Intelligent Data Management with SQL Server

Session: 9

Advanced Queries and Joins

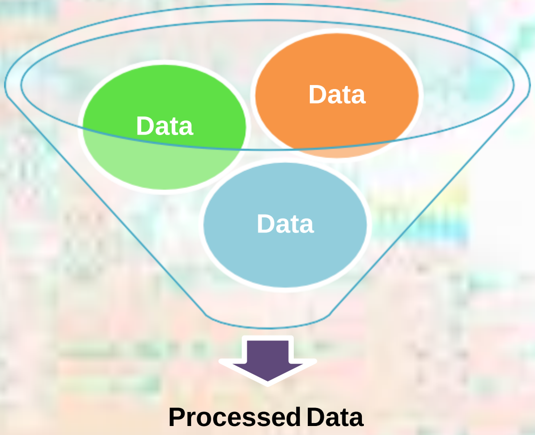
Objectives

- 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 2019 includes several powerful query features that helps to retrieve data efficiently and quickly.
- Data can be grouped and/or aggregated together in order to present summarized information.
- Joins help to combine column data from two or more tables based on a logical relationship between the tables.
- Set operators such as UNION and INTERSECT combines row data from two or more tables.
- IVOT and UNPIVOT operators are used to transform the orientation of data from column-oriented to row-oriented and vice versa.
- GROUPING SET subclause of the GROUP BY clause helps to specify multiple groupings in a single query.

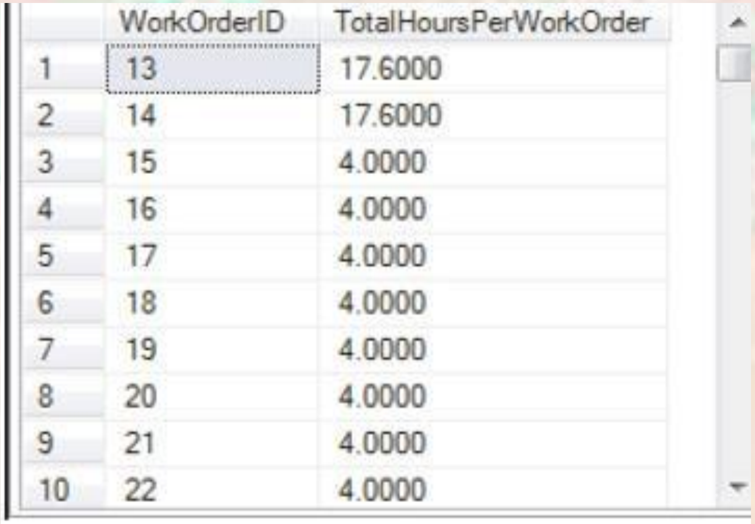
Table 1
Table 3
Table 2



Grouping Data 1-5

This clause partitions the resultset into one or more subsets and each subset has values and expressions in common.

- It is followed by a list of columns, known as grouped columns.
- It restricts the number of rows of the resultset.



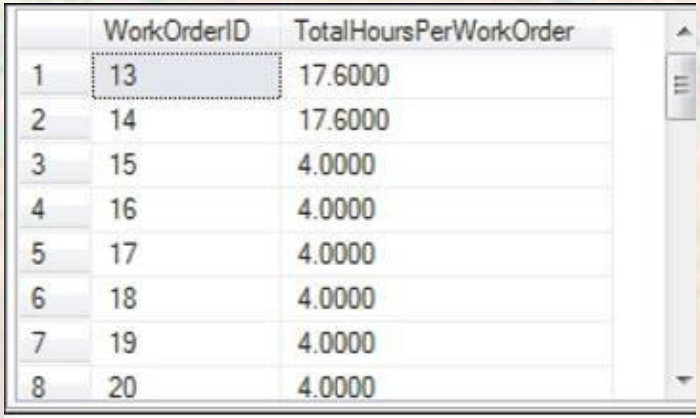
	WorkOrderID	TotalHoursPerWorkOrder
1	13	17.6000
2	14	17.6000
3	15	4.0000
4	16	4.0000
5	17	4.0000
6	18	4.0000
7	19	4.0000
8	20	4.0000
9	21	4.0000
10	22	4.0000

Using GROUP BY Clause

Grouping Data 2-5

GROUP BY with WHERE

- To restrict the rows for grouping.
- The rows 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.



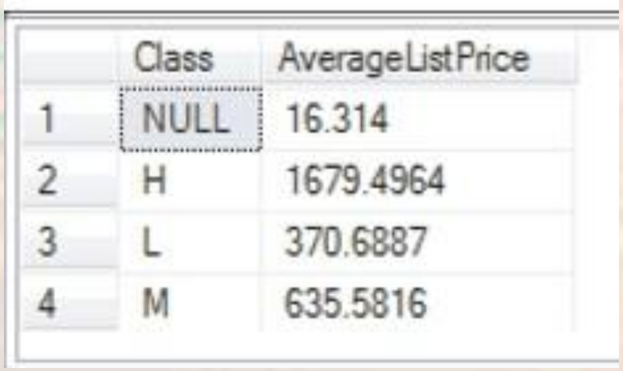
	WorkOrderID	TotalHoursPerWorkOrder
1	13	17.6000
2	14	17.6000
3	15	4.0000
4	16	4.0000
5	17	4.0000
6	18	4.0000
7	19	4.0000
8	20	4.0000

Output of GROUP BY with WHERE

Grouping Data 3-5

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.



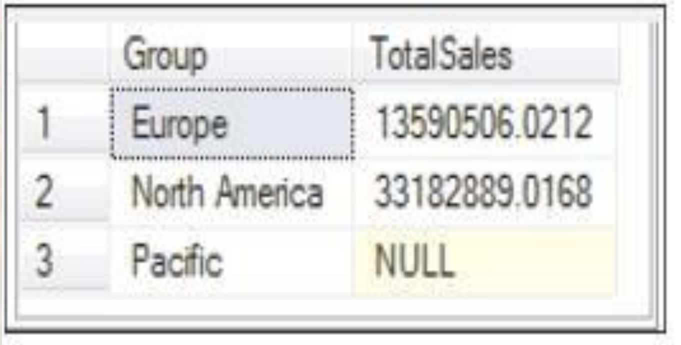
	Class	AverageListPrice
1	NULL	16.314
2	H	1679.4964
3	L	370.6887
4	M	635.5816

Output of GROUP BY with NULL

Grouping Data 4-5

GROUP BY with ALL

- The ALL keyword can also be used with the GROUP BY clause.
- It includes all the groups that the GROUP BY clause produces and even those which do not meet the search conditions.



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

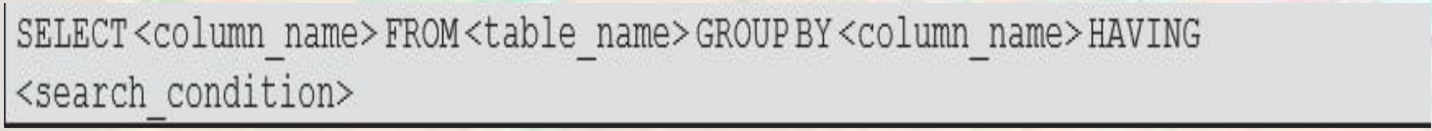
Output of GROUP BY with ALL

Grouping Data 5-5

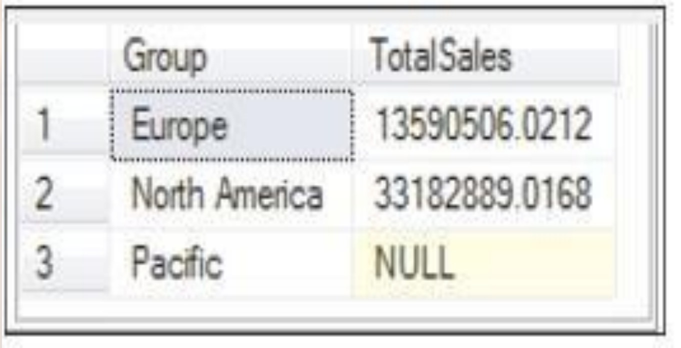
GROUP BY with HAVING

- 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().

Syntax:



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



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

Summarizing Data 1-4

- GROUP BY clause also uses operators such as CUBE and ROLLUP to return summarized data.

- Number of columns in the GROUP BY clause determines number of summary rows in the resultset.

Summarizing Data 2-4

CUBE:

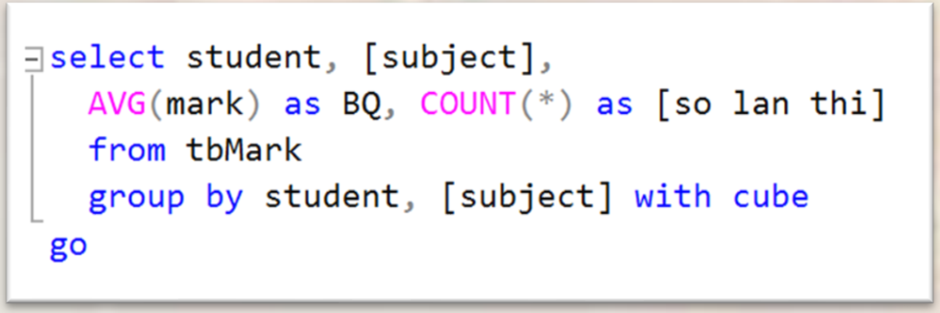
CUBE is an aggregate operator that produces a super-aggregate row.

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

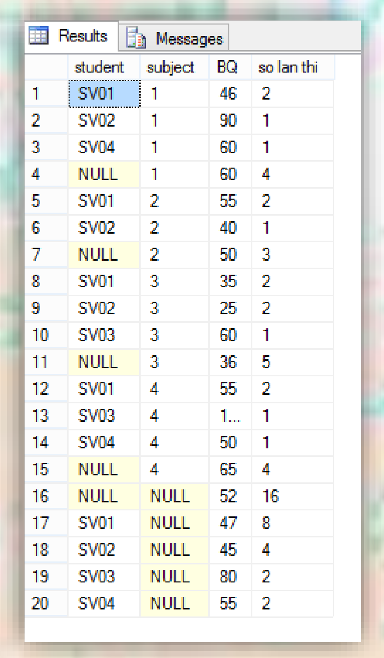
Syntax:

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

- Example



```
select student, [subject],
   AVG(mark) as BQ, COUNT(*) as [so lan thi]
   from tbMark
   group by student, [subject] with cube
go
```



	student	subject	BQ	so lan thi
1	SV01	1	46	2
2	SV02	1	90	1
3	SV04	1	60	1
4	NULL	1	60	4
5	SV01	2	55	2
6	SV02	2	40	1
7	NULL	2	50	3
8	SV01	3	35	2
9	SV02	3	25	2
10	SV03	3	60	1
11	NULL	3	36	5
12	SV01	4	55	2
13	SV03	4	1...	1
14	SV04	4	50	1
15	NULL	4	65	4
16	NULL	NULL	52	16
17	SV01	NULL	47	8
18	SV02	NULL	45	4
19	SV03	NULL	80	2
20	SV04	NULL	55	2

Summarizing Data 3-4

ROLLUP:

- It introduces summary rows into the resultset.
- Generates a resultset that shows groups arranged in a hierarchical order.
- It arranges the groups from the lowest to the highest.

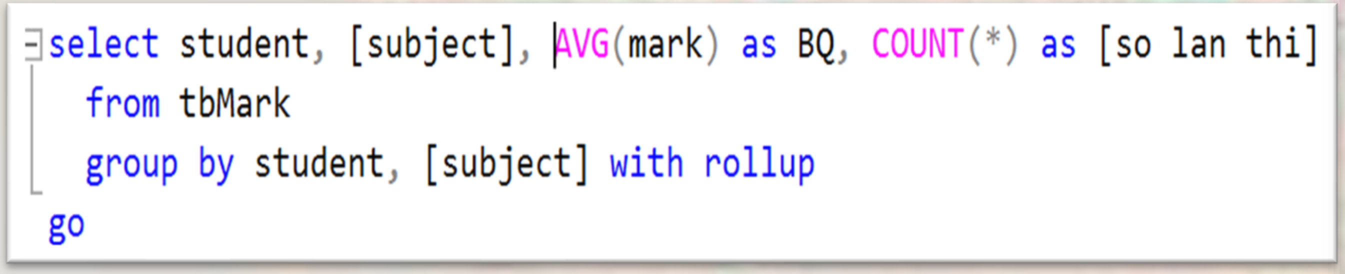
- ROLLUP assumes a hierarchy among the dimension columns and only generates grouping sets based on this hierarchy.
- ROLLUP is often used to generate subtotals and totals for reporting purposes.
- ROLLUP is commonly used to calculate the aggregates of hierarchical data such as sales by year → quarter → month.

