Module 9: Grouping and Aggregating Data

Contents:

Module Overview

In addition to row-at-a-time queries, you may need to summarize data to analyze it. Microsoft® SQL Server® provides built-in functions that can aggregate, or summarize, information across multiple rows. In this module, you will learn how to use aggregate functions. You will also learn how to use the GROUP BY and HAVING clauses to break up the data into groups for summarizing, and to filter the resulting groups.

Objectives

After completing this lesson, you will be able to:

- List the built-in aggregate functions provided by SQL Server.
- Write queries that use aggregate functions in a SELECT list to summarize all the rows in an input set.
- Describe the use of the DISTINCT option in aggregate functions.
- Write queries using aggregate functions that handle the presence of NULLs in source data.

Lesson 1: Using Aggregate Functions

In this lesson, you will learn how to use built-in functions to aggregate, or summarize, data in multiple rows. SQL Server provides functions such as SUM, MAX, and AVG to perform calculations that take multiple values and return a single result.

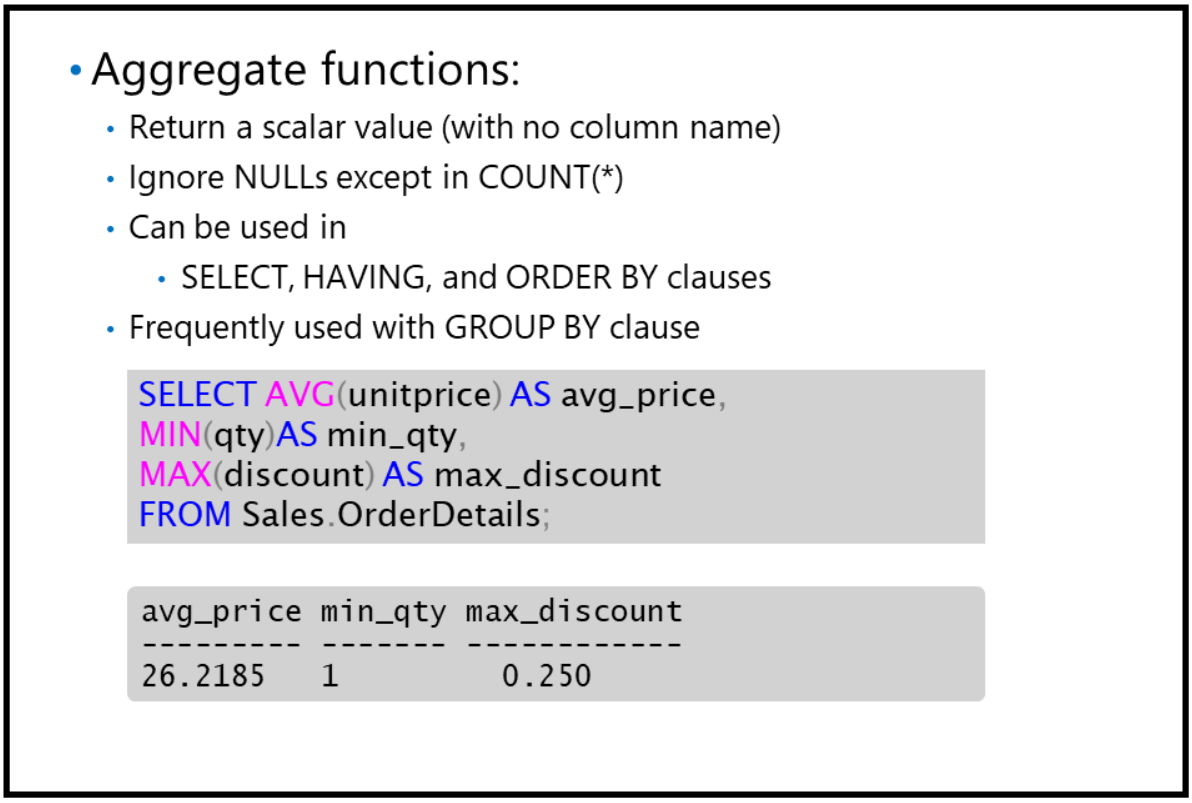
Lesson Objectives

After completing this lesson, you will be able to:

- List the built-in aggregate functions provided by SQL Server.
- Write queries that use aggregate functions in a SELECT list to summarize all the rows in an input set.

- Describe the use of the DISTINCT option in aggregate functions.
- Write queries using aggregate functions that handle the presence of NULLs in source data.

Working with Aggregate Functions



So far in this course, you have learned how to operate on a row at a time, using a WHERE clause to filter rows, adding computed columns to a SELECT list, and processing across columns, but within each row.

You may also need to perform analysis across rows, such as counting rows that meet your criteria, or summarizing total sales for all orders. To accomplish this, you will use aggregate functions capable of operating on multiple rows simultaneously.

Many aggregate functions are provided in SQL Server. In this course, you will learn about common functions such as SUM, MIN, MAX, AVG, and COUNT.

When working with aggregate functions, you need to consider the following:

- Aggregate functions return a single (scalar) value and can be used in SELECT statements where a single expression is used, such as SELECT, HAVING, and ORDER BY clauses.
- Aggregate functions ignore NULLs, except when using COUNT(*). You will learn more about this later in the lesson.
- 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, as in the slide above. You will learn more about GROUP BY in the next lesson.

