paper Clips	Stationery	2013-01-01	

# MERGE Statement 3-6

- Consider that you want to:
  - Compare last and first names of customers from both source and target tables
  - Update customer information in target table if the last and first names match
  - Insert new records in target table if the last and first names in source table do not exist in target table
  - Delete existing records in target table if the last and first names do not match with those of source table
- MERGE also allows you to optionally display those records that were inserted, updated, or deleted by using an OUTPUT clause.
- The syntax of the MERGE statement is as follows:

```
MERGE target_table
USING source_table
ON match_condition
WHEN MATCHED THEN UPDATE SET Col1 = val1 [, Col2 = val2...]
WHEN [TARGET] NOT MATCHED THEN INSERT (Col1 [,Col2...] VALUES (Val1 [,
Val2...])
WHEN NOT MATCHED BY SOURCE THEN DELETE
[OUTPUT $action, Inserted.Col1, Deleted.Col1,...] ;
```

where,

target\_table: is the table WHERE changes are being made.

# MERGE Statement 4-6

`source_table`: is the table from which rows will be inserted, updated, or deleted into the target table.

`match_conditions`: are the JOIN conditions and any other comparison operators.

`MATCHED`: true if a row in the `target_table` and `source_table` matches the `match_condition`.

`NOT MATCHED`: true if a row from the `source_table` does not exist in the `target_table`.

`SOURCE NOT MATCHED`: true if a row exists in the `target_table` but not in the `source_table`.

`OUTPUT`: An optional clause that allows to view those records that have been inserted/deleted/updated in `target_table`.

- MERGE statements are terminated with a semi-colon (;).



# MERGE Statement 5-6

- Following code snippet shows how to use MERGE statement. It makes use of the **Sterling** database:

```
MERGE INTO Products AS P1
USING
NewProducts AS P2
ON P1.ProductId = P2.ProductId
WHEN MATCHED THEN
UPDATE SET
P1.Name = P2.Name,
P1.Type = P2.Type,
P1.PurchaseDate = P2.PurchaseDate
WHEN NOT MATCHED THEN
INSERT (ProductId, Name, Type, PurchaseDate)
VALUES (P2.ProductId, P2.Name, P2.Type, P2.PurchaseDate)
WHEN NOT MATCHED BY SOURCE THEN
DELETE
OUTPUT $action, Inserted.ProductId, Inserted.Name, Inserted.Type,
Inserted.PurchaseDate, Deleted.ProductId, Deleted.Name, Deleted.Type,
Deleted.PurchaseDate;
```



# MERGE Statement 6-6

- Following figure shows the output:

	\$action	ProductId	Name	Type	PurchaseDate	ProductId	Name	Type	PurchaseDate
1	INSERT	107	Rings	Hardware	2013-01-15	NULL	NULL	NULL	NULL
2	INSERT	108	Paper Clips	Stationery	2013-01-01	NULL	NULL	NULL	NULL
3	DELETE	NULL	NULL	NULL	NULL	101	Rivets	Hardware	2012-12-01
4	UPDATE	102	Nuts	Hardware	2012-12-01	102	Nuts	Hardware	2012-12-01
5	UPDATE	103	Washers	Hardware	2011-01-01	103	Washers	Hardware	2011-01-01
6	DELETE	NULL	NULL	NULL	NULL	104	Rings	Hardware	2013-01-15
7	DELETE	NULL	NULL	NULL	NULL	105	Paper Clips	Stationery	2013-01-01

- The **NewProducts** table is the source table and **Products** table is the target table.
- The match condition is the column, **ProductId** of both tables.
- If the match condition evaluates to false (NOT MATCHED), then new records are inserted in the target table.
- If match condition evaluates to true (MATCHED), then records are updated into the target table from the source table.
- If records present in the target table do not match with those of source table (NOT MATCHED BY SOURCE), then these are deleted from the target table.
- The last statement displays a report consisting of rows that were inserted/updated/deleted as shown in the output.

# Common Table Expressions (CTEs) 1-5

CTE is similar to a temporary resultset defined within the execution scope of a single SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement.

A CTE is a named expression defined in a query.

A CTE is defined at the start of a query and can be referenced several times in the outer query.

A CTE that include references to itself is called as a recursive CTE.

Key advantages of CTEs are improved readability and ease in maintenance of complex queries.