Summarizing Data 4-4

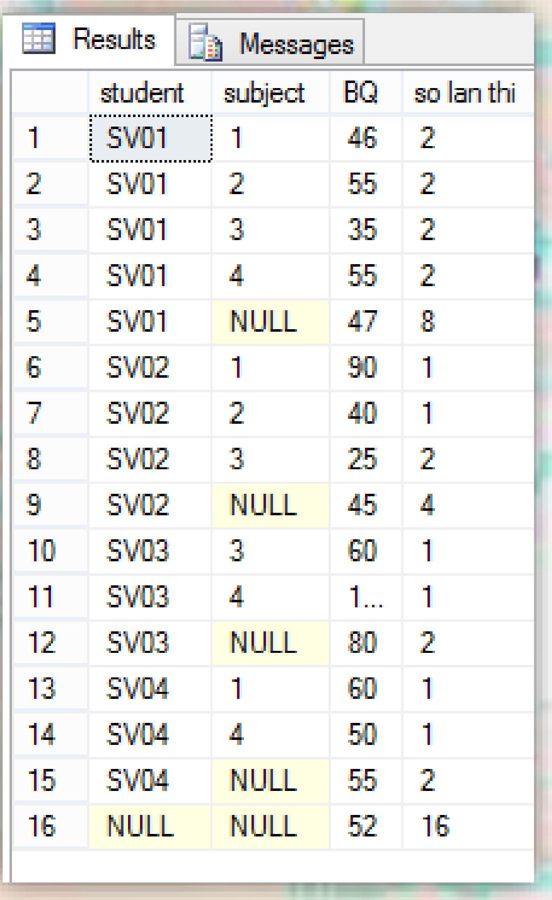
Syntax:

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

- For example



```
select student, [subject], AVG(mark) as BQ, COUNT(*) as [so lan thi]
  from tbMark
  group by student, [subject] with rollup
go
```



	student	subject	BQ	so lan thi
1	SV01	1	46	2
2	SV01	2	55	2
3	SV01	3	35	2
4	SV01	4	55	2
5	SV01	NULL	47	8
6	SV02	1	90	1
7	SV02	2	40	1
8	SV02	3	25	2
9	SV02	NULL	45	4
10	SV03	3	60	1
11	SV03	4	1...	1
12	SV03	NULL	80	2
13	SV04	1	60	1
14	SV04	4	50	1
15	SV04	NULL	55	2
16	NULL	NULL	52	16

Aggregate Functions 1-3

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

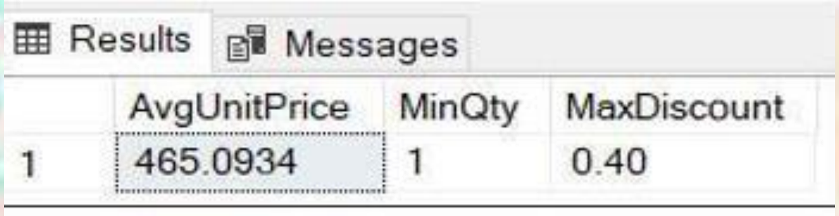
Aggregate functions enable to accomplish it.

Aggregate functions ignore NULLs, except when using COUNT(*).
Aggregate functions in a SELECT list do not generate a column alias.
Aggregate functions in a SELECT clause operate on all rows passed to the SELECT phase.

Aggregate Functions 2-3

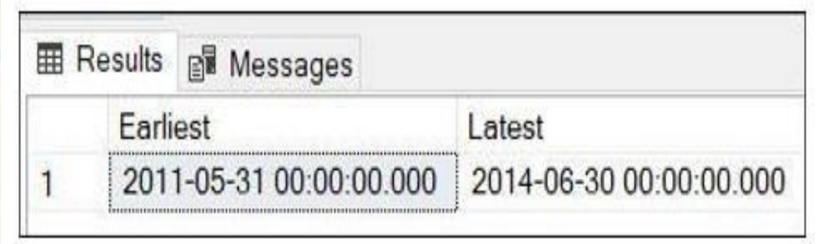
Function Name	Syntax	Description
AVG	AVG(<expression>)	Calculates the average of all the non-NULL numeric values in a column.
COUNT or COUNT_BIG	COUNT(*) or COUNT(<expression>)	When (*) is used, this function counts all rows, including those with NULL. The function returns count of non- NULL rows for the column when a column is specified as <expression>. The return value of COUNT function is an int. The return value of COUNT_BIG is a big_int.
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 3-3



	AvgUnitPrice	MinQty	MaxDiscount
1	465.0934	1	0.40

Using Aggregate Functions

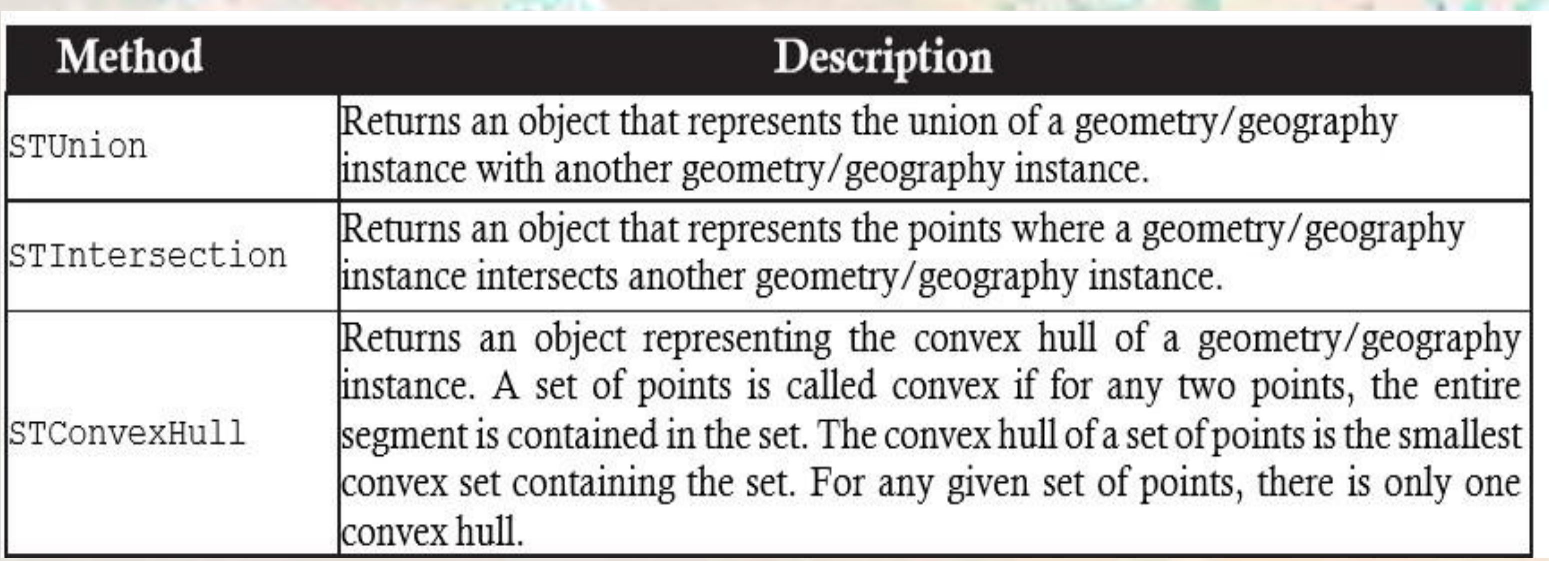


	Earliest	Latest
1	2011-05-31 00:00:00.000	2014-06-30 00:00:00.000

Using Aggregate Functions with Non-Numeric Data

Spatial Aggregates 1-3

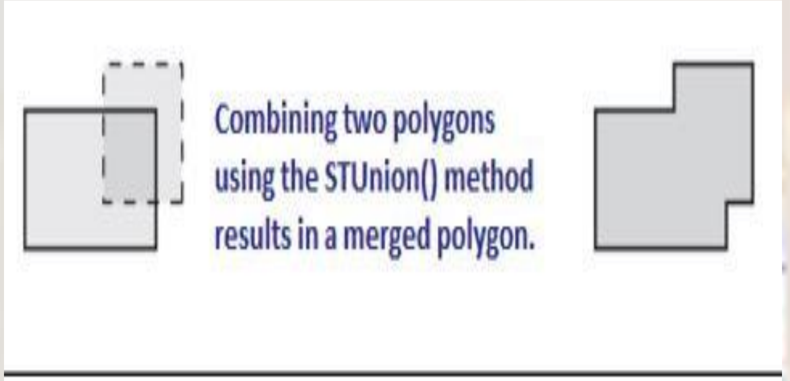
- SQL Server provides several methods that help to aggregate two individual items of geometry or geography data.



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.

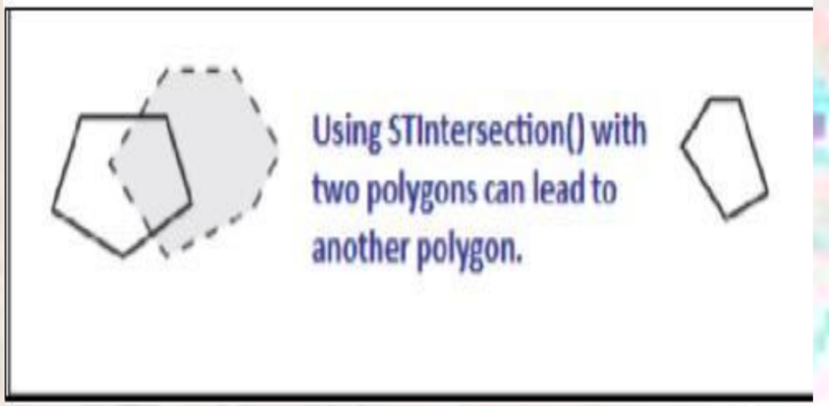
Spatial Aggregate Methods

Spatial Aggregates 2-3



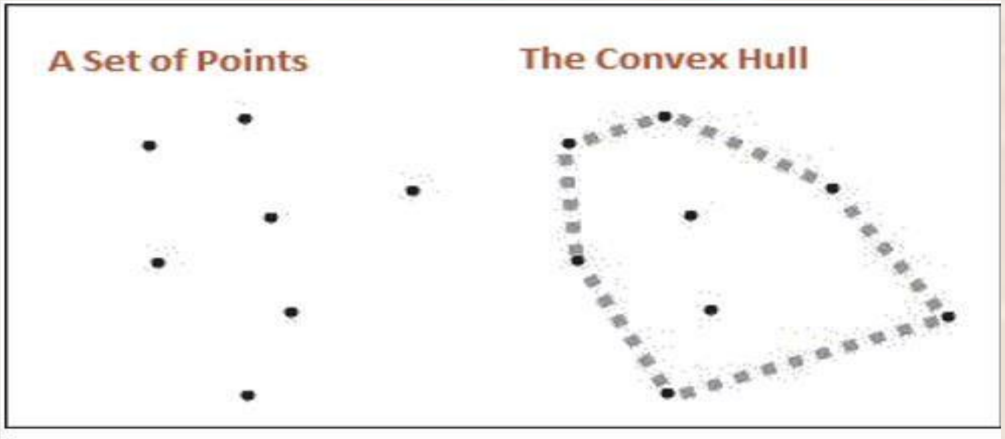
Combining two polygons using the STUnion() method results in a merged polygon.

STUnion()



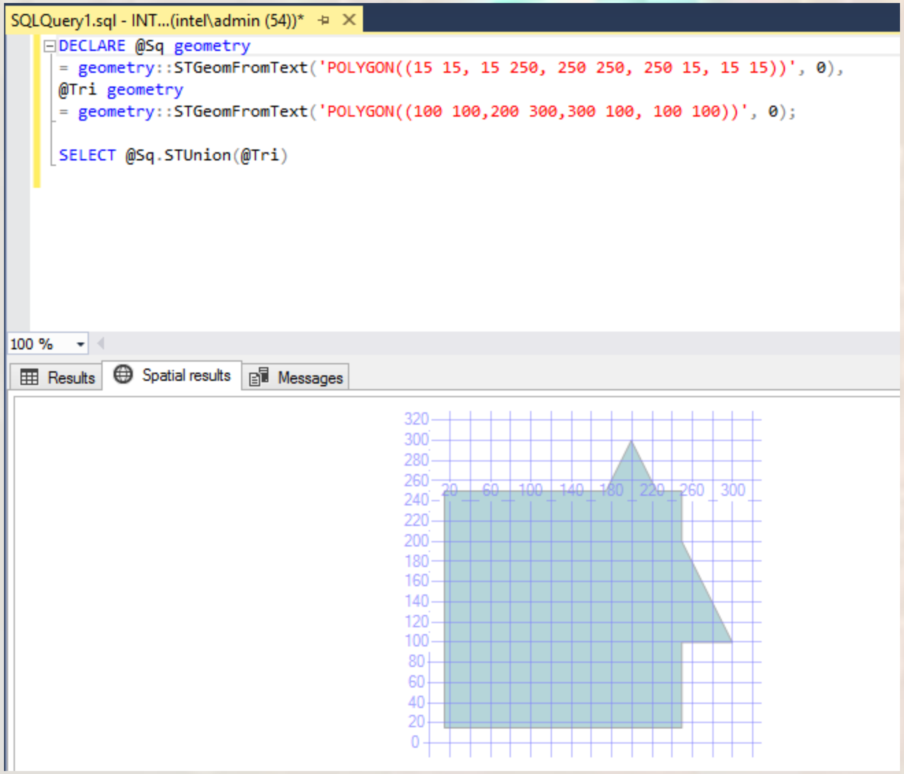
Using STIntersection() with two polygons can lead to another polygon.