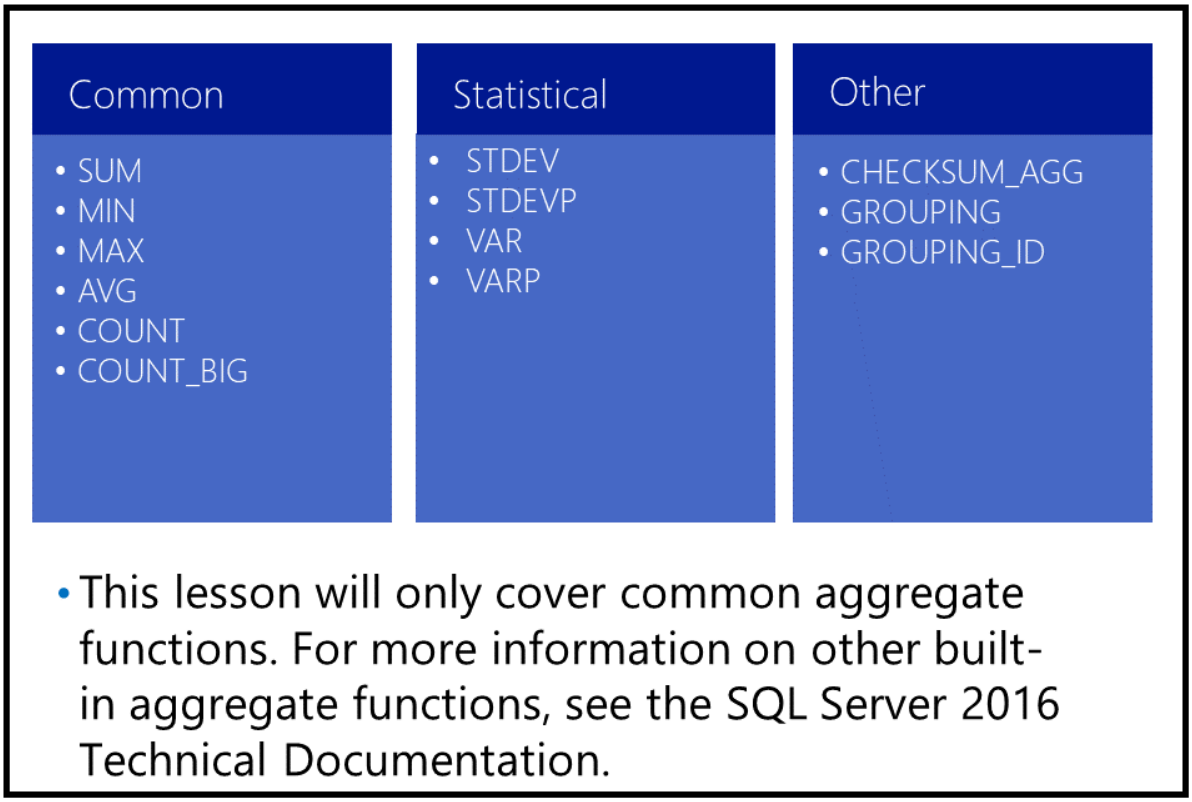
To extend beyond the built-in functions, SQL Server provides a mechanism for user-defined aggregate functions via the .NET Common Language Runtime (CLR).

For more information on other built-in aggregate functions, see Microsoft Docs:

Aggregate Functions (Transact-SQL)

http://aka.ms/wq6lku

Built-in Aggregate Functions



SQL Server provides many built-in aggregate functions. Commonly used functions include:

Function Name	Syntax	Description
SUM	SUM(<expression>)	Totals all the non-NULL numeric values in a column.
AVG	AVG(<expression>)	Averages all the non-NULL numeric values in a column (sum/count).
MIN	MIN(<expression>)	Returns the largest number, earliest date/time, or first-occurring string (according to collation sort rules).
MAX	MAX(<expression>)	Returns the largest number, latest date/time, or last-occurring string (according to collation sort rules).
COUNT or COUNT_BIG	COUNT(*) or COUNT(<expression>)	With (*), counts all rows, including those with NULL values. When a column is specified as <expression>, returns count of non-NULL rows for that column. COUNT returns an int; COUNT_BIG returns a big_int.

This lesson only covers common aggregate functions. For information on other built-in aggregate functions, see Microsoft Docs:

Aggregate Functions (Transact-SQL)

http://aka.ms/wq6lku

Aggregate Example

```
SELECT  AVG(unitprice) AS avg_price,
                MIN(qty)AS min_qty,
                MAX(discount) AS max_discount
FROM Sales.OrderDetails;
```

Note that the above example does not use a GROUP BY clause. Therefore, all rows from the Sales.OrderDetails table will be summarized by the aggregate formulas in the SELECT clause.

The results:

```
avg_price min_qty max_discount
--------- ------- ------------
26.2185   1       0.250
```

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

Partial Aggregate Error

```
SELECT orderid, AVG(unitprice) AS avg_price, MIN(qty)AS min_qty, MAX(discount) AS
max_discount
FROM Sales.OrderDetails;
```

This returns:

```
Msg 8120, Level 16, State 1, Line 1
Column 'Sales.OrderDetails.orderid' is invalid in the select list because it is not
contained in either an aggregate function or the GROUP BY clause.
```

Since our example is not using a GROUP BY clause, the query treats all rows as a single group. Therefore, all columns must be used as inputs to aggregate functions. Removing orderid from the previous example will prevent the error.

In addition to numeric data, such as the price and quantities in the previous example, aggregate expressions can also summarize date, time, and character data. The following examples show the use of aggregates with dates and characters:

Aggregating Character Data

```
SELECT MIN(companyname) AS first_customer, MAX(companyname) AS last_customer
FROM Sales.Customers;
```

Returns:

```
first_customer last_customer
-------------- --------------
Customer AHPOP Customer ZRNDE
```

Other functions may coexist with aggregate functions.