➤ The syntax to create a CTE is as follows:

## Syntax:

```
WITH <CTE_name>  
AS ( <CTE_definition> )
```

# Common Table Expressions (CTEs) 2-5

- The following code snippet retrieves and displays the customer count year-wise for orders present in the Sales.SalesOrderHeader table:

```
WITH CTE_OrderYear
AS
(
    SELECT YEAR(OrderDate) AS OrderYear, CustomerID
    FROM Sales.SalesOrderHeader
)
SELECT OrderYear, COUNT(DISTINCT CustomerID) AS CustomerCount
FROM CTE_OrderYear
GROUP BY OrderYear;
```

- Here, **CTE\_OrderYear** is specified as the CTE name.
- The **WITH...AS** keywords begins the CTE definition.
- Then, the CTE is used in the **SELECT** statement to retrieve and display the desired results. Following figure shows the output:

	OrderYear	CustomerCount
1	2007	9864
2	2008	11844
3	2005	1216
4	2006	3094

# Common Table Expressions (CTEs) 3-5

- The following guidelines need to be remembered while defining CTEs:

CTEs are limited in scope to the execution of the outer query. Hence, when the outer query ends, the lifetime of the CTE will end.

You need to define a name for a CTE and also, define unique names for each of the columns referenced in the SELECT clause of the CTE.

It is possible to use inline or external aliases for columns in CTEs.

A single CTE can be referenced multiple times in the same query with one definition.

Multiple CTEs can also be defined in the same WITH clause.

# Common Table Expressions (CTEs) 4-5

- Following code snippet defines two CTEs using a single WITH clause:

```
WITH CTE_Students
AS
(
Select StudentCode, S.Name, C.CityName, St.Status
FROM Student S
INNER JOIN City C
ON S.CityCode = C.CityCode
INNER JOIN Status St
ON S.StatusId = St.StatusId)
,
StatusRecord -- This is the second CTE being defined
AS
(
SELECT Status, COUNT(Name) AS CountofStudents
FROM CTE_Students
GROUP BY Status
)
SELECT * FROM StatusRecord
```

# Common Table Expressions (CTEs) 5-5

- The code defines two CTEs using a single `WITH` clause.
- This snippet assumes that three tables named `Student`, `City`, and `Status` are created.
- Assuming some records are inserted in all three tables, the output may be as shown in the following figure:

	Status	Count of Students
1	Failed	2
2	Passed	2

# UNION Operator 1-2

The results from two different query statements can be combined into a single resultset using the UNION operator.

The query statements must have compatible column types and equal number of columns.

The column names can be different in each statement but the data types must be compatible.

➤ The syntax of the UNION operator is as follows:

## Syntax:

```
Query_Statement1  
UNION [ALL]  
Query_Statement2
```

where,

Query\_Statement1 and Query\_Statement2 are SELECT statements.

## UNION Operator 2-2

- Following code snippet demonstrates the use of UNION operator:

```
SELECT Product.ProductId FROM Production.Product
UNION
SELECT ProductId FROM Sales.SalesOrderDetail
```

- This will list all the product ids of both tables that match with each other.
- If you include the ALL clause, all rows are included in the resultset including duplicate records.

```
SELECT Product.ProductId FROM Production.Product
UNION ALL
SELECT ProductId FROM Sales.SalesOrderDetail
```

- By default, the UNION operator removes duplicate records from the resultset.
- However, if you use the ALL clause with UNION operator, then all the rows are returned.
- Apart from UNION, the other operators that are used to combine data from multiple tables are INTERSECT and EXCEPT.



# INTERSECT Operator 1-2

The INTERSECT operator is used with two query statements to return a distinct set of rows that are common to both the query statements.

- The syntax of the INTERSECT operator is as follows:

## Syntax:

```
Query_statement1  
INTERSECT  
Query_statement2
```

where,

Query\_Statement1 and Query\_Statement2 are SELECT statements.

- Following code snippet demonstrates the use of INTERSECT operator:

```
SELECT Product.ProductId FROM Production.Product  
INTERSECT  
SELECT ProductId FROM Sales.SalesOrderDetail
```

# INTERSECT Operator 2-2

- The basic rules for using INTERSECT are as follows:

The number of columns and the order in which they are given must be the same in both the queries.

The data types of the columns being used must be compatible.

- The result of the intersection of the `Production.Product` and `Sales.SalesOrderDetail` tables would be only those product ids that have matching records in `Production.Product` table.

# EXCEPT Operator 1-2

The EXCEPT operator returns all the distinct rows from the query given on the left of the EXCEPT operator and removes all the rows from the resultset that match the rows on the right of the EXCEPT operator.

- The syntax of the EXCEPT operator is as follows:

## Syntax:

```
Query_statement1  
EXCEPT  
Query_statement2
```

where,

Query\_Statement1 and Query\_Statement2 are SELECT statements.