STIntersection()



STConvexHull()

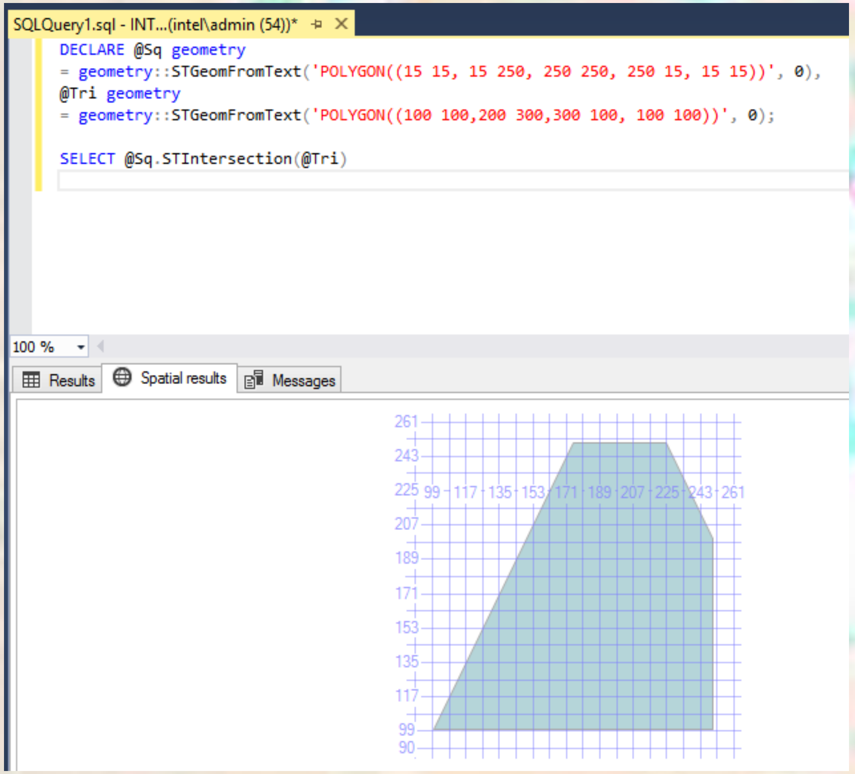
Spatial Aggregates 3-3



```
DECLARE @Sq geometry
= geometry::STGeomFromText('POLYGON((15 15, 15 250, 250 250, 250 15, 15 15))', 0),
@Tri geometry
= geometry::STGeomFromText('POLYGON((100 100,200 300,300 100, 100 100))', 0);

SELECT @Sq.STUnion(@Tri)
```

Using STUnion() with a geographyType

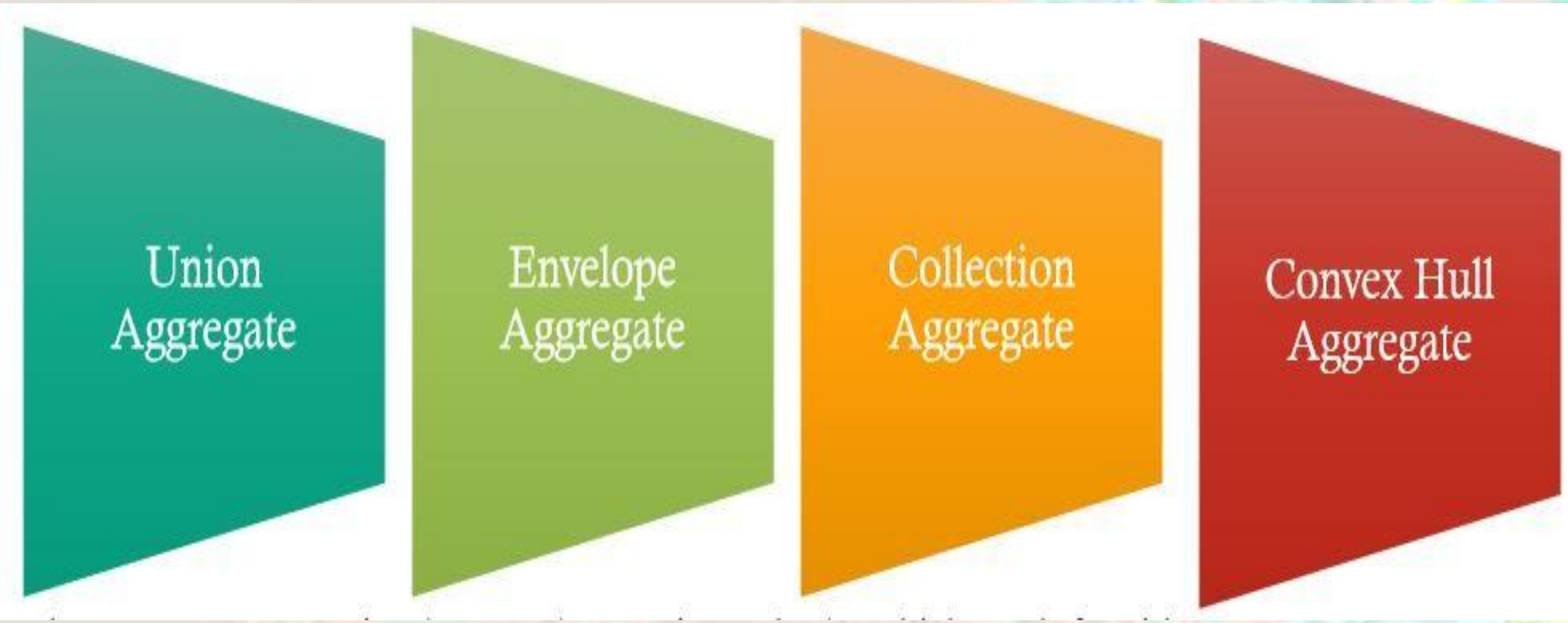


```
DECLARE @Sq geometry
= geometry::STGeomFromText('POLYGON((15 15, 15 250, 250 250, 250 15, 15 15))', 0),
@Tri geometry
= geometry::STGeomFromText('POLYGON((100 100,200 300,300 100, 100 100))', 0);

SELECT @Sq.STIntersection(@Tri)
```

Using STIntersection() with a geographyType

More Spatial Aggregates 1-5

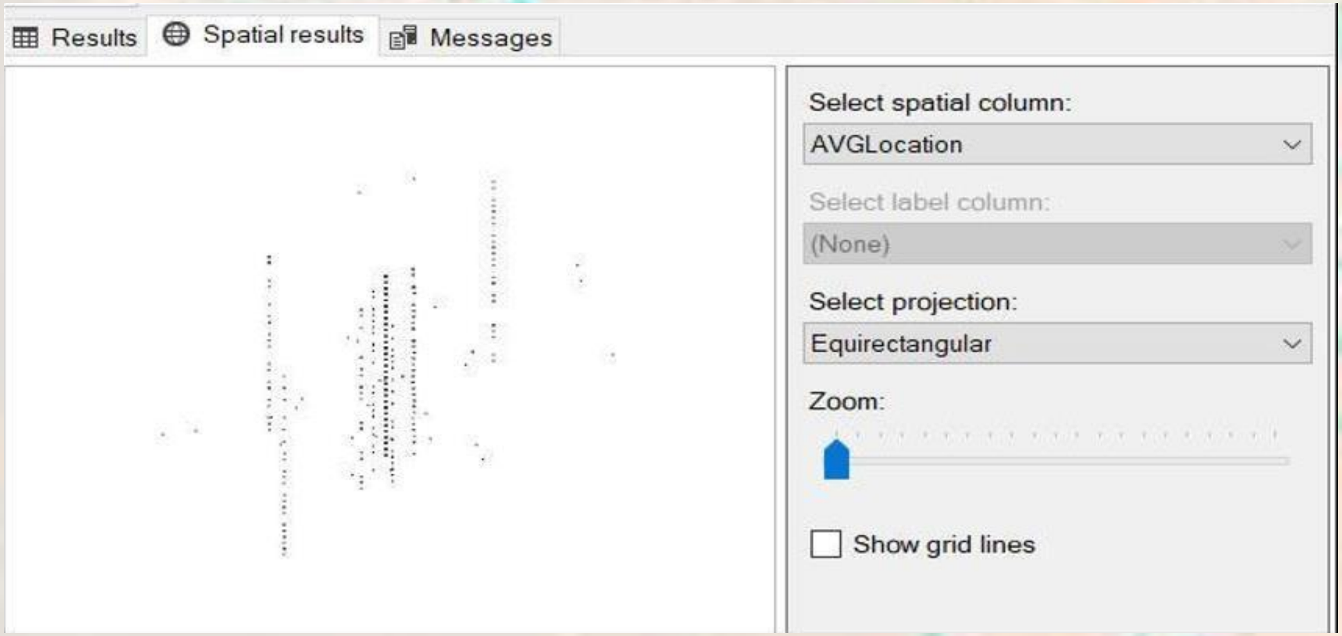


- These aggregates are implemented as static methods, which work for either geography or geometry data types.

More Spatial Aggregates 2-5

Union Aggregate

- 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.



Select spatial column: AVGLocation

Select label column: (None)

Select projection: Equirectangular

Zoom:

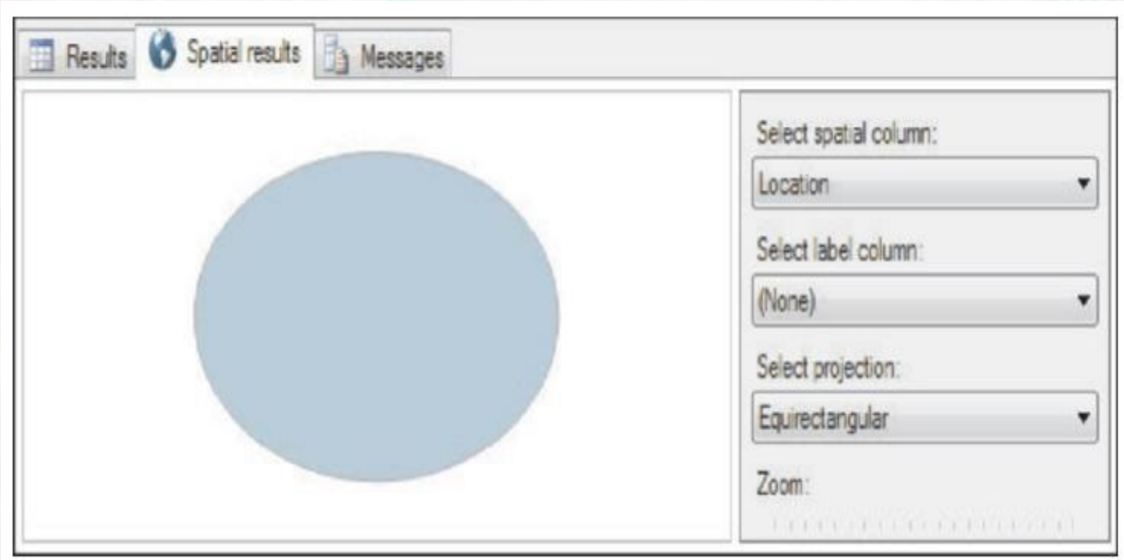
□ Show grid lines

Viewing Spatial Results

More Spatial Aggregates 3-5

Envelope Aggregate

- The Envelope Aggregate returns a bounding area for a given set of geometry or geography objects.
- It exhibits different behaviors for geography and geometry types.



Select spatial column: Location

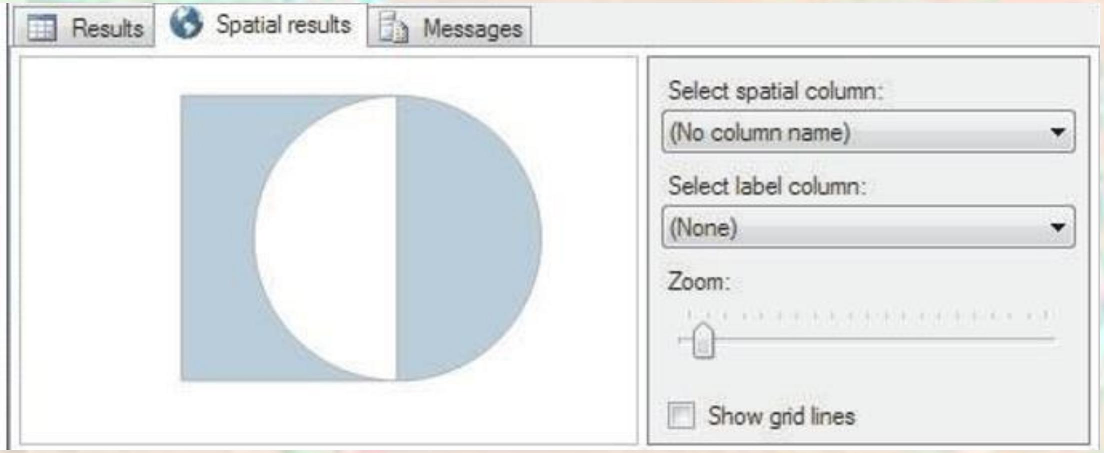
Select label column: (None)

Select projection: Equirectangular

More Spatial Aggregates 4-5

Collection Aggregate

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



Select spatial column: (No column name)

Select label column: (None)