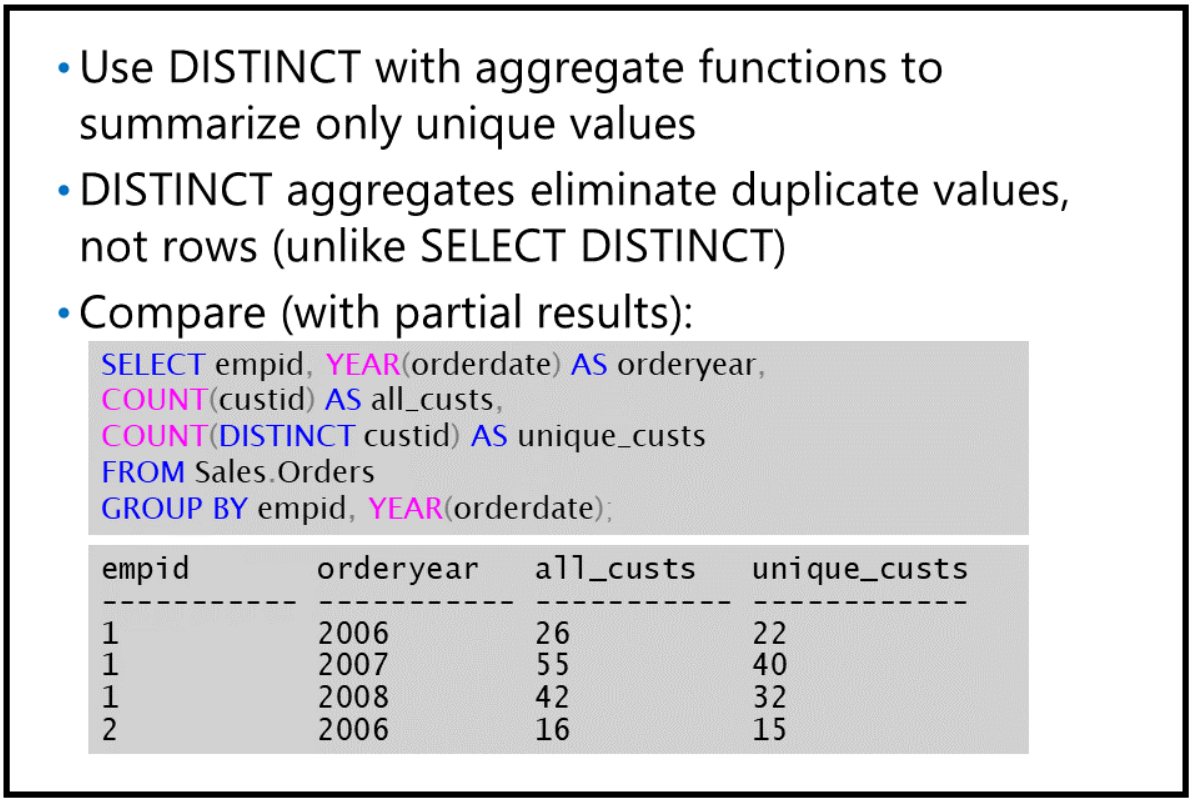
Aggregating with Functions

```
SELECT MIN(YEAR(orderdate))AS earliest, MAX(YEAR(orderdate)) AS latest
FROM Sales.Orders;
```

Returns:

```
earliest latest
-------- -------
2006     2008
```

Using DISTINCT with Aggregate Functions



Earlier in this course, you learned about the use of DISTINCT in a SELECT clause to remove duplicate rows. When used with an aggregate function, DISTINCT removes duplicate values from the input column before computing the summary value. This is useful when summarizing unique occurrences of values, such as customers in the TSQL orders table.

Summarizing Distinct Values

```
SELECT empid, YEAR(orderdate) AS orderyear,
        COUNT(custid) AS all_custs,
        COUNT(DISTINCT custid) AS unique_custs
FROM Sales.Orders
GROUP BY empid, YEAR(orderdate);
```

Note that the above example uses a GROUP BY clause. GROUP BY will be covered in the next lesson. It is used here as a useful example for comparing DISTINCT and non-DISTINCT aggregate functions.

This returns, in part:

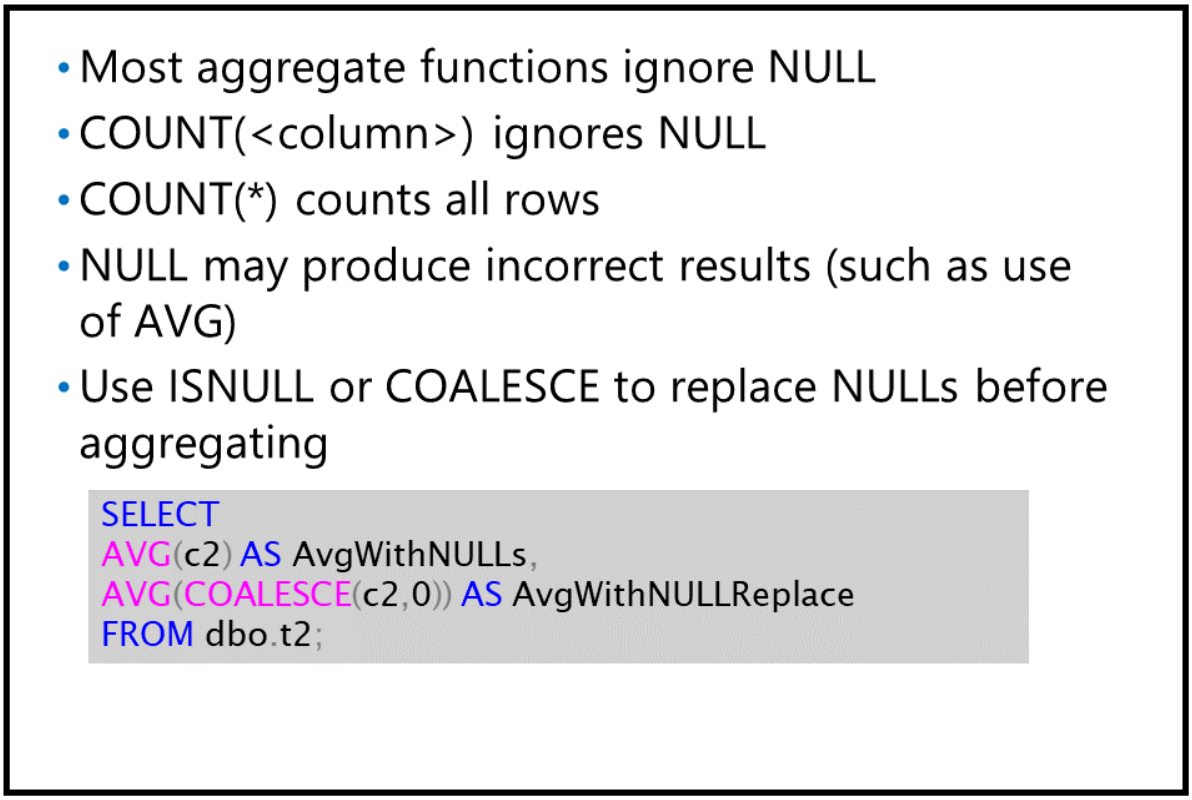
```
empid orderyear all_custs unique_custs
------------- --------- ------------
1     2006      26        22
1     2007      55        40
1     2008      42        32
2     2006      16        15
2     2007      41        35
```