- The two rules that apply to INTERSECT operator are also applicable for EXCEPT operator.

## EXCEPT Operator 2-2

- Following code snippet demonstrates the use of EXCEPT:

```
SELECT Product.ProductId FROM Production.Product  
EXCEPT  
SELECT ProductId FROM Sales.SalesOrderDetail
```

- If the order of the two tables in this example is interchanged, only those rows are returned from Production.
- Product table which do not match with the rows present in Sales.SalesOrderDetail.
- Thus, EXCEPT operator selects all the records from the first table except those which match with the second table.
- Hence, when you are using EXCEPT operator, the order of the two tables in the queries is important.
- Whereas, with the INTERSECT operator, it does not matter which table is specified first.

# PIVOT Operator 1-6

- The brief syntax for PIVOT operator is as follows:

## Syntax:

```
SELECT <non-pivoted column>,  
    [first pivoted column] AS <column name>,  
    [second pivoted column] AS <column name>,  
    ...  
    [last pivoted column] AS <column name>  
FROM  
    (<SELECT query that produces the data>  
    AS <alias for the source query>  
PIVOT  
(  
    <aggregation function>(<column being aggregated>  
FOR  
[<column that contains the values that will become column headers>]  
    IN ( [first pivoted column], [second pivoted column],  
    ... [last pivoted column])  
) AS <alias for the pivot table>  
<optional ORDER BY clause>;
```

# PIVOT Operator 2-6

where,

`table_source`: is a table or table expression.

`aggregate_function`: is a user-defined or in-built aggregate function that accepts one or more inputs.

`value_column`: is the value column of the PIVOT operator.

`pivot_column`: is the pivot column of the PIVOT operator. This column must be of a type that can implicitly or explicitly be converted to `nvarchar()`.

`IN (column_list)`: are values in the `pivot_column` that will become the column names of the output table. The list must not include any column names that already exist in the input `table_source` being pivoted.

`table_alias`: is the alias name of the output table.

- The output of this will be a table containing all columns of the `table_source` except the `pivot_column` and `value_column`.
- These columns of the `table_source`, excluding the `pivot_column` and `value_column`, are called the grouping columns of the pivot operator.

# PIVOT Operator 3-6

- In simpler terms, to use the `PIVOT` operator, you need to supply three elements to the operator:

## Grouping

- In the `FROM` clause, the input columns must be provided.
- The `PIVOT` operator uses those columns to determine which column(s) to use for grouping the data for aggregation.

## Spreading

- Here, a comma-separated list of values that occur in the source data is provided that will be used as the column headings for the pivoted data.

## Aggregation

- An aggregation function, such as `SUM`, to be performed on the grouped rows.

# PIVOT Operator 4-6

- Following code snippet is shown without the PIVOT operator and demonstrates a simple GROUP BY aggregation.
- As the number of records would be huge, the resultset is limited to 5 by specifying TOP 5.

```
SELECT TOP 5 SUM(SalesYTD) AS TotalSalesYTD, Name  
FROM Sales.SalesTerritory  
GROUP BY Name
```

- Following figure shows the output:

	TotalSalesYTD	Name
1	7887186.7882	Northwest
2	2402176.8476	Northeast
3	3072175.118	Central
4	10510853.8739	Southwest
5	2538667.2515	Southeast



# PIVOT Operator 5-6

- The top 5 year to date sales along with territory names grouped by territory names are displayed.
- The same query is rewritten in the following code snippet using a PIVOT so that the data is transformed from a row-based orientation to a column-based orientation:

```
-- Pivot table with one row and six columns
SELECT TOP 5 'TotalSalesYTD' AS GrandTotal,
[Northwest], [Northeast], [Central], [Southwest], [Southeast]
FROM
(SELECT TOP 5 Name, SalesYTD
FROM Sales.SalesTerritory
) AS SourceTable
PIVOT
(
SUM(SalesYTD)
FOR Name IN ([Northwest], [Northeast], [Central], [Southwest],
[Southeast])
) AS PivotTable;
```

# PIVOT Operator 6-6

- Following figure shows the output:

	GrandTotal	Northwest	Northeast	Central	Southwest	Southeast
1	TotalSalesYTD	7887186.7882	2402176.8476	3072175.118	10510853.8739	2538667.2515

- As shown in the figure, the data is transformed and the territory names are now seen as columns instead of rows. This improves readability.
- A major challenge in writing queries using PIVOT is the need to provide a fixed list of spreading elements to the PIVOT operator, such as the specific territory names given in the code.
- It would not be feasible or practical to implement this for large number of spreading elements.
- To overcome this, developers can use dynamic SQL.
- Dynamic SQL provides a means to build a character string that is passed to SQL Server, interpreted as a command, and then, executed.