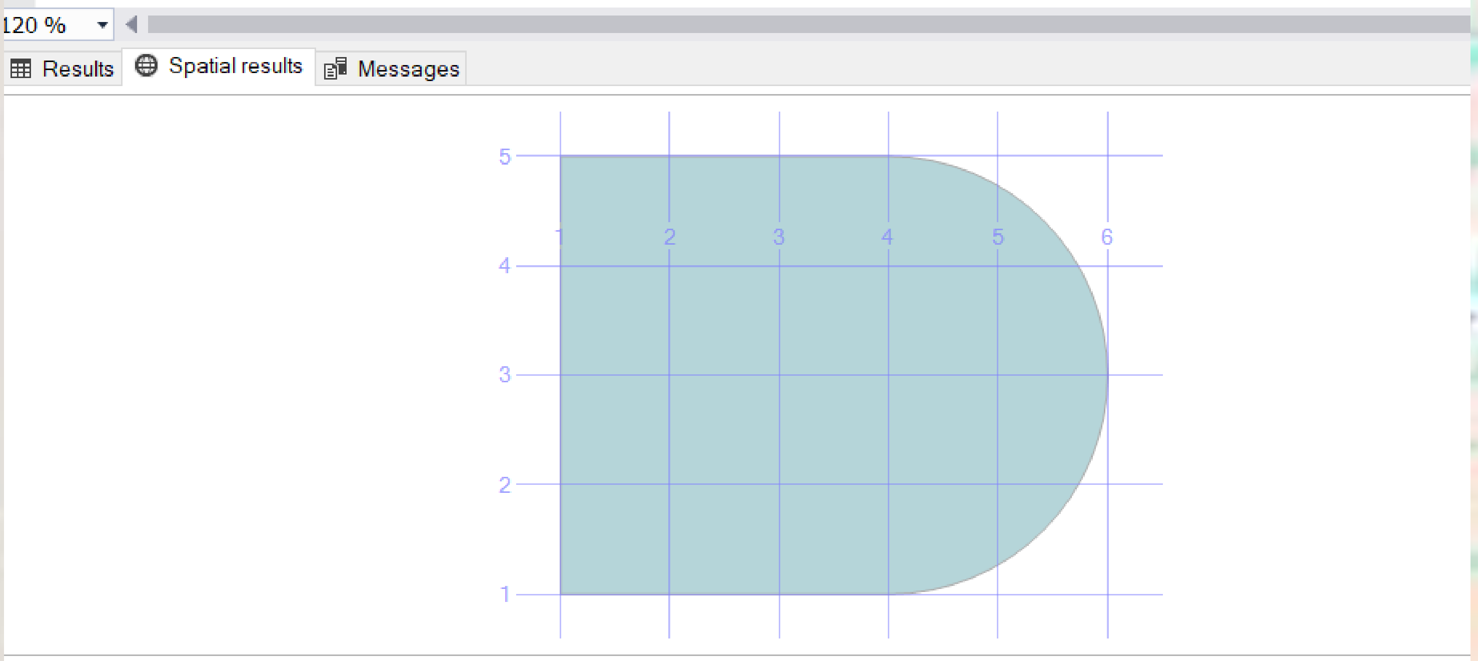
□ Show grid lines

Using CollectionAggregate

More Spatial Aggregates 5-5

Convex Hull Aggregate

- ➢ It returns a convex hull polygon, which encloses one or more spatial objects for a given set of geometry/geography objects.



Using ConvexHullAggregate

Subqueries 1-2

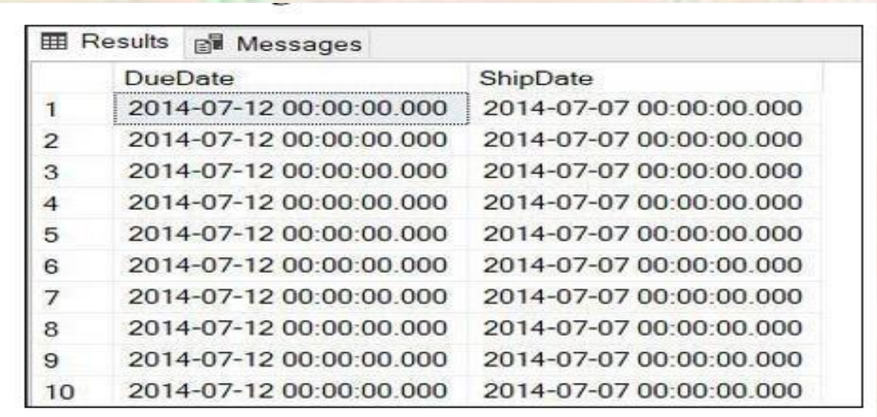
- The outer query is called parent query and the inner query is called a subquery.
- The purpose of a subquery is to return results to the outer query.
- In other words, the inner query statement should return the column or columns used in the criteria of the outer query statement.

Syntax:

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

- Example

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



	DueDate	ShipDate
1	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
2	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
3	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
4	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
5	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
6	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
7	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
8	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
9	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000
10	2014-07-12 00:00:00.000	2014-07-07 00:00:00.000

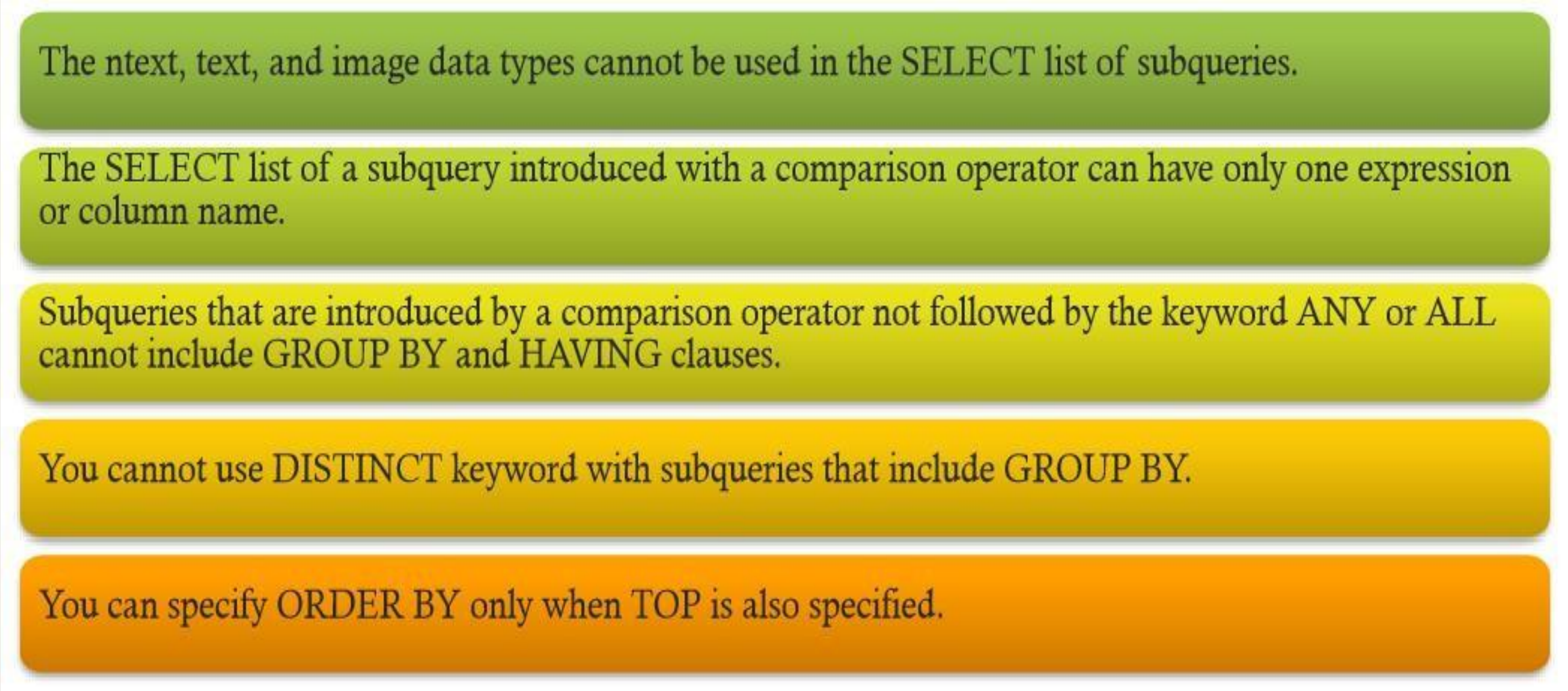
Subqueries 2-2

Based on results returned by inner query, a subquery can be classified as a **scalar subquery** or a **multi-valued subquery**:

Scalar subqueries return a single value. Here, the outer query must be written to process a single result.

Multi-valued subqueries return a result similar to a single-column table. Here, the outer query must be written to handle multiple possible results.

Working with Multi-valued Queries



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.

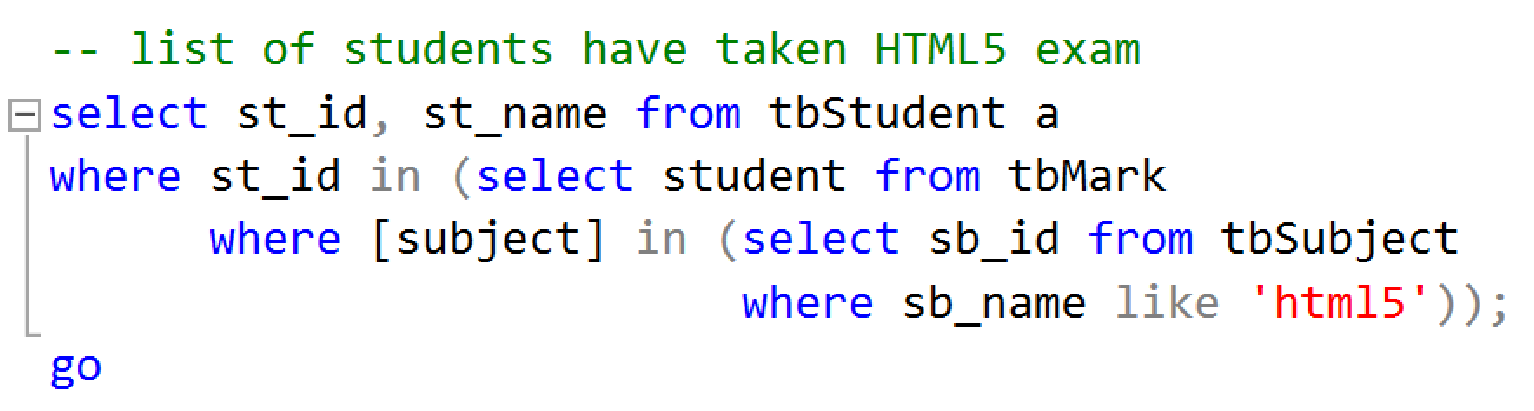
You cannot use DISTINCT keyword with subqueries that include GROUP BY.

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

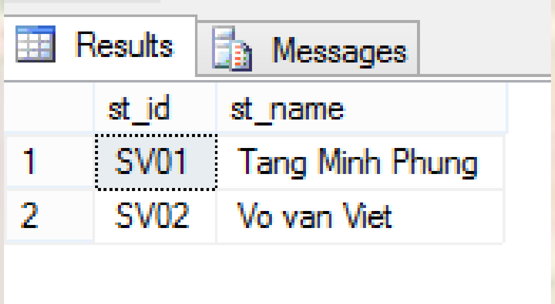
Nested Subqueries

- A subquery that is defined inside another subquery is called a nested subquery.

For example:



```
-- list of students have taken HTML5 exam
select st_id, st_name from tbStudent a
where st_id in (select student from tbMark
      where [subject] in (select sb_id from tbSubject
                  where sb_name like 'html5'));
go
```



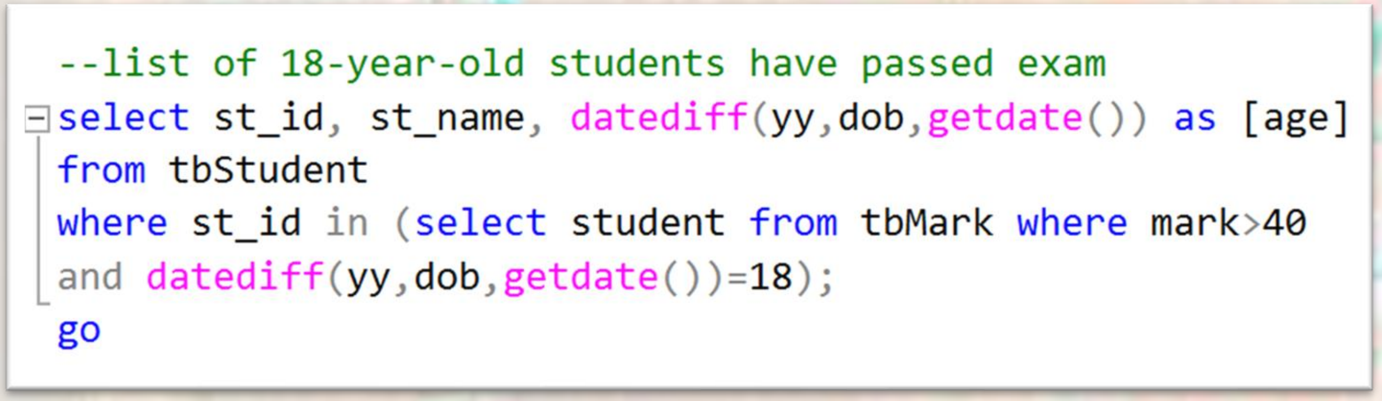
	st_id	st_name
1	SV01	Tang Minh Phung
2	SV02	Vo van Viet

Output of Nested Subqueries

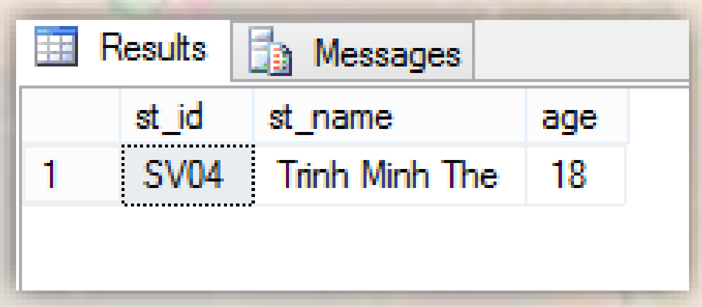
Correlated Queries

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

For example:



```
--list of 18-year-old students have passed exam
select st_id, st_name, datediff(yy,dob,getdate()) as [age]
from tbStudent
where st_id in (select student from tbMark where mark>40
and datediff(yy,dob,getdate())=18);
go
```



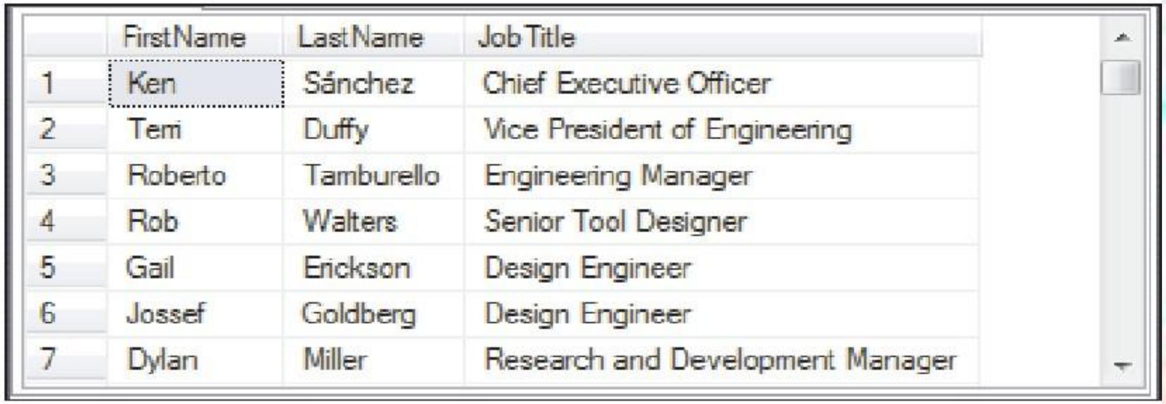
	st_id	st_name	age
1	SV04	Trinh Minh The	18

Outputof Correlated Queries

Joins

- 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.