2	2008	39	34
3	2006	18	16
3	2007	71	46
3	2008	38	30

Note the difference in each row between the COUNT of custid (in column 3) and the DISTINCT COUNT in column 4. Column 3 simply returns all rows except those containing NULL. Column 4 excludes duplicate custids (repeat customers) and returns a count of unique customers, answering the question: “How many customers per employee?”

Question: Could you accomplish the same output with the use of SELECT DISTINCT?

Using Aggregate Functions with NULL



As you have learned in this course, it is important to be aware of the possible presence of NULLs in your data, and of how NULL interacts with T-SQL query components. This is also true with aggregate expressions. There are a few considerations to be aware of:

- With the exception of COUNT used with the (*) option, T-SQL aggregate functions ignore NULLs. This means, for example, that a SUM function will add only non-NULL values. NULLs do not evaluate to zero.
- The presence of NULLs in a column may lead to inaccurate computations for AVG, which will sum only populated rows and divide that sum by the number of non-NULL rows. There may be a difference in results between AVG(<column>) and (SUM(<column>)/COUNT(*)).

For example, the following table named t1:

C1	C2
1	NULL
2	10
3	20
4	30
5	40
6	50

Aggregating NULL Example

```
SELECT SUM(c2) AS sum_nonnulls,
        COUNT(*)AS count_all_rows,
        COUNT(c2)AS count_nonnulls,
        AVG(c2) AS [avg],
        (SUM(c2)/COUNT(*))AS arith_avg
FROM t1;
```

The result:

```
sum_nonnulls count_all_rows count_nonnulls avg arith_avg
------------ -------------- -------------- --- ---------
150          6              5              30  25
```

If you need to summarize all rows, whether NULL or not, consider replacing the NULLs with another value that can be used by your aggregate function.

The following example replaces NULLs with 0 before calculating an average. The table named t2 contains the following rows:

```
c1          c2
----------- -----------
1           1
2           10
3           1
4           NULL
5           1
6           10
7           1
8           NULL
9           1
10          10
```

```
11          1
12          10
```

Replace NULLs with Zeros Example

```
SELECT AVG(c2) AS AvgWithNULLs, AVG(COALESCE(c2,0)) AS AvgWithNULLReplace
FROM dbo.t2;
```