# UNPIVOT Operator 1-3

- UNPIVOT performs almost the reverse operation of PIVOT, by rotating columns into rows.
- Unpivoting does not restore the original data.
- UNPIVOT has no ability to allocate values to return to the original detail values.
- Instead of turning rows into columns, unpivoting results in columns being transformed into rows.
- SQL Server provides the UNPIVOT table operator to return a row-oriented tabular display from a pivoted data.
- When unpivoting data, one or more columns are defined as the source to be converted into rows.
- The data in those columns is spread, or split, into one or more new rows, depending on how many columns are being unpivoted.
- To use the UNPIVOT operator, you need to provide three elements as follows:

Source columns to be unpivoted

A name for the new column that will display the unpivoted values

A name for the column that will display the names of the unpivoted values

# UNPIVOT Operator 2-3

- Following code snippet shows the code to convert the temporary pivot table into a permanent one so that the same table can be used for demonstrating UNPIVOT operations:

```
-- Pivot table with one row and six columns
SELECT TOP 5 'TotalSalesYTD' AS GrandTotal,
[Northwest], [Northeast], [Central], [Southwest],[Southeast]
FROM
(SELECT TOP 5 Name, SalesYTD
FROM Sales.SalesTerritory
) AS SourceTable
PIVOT
(
SUM(SalesYTD)
FOR Name IN ([Northwest], [Northeast], Central], [Southwest],
[Southeast])
) AS PivotTable;
```

# UNPIVOT Operator 3-3

- Following code snippet demonstrates the use of UNPIVOT operator:

```
SELECT Name, SalesYTD FROM
(SELECT GrandTotal, Northwest, Northeast, Central, Southwest, Southeast
 FROM TotalTable) P
UNPIVOT
(SalesYTD FOR Name IN
(Northwest, Northeast, Central, Southwest, Southeast)
)AS unpvt;
```

- The following figure displays the output:

	GrandTotal	Northwest	Northeast	Central	Southwest	Southeast
1	TotalSalesYTD	7887186.7882	2402176.8476	3072175.118	10510853.8739	2538667.2515

# GROUPING SETS 1-3

The `GROUPING SETS` operator allows you to group together multiple groupings of columns followed by an optional grand total row, denoted by parentheses, `()`.

It is more efficient to use `GROUPING SETS` operators instead of multiple `GROUP BY` with `UNION` clauses because the latter adds more processing overheads on the database server.

- The syntax of the `GROUPING SETS` operator is as follows:

## Syntax:

```
GROUP BY  
GROUPING SETS ( <grouping set list> )
```

where,

grouping set list: consists of one or more columns, separated by commas.

- A pair of parentheses, `()`, without any column name denotes grand total.

# GROUPING SETS 2-3

- Following code snippet demonstrates the GROUPING SETS operator.
- It assumes that a table **Students** is created with fields named **Id**, **Name**, and **Marks** respectively.

```
SELECT Id, Name, AVG(Marks) Marks
FROM Students
GROUP BY
GROUPING SETS
(
(Id, Name, Marks),
(Id),
()
)
```

# GROUPING SETS 3-3

- Following figure shows the output of the code:

	Id	Name	Marks
1	91	Sasha Goldsmith	78
2	91	NULL	78
3	92	Karen Hues	55
4	92	NULL	55
5	93	William Pinter	67
6	93	NULL	67
7	94	Yuri Gogol	89
8	94	NULL	89
9	NULL	NULL	72

- Here, the code uses `GROUPING SETS` to display average marks for every student.
- `NULL` values in **Name** indicate average marks for every student.
- `NULL` value in both **Id** and **Name** columns indicate grand total.



# Summary

- The GROUP BY clause and aggregate functions enabled to group and/or aggregate data together in order to present summarized information.
- Spatial aggregate functions are newly introduced in SQL Server 2012.
- A subquery allows the resultset of one SELECT statement to be used as criteria for another SELECT statement.
- Joins help you to combine column data from two or more tables based on a logical relationship between the tables.
- Set operators such as UNION and INTERSECT help you to combine row data from two or more tables.
- The PIVOT and UNPIVOT operators help to change the orientation of data from column-oriented to row-oriented and vice versa.
- The GROUPING SET subclause of the GROUP BY clause helps to specify multiple groupings in a single query.