	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

Output of Join

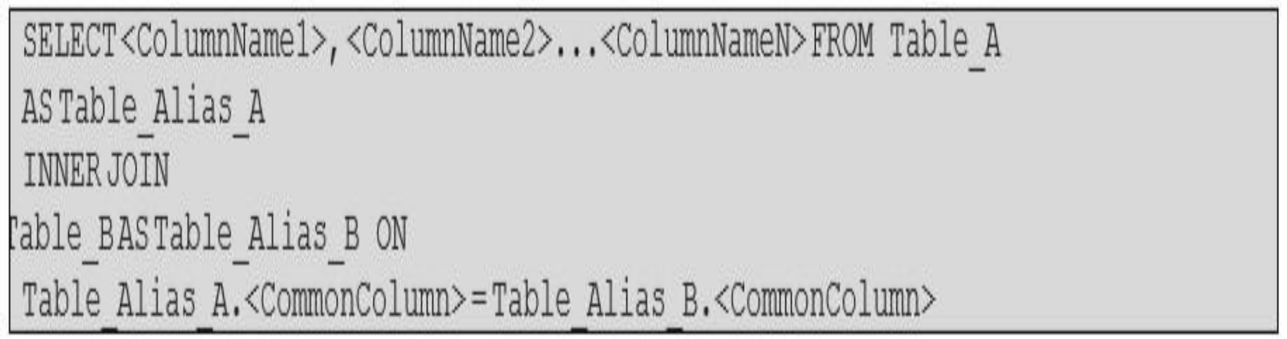
Three types of joins:

Inner Joins | Outer Joins | Self-Joins

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

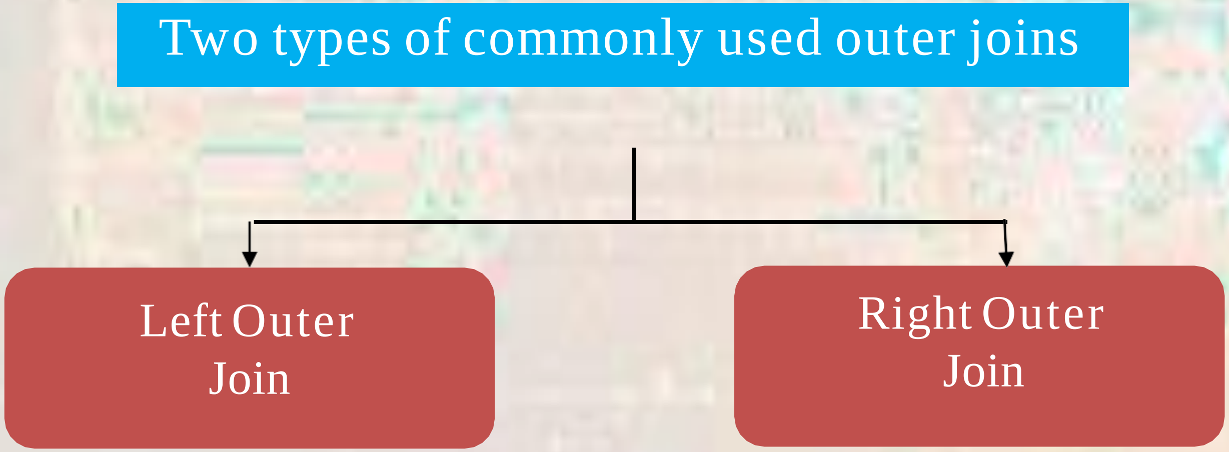
Following is the syntax of an inner join:



```
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>
```

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.



Two types of commonly used outer joins

Left Outer Join

Right Outer Join

Left Outer Join

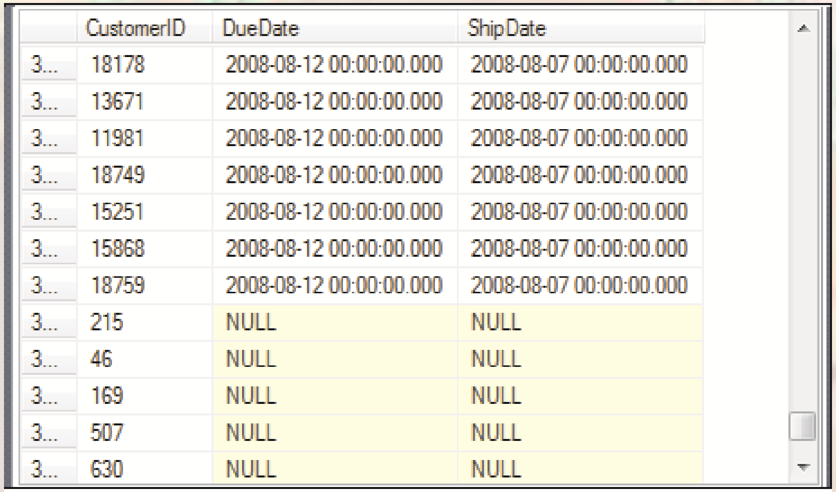
Returns all the records from the left table and only matching records from the right table.

- The syntax :

```
SELECT <ColumnName1>, <ColumnName2>...<ColumnNameN>
  FROM Table_A  AS  Alias_A  LEFT OUTER JOIN  Table_B  AS  Alias_B
     ON  Alias_A.<CommonColumn> = Alias_B.<CommonColumn>
```

- For example :

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



	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

Right Outer Join

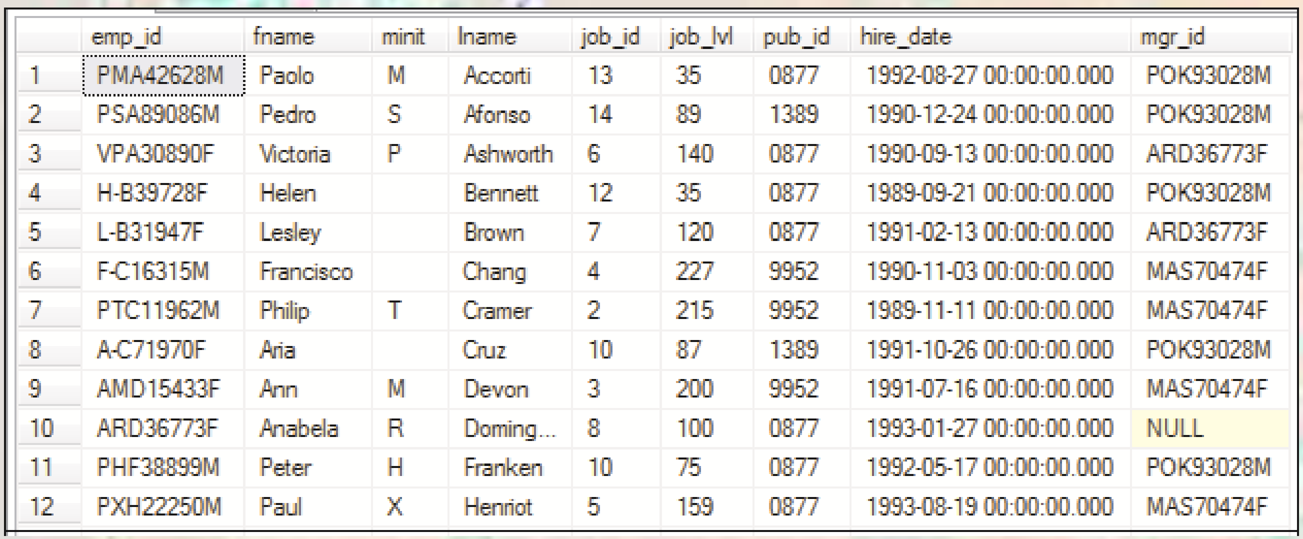
- Retrieves all the records from the second table in the join regardless of whether there is matching data in the first table or not.

Syntax:

```
SELECT <Column List>
  FROM Table_A  AS  Alias_A  RIGHT OUTER JOIN  Table_B  AS  Alias_B
     ON  Alias_A.<CommonColumn> = Alias_B.<CommonColumn>
```

Self-Join

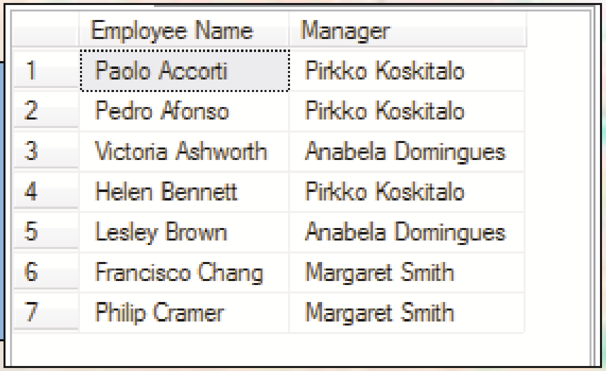
- A self-join is used to find records in a table that are related to other records in the same table.
- A table is joined to itself in a self-join.



	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

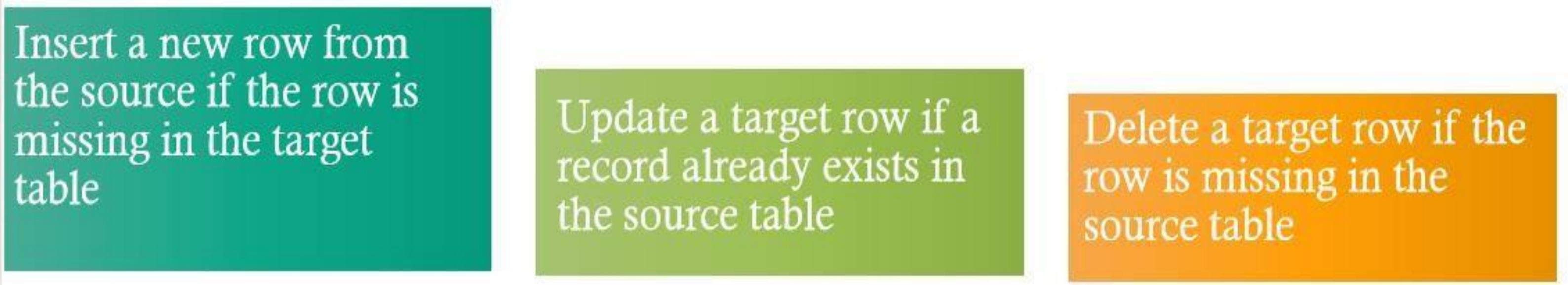
- For example:

```
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
```