This returns the following results, with a warning message:

```
AvgWithNULLs AvgWithNULLReplace
------------ -----------------
4            3

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

> **Note:** This example cannot be executed against the sample database used in this course. You will find a script to create the table in the upcoming demonstration.

Demonstration: Using Aggregate Functions

In this demonstration, you will see how to use built-in aggregate functions.

Demonstration Steps

Use Built-in Aggregate Functions

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **D:\Demofiles\Mod09\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. Start SQL Server Management Studio and connect to the **MIA-SQL** database instance using Windows authentication.
5. On the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, navigate to the **D:\Demofiles\Mod09\Demo** folder, click **Demo.ssmssln**, and then click **Open**.
7. In Solution Explorer, expand **Queries**, and then double-click **11 - Demonstration A.sql**.
8. Select the code under the comment **Step 1**, and then click **Execute**.
9. Select the code under the comment **Step 2a**, and then click **Execute**.

10. Select the code under the comment **Step 2b**, and then click **Execute**.
11. Select the code under the comment **Step 2c**, and then click **Execute**.
12. Select the code under the comment **Step 2d**, and then click **Execute**.
13. Select the code under the comment **Step 2e**, and then click **Execute**.
14. Select the code under the comment **Step 2f**, and then click **Execute**.
15. Select the code under the comment **Step 2g**, and then click **Execute**.
16. Select the code under the comment **Step 3a**, and then click **Execute**.
17. Select the code under the comment **Step 3b**, and then click **Execute**.
18. Select the code under the comment **Step 3c**, and then click **Execute**.
19. Select the code under the comment **Step 3d**, and then click **Execute**.
20. Select the code under the comment **Step 3e**, and then click **Execute**.
21. Select the code under the comment **Step 3f**, and then click **Execute**.
22. Select the code under the comment **Step 3g**, and then click **Execute**.
23. Select the code under the comment **Step 3h**, and then click **Execute**.
24. Select the code under the comment **Step 3i**, and then click **Execute**.
25. Select the code under the comment **Step 4**, and then click **Execute**.
26. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Discovery

You have the following query:
SELECT COUNT(*) AS RecordCount
FROM Sales.Products;
There are 250 records in the Products table. How many rows will be returned by this query?

Show solution Reset

One. When you use an aggregate function without a GROUP BY clause, all rows are aggregated into a single result.

Lesson 2: Using the GROUP BY Clause

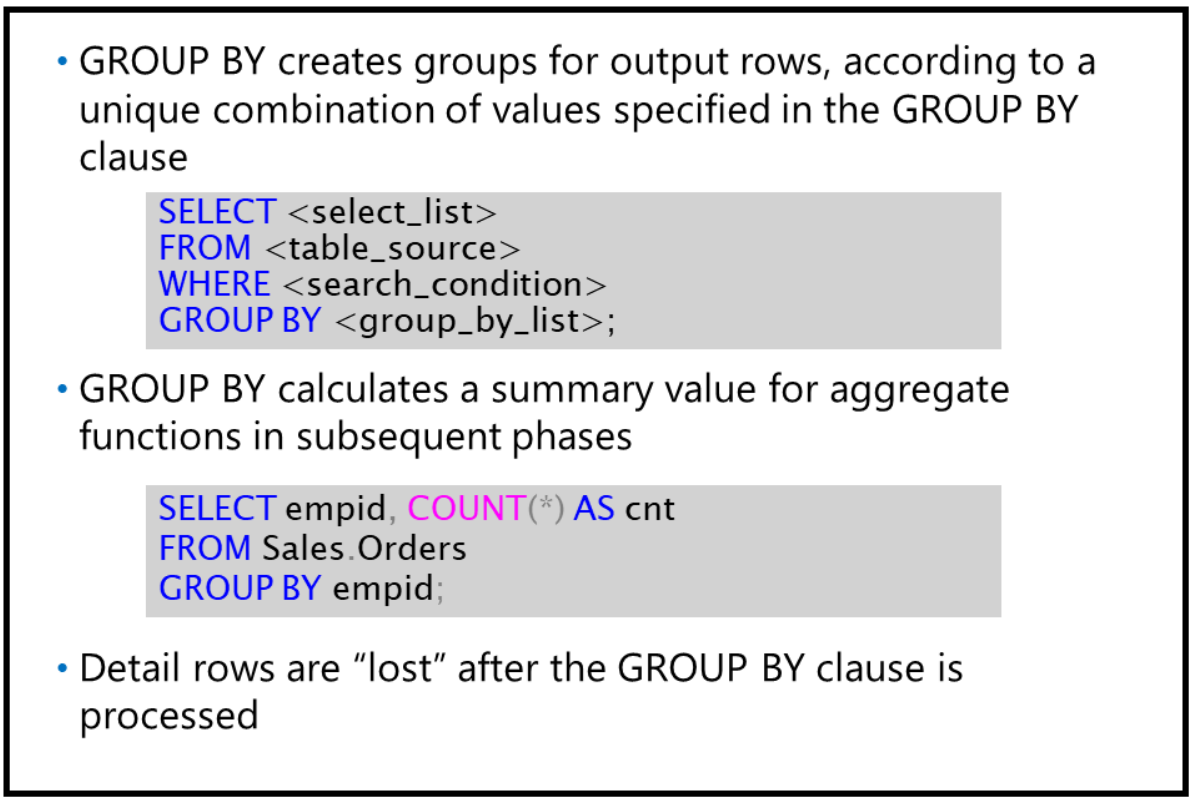
While aggregate functions are useful for analysis, you may wish to arrange your data into subsets before summarizing it. In this lesson, you will learn how to accomplish this using the GROUP BY clause.

Lesson Objectives

After completing this lesson, you will be able to:

- Write queries that separate rows into groups using the GROUP BY clause.
- Describe the role of the GROUP BY clause in the logical order of operations for processing a SELECT statement.
- Write SELECT clauses that reflect the output of a GROUP BY clause.
- Use GROUP BY with aggregate functions.

Using the GROUP BY Clause



As you have learned, when your SELECT statement is processed, after the FROM clause and WHERE clause (if present) have been evaluated, a virtual table is created. The contents of the virtual table are now available for further processing. You can use the GROUP BY clause to subdivide the results of the preceding query phases into groups of rows.

GROUP BY Syntax

```
GROUP BY <value1> [, <value2>, …]
```

GROUP BY creates groups and places rows into each group as determined by unique combinations of the elements specified in the clause.

GROUP BY Snippet

```
FROM SalesOrders
GROUP BY empid;
```

Once the GROUP BY clause has been processed and rows have been associated with a group, subsequent phases of the query must aggregate any elements of the source rows that do not appear in the GROUP BY list. This will have an impact on how you write your SELECT and HAVING clauses.

To see the results of the GROUP BY clause, you will need to add a SELECT clause.

GROUP BY Example

```
SELECT empid, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY empid;
```

The result:

```
empid cnt
----- -----
1     123
2     96
3     127
4     156
5     42
6     67
7     72
8     104
9     43
(9 row(s) affected)
```

To learn more about GROUP BY, see SELECT - GROUP BY - Transact SQL in Microsoft Docs:

SELECT - GROUP BY - Transact-SQL

http://aka.ms/ro266s

GROUP BY and the Logical Order of Operations

Logical Order	Phase	Comments
5	SELECT	
1	FROM	
2	WHERE	
3	GROUP BY	Creates groups
4	HAVING	Operates on groups
6	ORDER BY	

- If a query uses GROUP BY, all subsequent phases operate on the groups, not source rows
- HAVING, SELECT, and ORDER BY must return a single value per group
- All columns in SELECT, HAVING, and ORDER BY must appear in the GROUP BY clause or be inputs to aggregate expressions

A common obstacle to becoming comfortable with using GROUP BY in SELECT statements is understanding why the following type of error message occurs:

```
Msg 8120, Level 16, State 1, Line 2
Column <column_name> is invalid in the select list because it is not contained in either an
aggregate function or the GROUP BY clause.
```

A review of the logical order of operations during query processing will help clarify this issue.

As mentioned earlier in the course, the SELECT clause is not processed until after the FROM, WHERE, GROUP BY, and HAVING clauses (if present) are processed. When discussing the use of GROUP BY, it is important to remember that not only does GROUP BY precede SELECT, but it also replaces the results of the FROM and WHERE clauses with its own results. The final outcome of the query will only return one row per qualifying group (if a HAVING clause is present). Therefore, any operations performed after GROUP BY, including SELECT, HAVING, and ORDER BY, are performed on the groups, not the original detail rows. Columns in the SELECT list, for example, must return a scalar value per group. This may include the column(s) being grouped on, or aggregate functions being performed on, each group.

GROUP BY Example

```
SELECT empid, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY empid;
```

This returns:

```
empid count
----- -----
1     123
2     96
3     127
4     156
5     42
6     67
7     72
8     104
9     43
```


Missing GROUP BY Value

```
SELECT empid, orderdate, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY empid;
```


This returns:

```
Msg 8120, Level 16, State 1, Line 1
Column 'Sales.Orders.orderdate' is invalid in the select list because it is not contained in
either an aggregate function or the GROUP BY clause.
```

Correct GROUP BY Example

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY empid, YEAR(orderdate)
ORDER BY empid, YEAR(orderdate);
```


This returns (in part):

```
empid orderyear count
----- --------- -----
1     2006      26
1     2007      55
1     2008      42
2     2006      16
2     2007      41
```

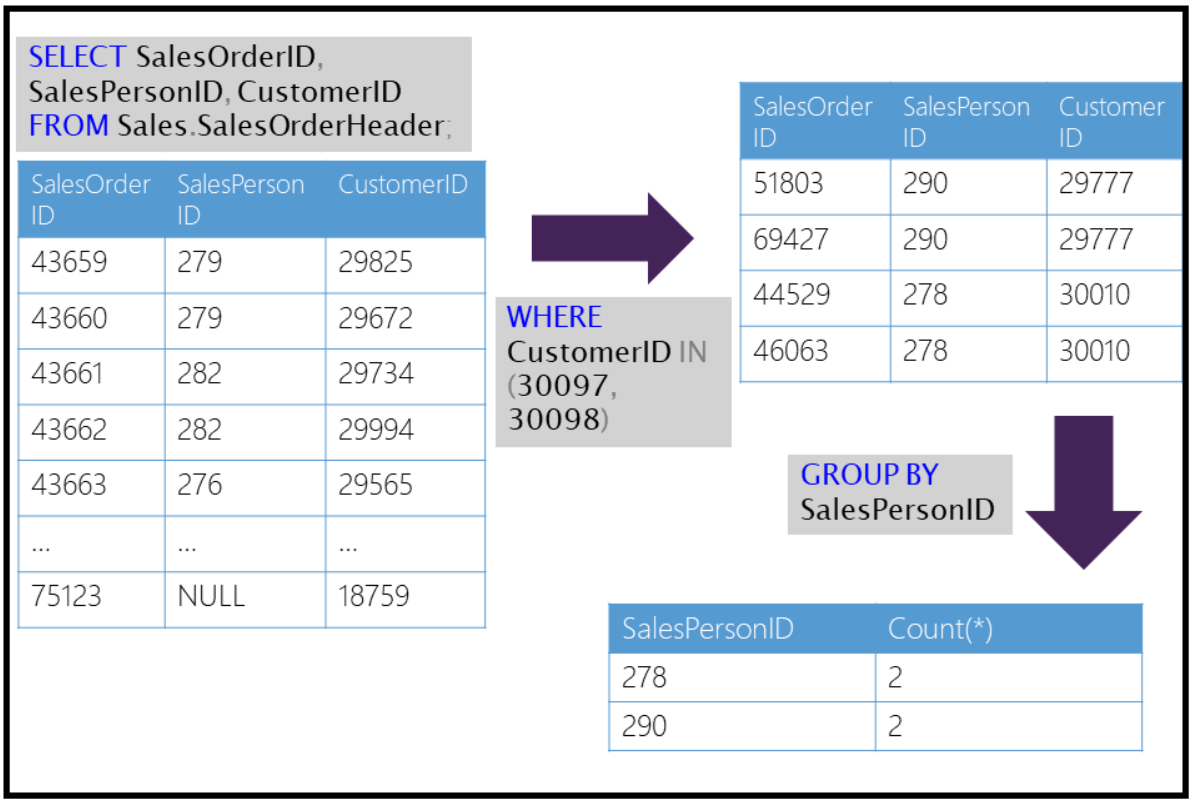
The net effect of this behavior is that you cannot combine a view of summary data with the detailed source date, using the T-SQL tools you have learned about so far. You will learn some approaches to solving the problem later in this course.

For more information about troubleshooting GROUP BY errors, see:

Troubleshooting GROUP BY Errors

http://aka.ms/yi931j

GROUP BY Workflow



Initially, the WHERE clause is processed followed by the GROUP BY. The slide shows the results of the WHERE clause, followed by the GROUP BY being performed on these results.