	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

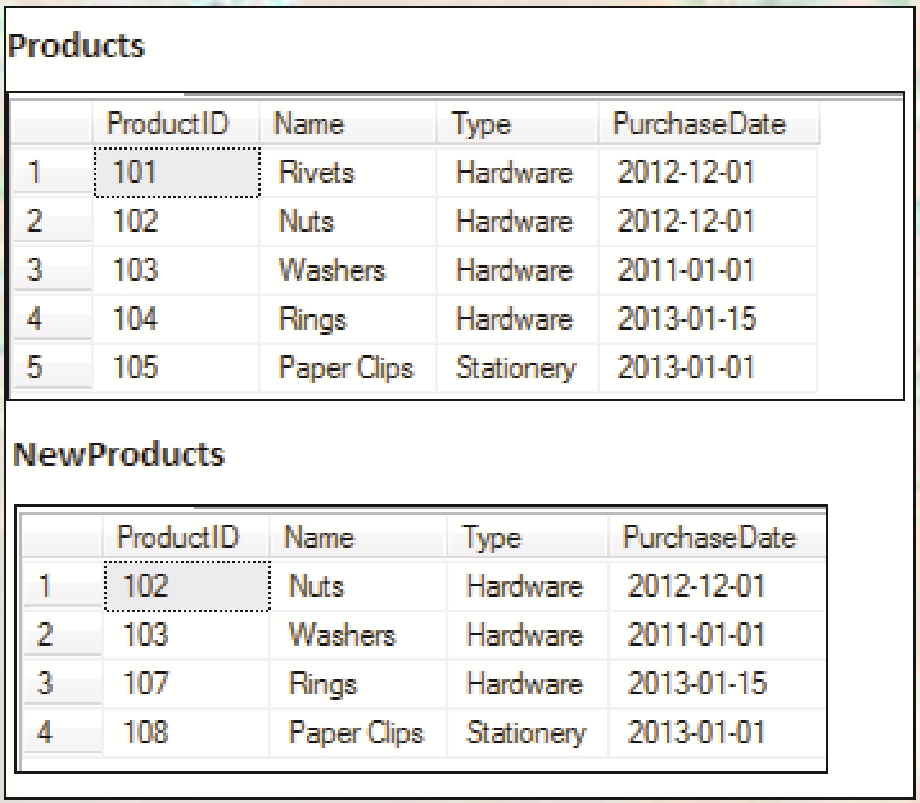
MERGE Statement



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

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



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

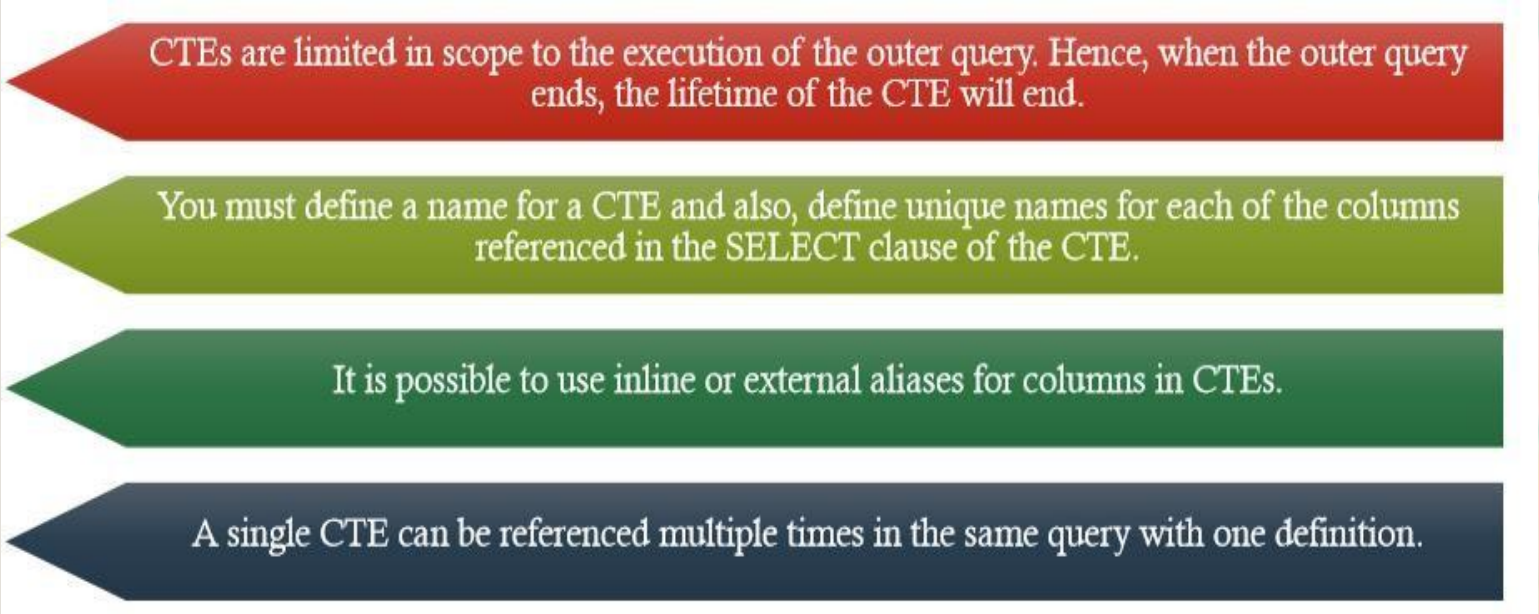
NewProducts

	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

Common Table Expressions (CTEs)

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



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

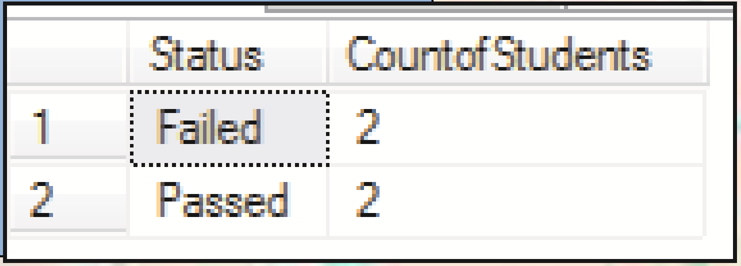
It is possible to use inline or external aliases for columns in CTEs.

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

Common Table Expressions (CTEs)

- 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
```



	Status	CountofStudents
1	Failed	2
2	Passed	2

Combining Data Using SET Operators

- SQL Server 2019 provides certain keywords, also called as operators, to combine data from multiple tables.

These operators are as follows:

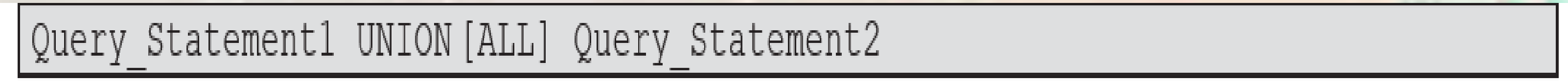
- UNION
- INTERSECT
- EXCEPT

UNION Operator 1-2

- Results from two different query statements can be combined into a single resultset using UNION operator.
- 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.
- By compatible data types, it means that it should be possible to convert the contents of one of the columns into another.

Following is the syntax of the UNION operator.



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

- If you include the ALL clause, all rows are included in the resultset including duplicate records.

UNION Operator 2-2

- For example: Union

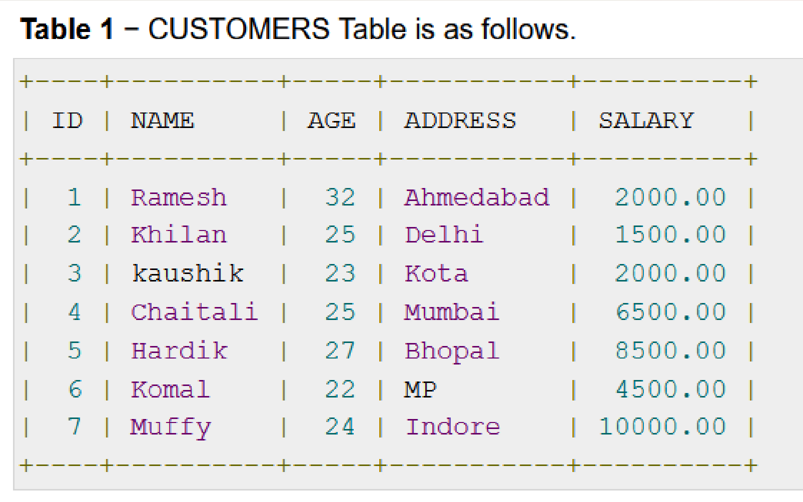


Table 1 – CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

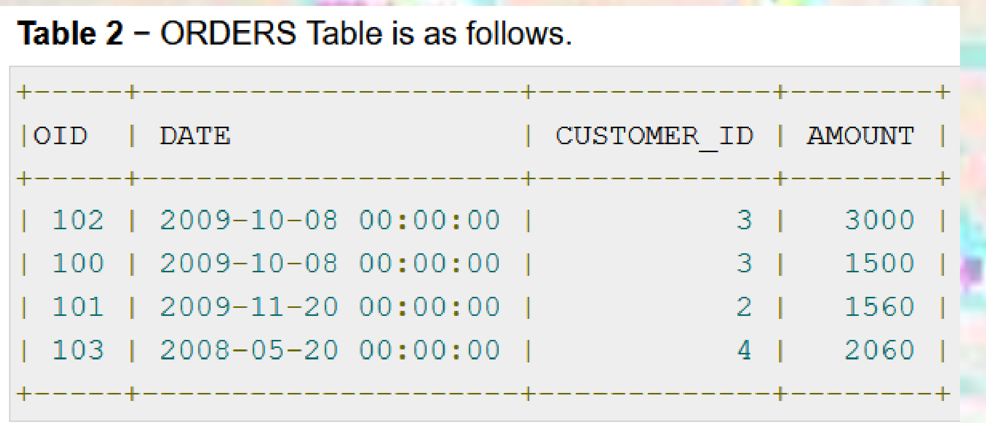
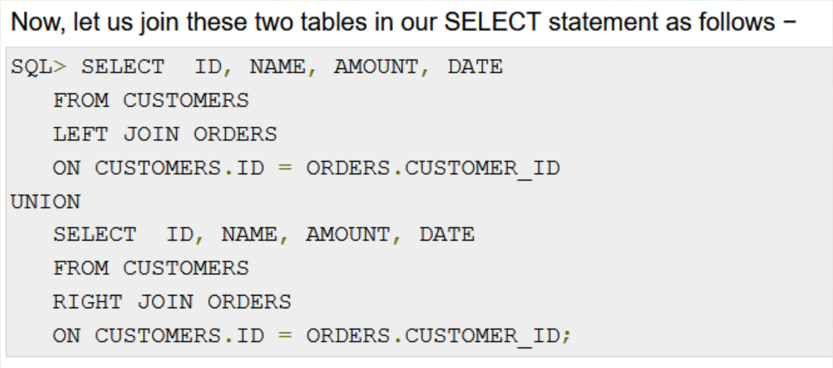


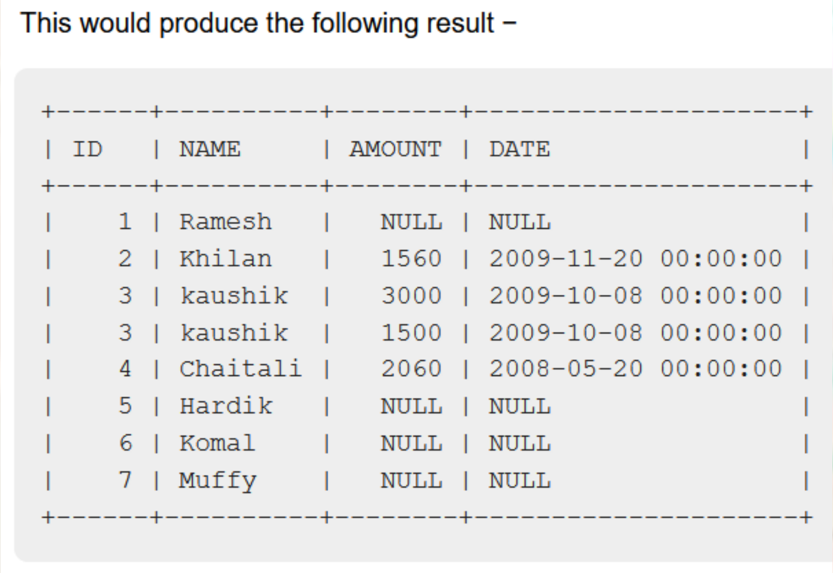
Table 2 – ORDERS Table is as follows.

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```



Now, let us join these two tables in our SELECT statement as follows –

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   LEFT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION
   SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   RIGHT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```



This would produce the following result –

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    1 | Ramesh   |   NULL | NULL                |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|    5 | Hardik   |   NULL | NULL                |
|    6 | Komal    |   NULL | NULL                |
|    7 | Muffy    |   NULL | NULL                |
+------+----------+--------+---------------------+
```