Source Queries

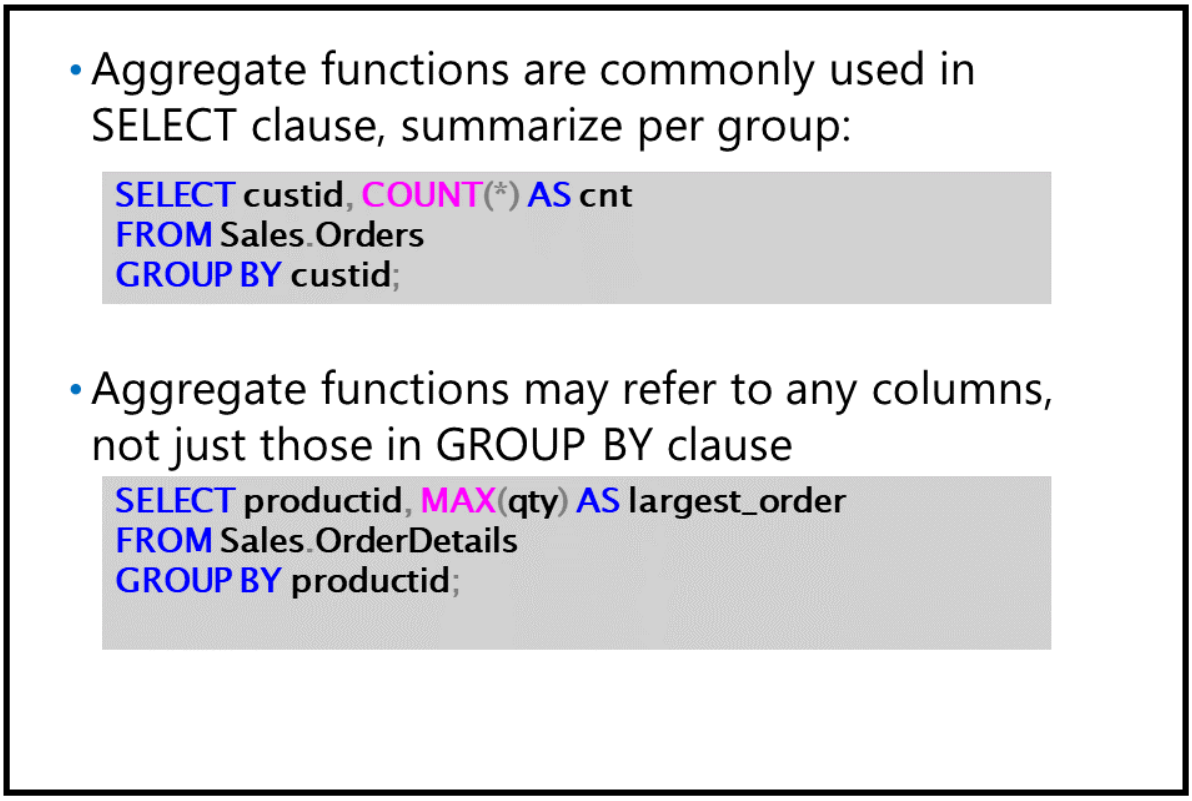
```
SELECT SalesOrderID, SalesPersonID, CustomerID
FROM Sales.SalesOrderHeader;

SELECT SalesOrderID, SalesPersonID, CustomerID
FROM Sales.SalesOrderHeader
WHERE CustomerID IN (29777, 30010);

SELECT SalesPersonID, COUNT(*)
FROM Sales.SalesOrderHeader
```

```
WHERE CustomerID IN (29777, 30010)
GROUP BY SalesPersonID;
```

Using GROUP BY with Aggregate Functions



As you have seen, if you use a GROUP BY clause in a T-SQL query, all columns listed in the SELECT clause must either be used in the GROUP BY clause itself, or be inputs to aggregate functions operating on each group.

You have seen the use of the COUNT function in conjunction with GROUP BY queries.

GROUP BY with Aggregate Example

```
SELECT productid, MAX(qty) AS largest_order
FROM Sales.OrderDetails
GROUP BY productid;
```

This returns (in part):

```
productid   largest_order
----------- -------------
23          70
46          60
69          65
```

```
29          80
75          120
```

> **Note:** The qty column, used as an input to the MAX function, is not used in the GROUP BY clause. This illustrates that, even though the detail rows returned by the FROM ... WHERE phase are lost to the GROUP BY phase, the source columns are still available for aggregation.

Demonstration: Using GROUP BY

In this demonstration, you will see how to use the GROUP BY clause.

Demonstration Steps

Use the GROUP BY Clause

1. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2a**, and then click **Execute**.
4. Select the code under the comment **Step 2b**, and then click **Execute**.
5. Select the code under the comment **Step 3**, and then click **Execute**.
6. Select the code under the comment **Step 4a**, and then click **Execute**.
7. Select the code under the comment **Step 4b**, and then click **Execute**.
8. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Select the best answer

You are writing the following T-SQL query to find out how many employees work in each department in your organization:

```
SELECT d.DepartmentID, d.DepartmentName, COUNT(e.EmployeeID) AS EmployeeCount
FROM HumanResources.Departments AS d
INNER JOIN HumanResources.Employees AS e
ON d.DepartmentID = e.DepartmentID
GROUP BY
```

Which columns should be included in the GROUP BY clause?

- All Columns
- EmployeeCount
- DepartmentID, DepartmentName
- DepartmentID

Check answer Show solution Reset

When using an aggregate function in the SELECT clause, all columns not included in an aggregate function must be included in the GROUP BY clause, otherwise an error will occur.

Lesson 3: Filtering Groups with HAVING

When you have created groups with a GROUP BY clause, you can further filter the results. The HAVING clause acts as a filter on groups, much like the WHERE clause acts as a filter on rows returned by the FROM clause. In this lesson, you will learn how to write a HAVING clause and understand the differences between HAVING and WHERE.

Lesson Objectives

After completing this lesson, you will be able to:

- Write queries that use the HAVING clause to filter groups.
- Compare HAVING to WHERE.
- Choose the appropriate filter for a scenario: WHERE or HAVING.