UNION Operator 2-3

- For example: Union All

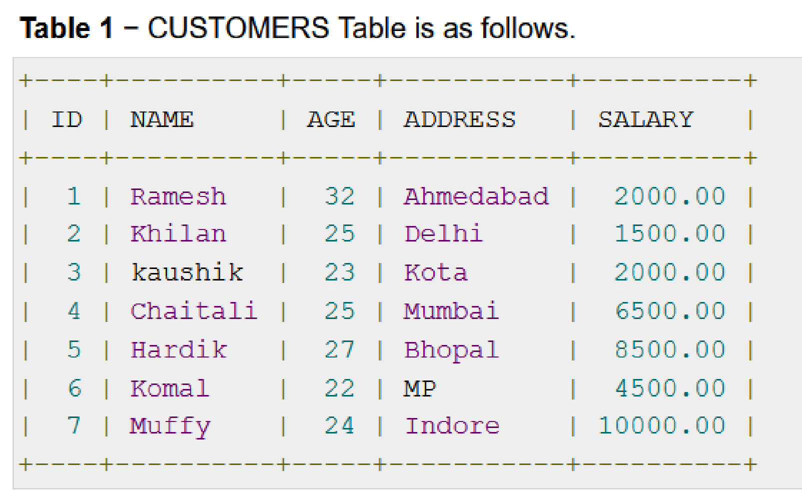


Table 1 – CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

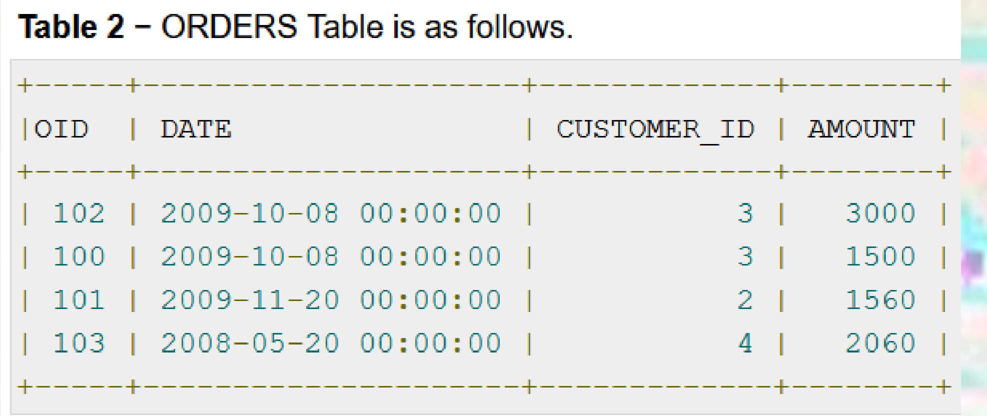
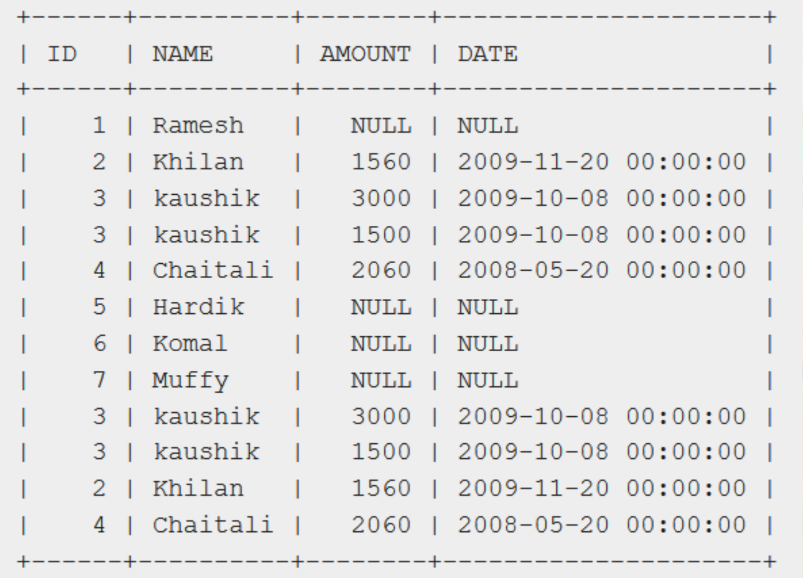
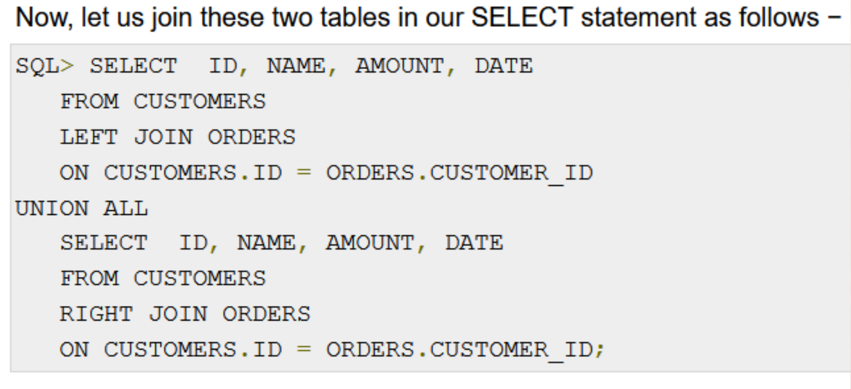


Table 2 – ORDERS Table is as follows.

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```



```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    1 | Ramesh   |   NULL | NULL                |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|    5 | Hardik   |   NULL | NULL                |
|    6 | Komal    |   NULL | NULL                |
|    7 | Muffy    |   NULL | NULL                |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```



Now, let us join these two tables in our SELECT statement as follows –

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   LEFT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION ALL
   SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   RIGHT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

INTERSECT Operator

- 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 basic rules for using INTERSECT are as follows:

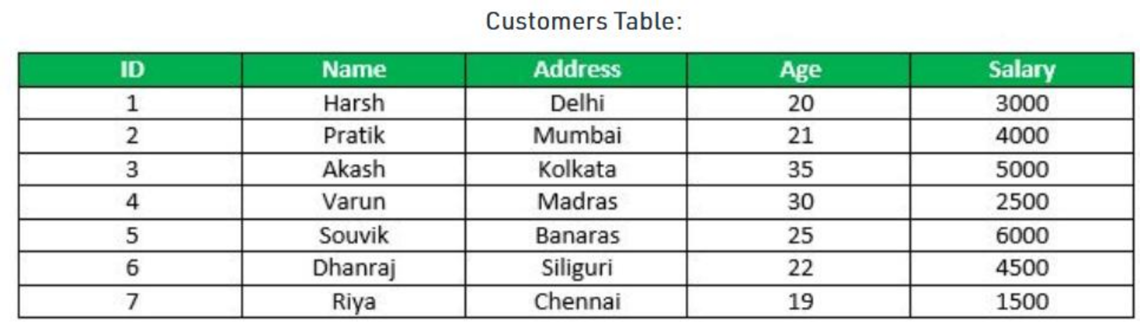
- Number of columns and order in which they are given must be same in both queries.
- Data types of the columns being used must be compatible.

Syntax:

```
Query_statement1
INTERSECT
Query_statement2
```

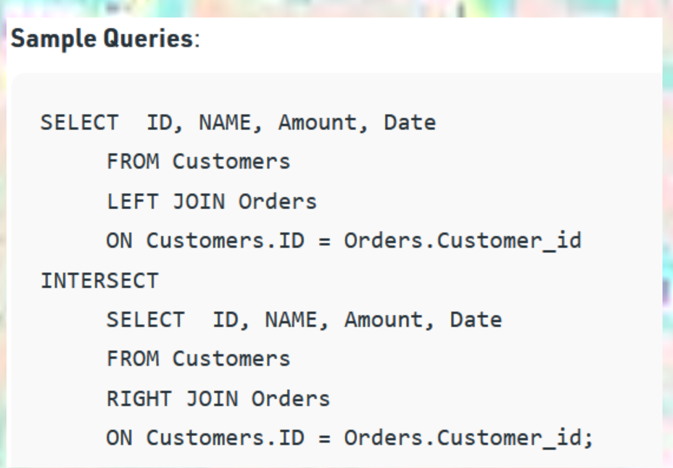
INTERSECT Operator

- For example:



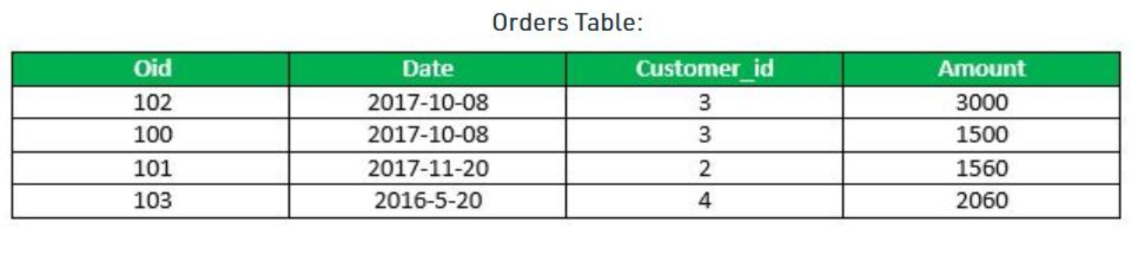
Customers Table:

ID	Name	Address	Age	Salary
1	Harsh	Delhi	20	3000
2	Pratik	Mumbai	21	4000
3	Akash	Kolkata	35	5000
4	Varun	Madras	30	2500
5	Souvik	Banaras	25	6000
6	Dhanraj	Siliguri	22	4500
7	Riya	Chennai	19	1500



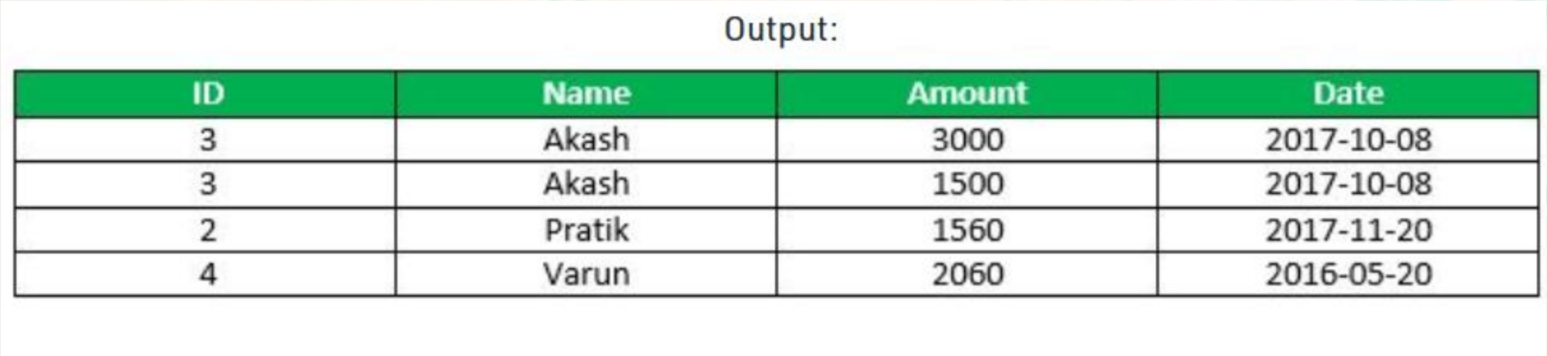
Sample Queries:

```
SELECT  ID, NAME, Amount, Date
     FROM Customers
     LEFT JOIN Orders
     ON Customers.ID = Orders.Customer_id
INTERSECT
     SELECT  ID, NAME, Amount, Date
     FROM Customers
     RIGHT JOIN Orders
     ON Customers.ID = Orders.Customer_id;
```



Orders Table:

Oid	Date	Customer_id	Amount
102	2017-10-08	3	3000
100	2017-10-08	3	1500
101	2017-11-20	2	1560
103	2016-5-20	4	2060



Output:

ID	Name	Amount	Date
3	Akash	3000	2017-10-08
3	Akash	1500	2017-10-08
2	Pratik	1560	2017-11-20
4	Varun	2060	2016-05-20

EXCEPT Operator

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

Syntax:

```
Query_statement1
EXCEPT
Query_statement2
```

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

```
SELECT ProductId FROM Product
EXCEPT
SELECT ProductId FROM SalesOrderDetail
```

EXCEPT Operator

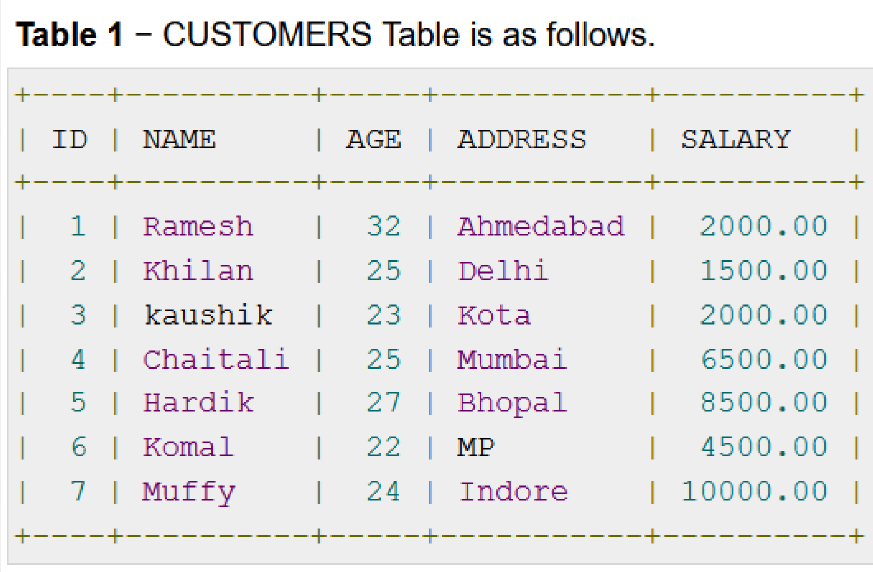


Table 1 – CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

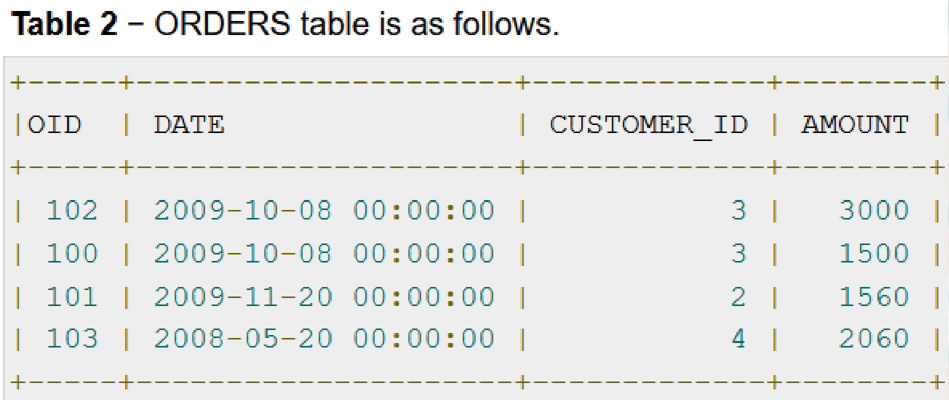
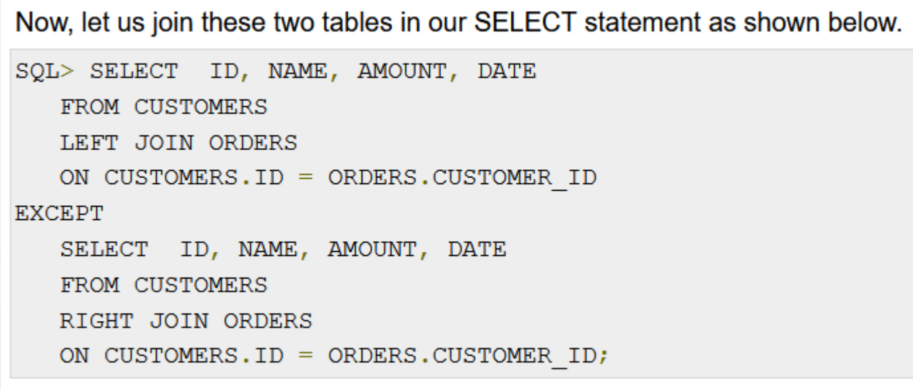


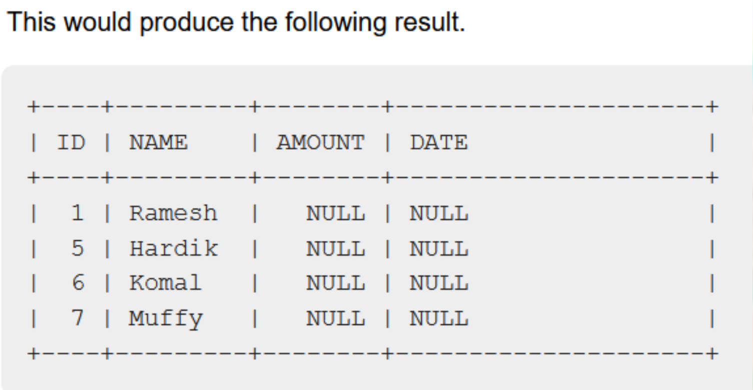
Table 2 – ORDERS table is as follows.

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```



Now, let us join these two tables in our SELECT statement as shown below.

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   LEFT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
EXCEPT
   SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   RIGHT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```



This would produce the following result.

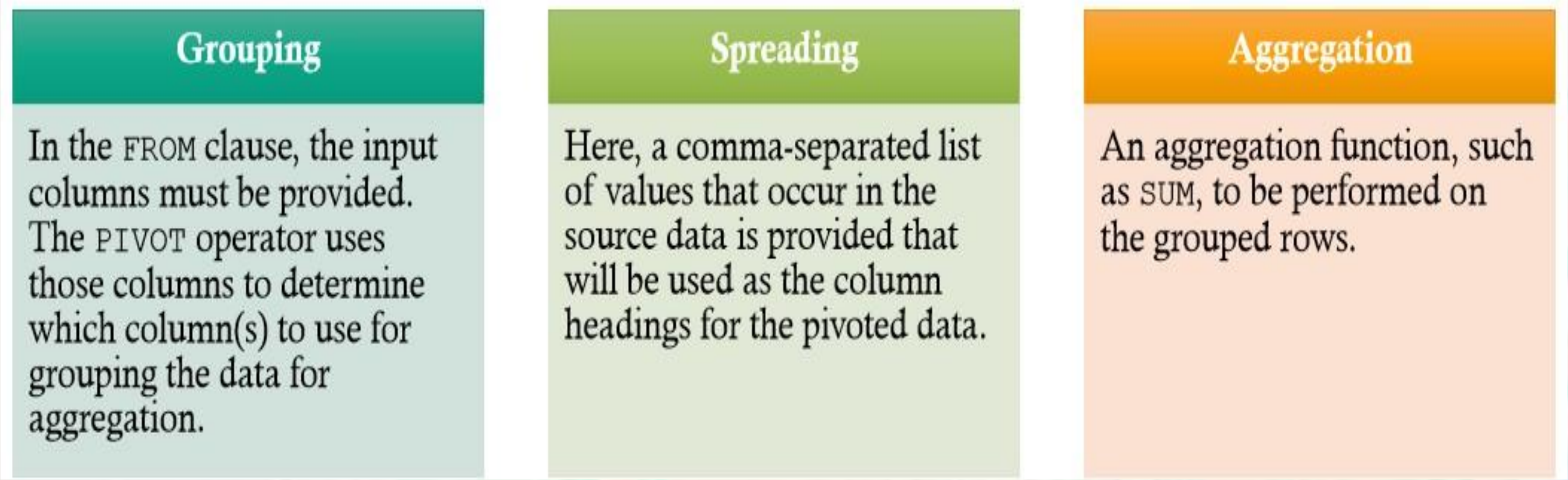
```
+----+---------+--------+---------------------+
| ID | NAME    | AMOUNT | DATE                |
+----+---------+--------+---------------------+
|  1 | Ramesh  |   NULL | NULL                |
|  5 | Hardik  |   NULL | NULL                |
|  6 | Komal   |   NULL | NULL                |
|  7 | Muffy   |   NULL | NULL                |
+----+---------+--------+---------------------+
```

Pivoting and Grouping Set Operations

- The process of transforming data from a row-based orientation to a column-based orientation is calledpivoting.

- The PIVOT and UNPIVOT operators of SQL Server help to change the orientation of data from column-oriented to row-oriented and vice versa.

PIVOT 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.

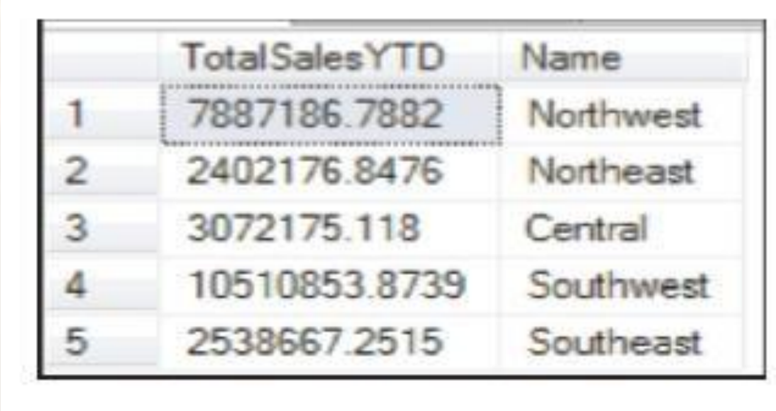
Spreading

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.

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



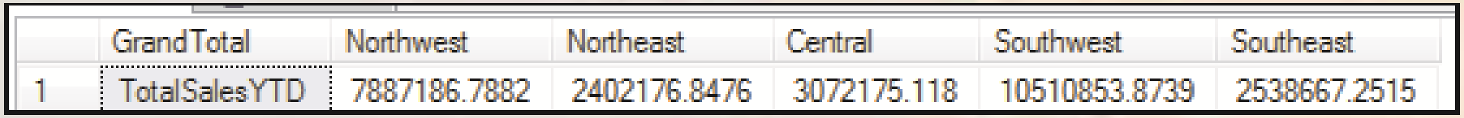
	TotalSalesYTD	Name
1	7887186.7882	Northwest
2	2402176.8476	Northeast
3	3072175.118	Central
4	10510853.8739	Southwest
5	2538667.2515	Southeast

Grouping without PIVOT

PIVOT Operator

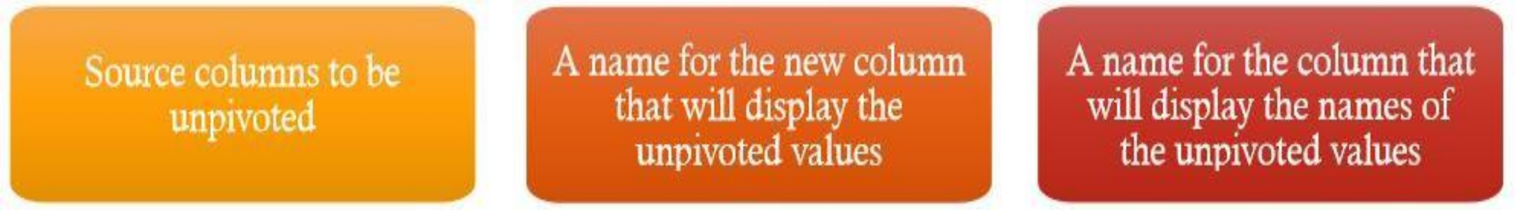
- The top 5 year to date sales along with territory names grouped by territory names are displayed.
- The same query is rewritten in the following code snippet using a PIVOT so that the data is transformed from a row-based orientation to a column-based orientation:

```
-- 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;
```



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

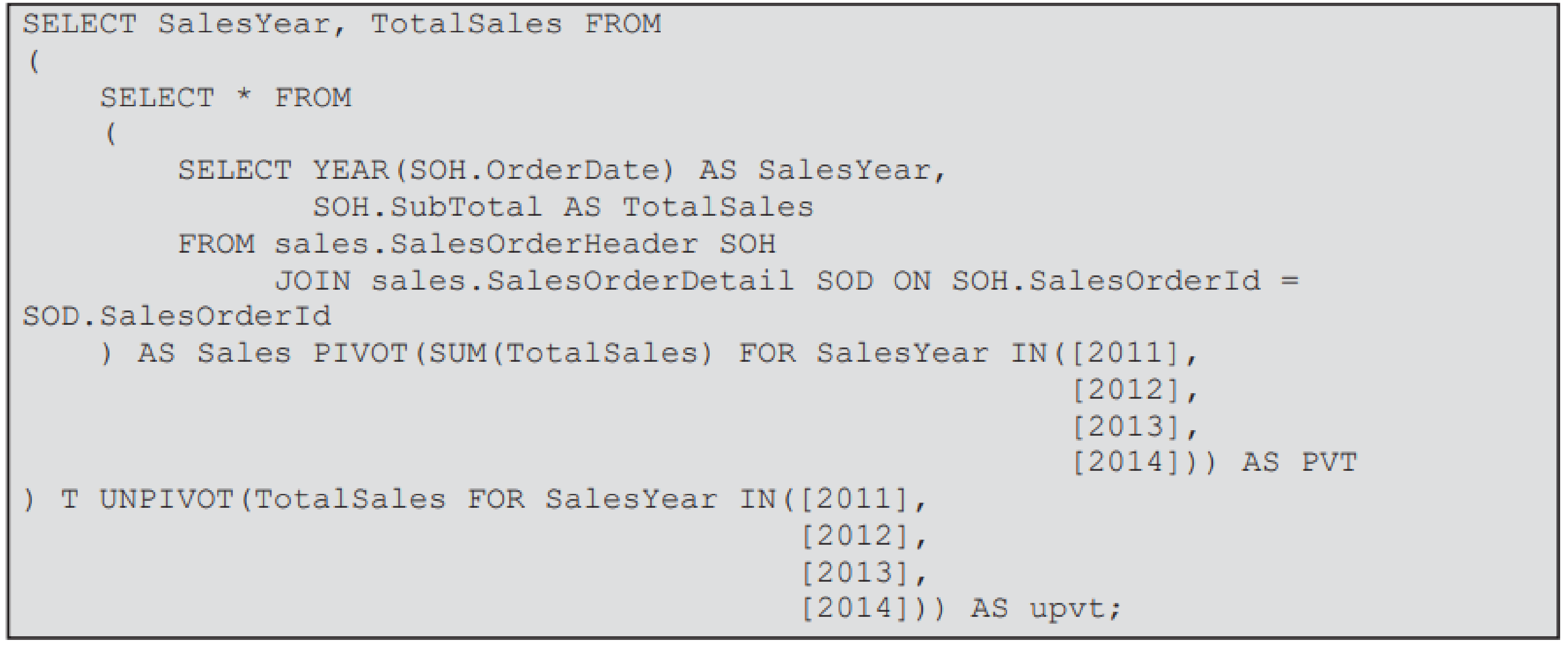
UNPIVOT Operator



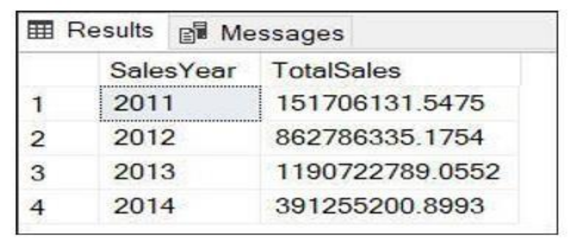
Source columns to be unpivoted

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



```
SELECT SalesYear, TotalSales FROM
(
    SELECT * FROM
    (
        SELECT YEAR(SOH.OrderDate) AS SalesYear,
               SOH.SubTotal AS TotalSales
        FROM sales.SalesOrderHeader SOH
             JOIN sales.SalesOrderDetail SOD ON SOH.SalesOrderId =
SOD.SalesOrderId
    ) AS Sales PIVOT(SUM(TotalSales) FOR SalesYear IN([2011],
                                                       [2012],
                                                       [2013],
                                                       [2014])) AS PVT
) T UNPIVOT(TotalSales FOR SalesYear IN([2011],
                                        [2012],
                                        [2013],
                                        [2014])) AS upvt;
```



	SalesYear	TotalSales
1	2011	151706131.5475
2	2012	862786335.1754
3	2013	1190722789.0552
4	2014	391255200.8993

Output forUNPIVOT

Summary

- The GROUP BY clause and aggregate functions enable to group and/or aggregate data together in order to present summarized information.
- Spatial aggregate functions were first introduced in SQL Server 2012 and are supported in SQL Server 2019 as well.
- 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.