Filtering Grouped Data Using the HAVING Clause

- HAVING clause provides a search condition that each group must satisfy
- HAVING clause is processed after GROUP BY

```
SELECT custid, COUNT(*) AS count_orders
FROM Sales.Orders
GROUP BY custid
HAVING COUNT(*) > 10;
```

If a WHERE clause and a GROUP BY clause are present in a T-SQL SELECT statement, the HAVING clause is the fourth phase of logical query processing:

Logical Order	Phase	Comments
5	SELECT	
1	FROM	
2	WHERE	Operates on rows
3	GROUP BY	Creates groups
4	HAVING	Operates on groups
6	ORDER BY	

A HAVING clause enables you to create a search condition, conceptually similar to the predicate of a WHERE clause, which then tests each group returned by the GROUP BY clause.

GROUP BY Without HAVING Clause

```
SELECT custid, COUNT(*) AS count_orders
FROM Sales.Orders
GROUP BY custid;
```

Returns the groups, with the following message:

```
(89 row(s) affected)
```

GROUP BY with HAVING Clause

```
SELECT custid, COUNT(*) AS count_orders
FROM Sales.Orders
GROUP BY custid
HAVING COUNT(*) >= 10;
```

Returns the groups with the following message:

```
(28 row(s) affected)
```

> **Note:** Remember that HAVING is processed before the SELECT clause, so any column aliases created in a SELECT clause are not available to the HAVING clause.

HAVING (Transact-SQL)

http://aka.ms/wsrrp0

Compare HAVING to WHERE

- Using a COUNT(*) expression in a HAVING clause is useful to solve common business problems:
- Show only customers who have placed more than one order:

```
SELECT c.custid, COUNT(*) AS cnt
FROM Sales.Customers AS c
JOIN Sales.Orders AS o ON c.custid = o.custid
GROUP BY c.custid
HAVING COUNT(*) > 1;
```

- Show only products that appear on 10 or more orders:

```
SELECT p.productid, COUNT(*) AS cnt
FROM Production.Products AS p JOIN Sales.OrderDetails AS
od ON p.productid = od.productid
GROUP BY p.productid
HAVING COUNT(*) >= 10;
```

While both HAVING and WHERE clauses filter data, it is important to remember that WHERE operates on rows returned by the FROM clause. If a GROUP BY ... HAVING section exists in your query following a WHERE clause, the WHERE clause will filter rows before GROUP BY is processed—potentially limiting the groups that can be created.

A HAVING clause is processed after GROUP BY and only operates on groups, not detail rows. To summarize:

- A WHERE clause controls which rows are available to the next phase of the query.
- A HAVING clause controls which groups are available to the next phase of the query.

> **Note:** WHERE and HAVING clauses are not mutually exclusive.

You will see a comparison between WHERE and HAVING in the next demonstration.

Demonstration: Filtering Groups with HAVING

In this demonstration, you will see how to filter grouped data using the HAVING clause.

Demonstration Steps

Filter Grouped Data Using the HAVING Clause

1. In Solution Explorer, open the **31 - Demonstration C.sql** script file.

2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2a**, and then click **Execute**.
4. Select the code under the comment **Step 2b**, and then click **Execute**.
5. Select the code under the comment **Step 2c**, and then click **Execute**. Note the error message.
6. Select the code under the comment **Step 2d**, and then click **Execute**.
7. Select the code under the comment **Step 2e**, and then click **Execute**.
8. Select the code under the comment **Step 2f**, and then click **Execute**.
9. Select the code under the comment **Step 2g**, and then click **Execute**.
10. Select the code under the comment **Step 2h**, and then click **Execute**.
11. Select the code under the comment **Step 2i**, and then click **Execute**.
12. Close SQL Server Management Studio without saving any files.

Check Your Knowledge

Discovery

You are writing a query to count the number of orders placed for each product. You have the following query:

```
SELECT p.ProductName, COUNT(*) AS OrderCount
FROM Sales.Products AS p
JOIN Sales.OrderItems AS o
ON p.ProductID = o.ProductID
GROUP BY p.ProductName;
```

You want to change the query to return only products that cost more than $10. Should you add a HAVING clause or a WHERE clause?

Show solution Reset

Add a WHERE clause such as WHERE p.Price > 10. This must be done in the WHERE clause as the groups returned do not include a price field for a HAVING clause to operate on.

Lab: Grouping and Aggregating Data

Scenario

You are an Adventure Works business analyst, who will be writing reports using corporate databases stored in SQL Server. You have been given a set of business requirements for data and you will write T-SQL queries to retrieve it from the databases. You will need to perform calculations upon groups of data and filter according to the results.

Objectives

After completing this lab, you will be able to:

•

Write queries that use the GROUP BY clause.

- Write queries that use aggregate functions.
- Write queries that use distinct aggregate functions.
- Write queries that filter groups with the HAVING clause.

Lab Setup

Estimated Time: 60 minutes

Virtual machine: **20761C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Writing Queries That Use the GROUP BY Clause

Scenario

The sales department want to create additional upsell opportunities from existing customers. The staff need to analyze different groups of customers and product categories, depending on several business rules. Based on these rules, you will write SELECT statements to retrieve the needed rows from the Sales.Customers table.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT Statement to Retrieve Different Groups of Customers
3. Add an Additional Column From the Sales.Customers Table
4. Write a SELECT Statement to Retrieve the Customers with Orders for Each Year
5. Write a SELECT Statement to Retrieve Groups of Product Categories Sold in a Specific Year

Detailed Steps ▲

Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab09\Starter** folder as Administrator.

Detailed Steps ▲

Task 2: Write a SELECT Statement to Retrieve Different Groups of Customers

1. Open the project file **D:\Labfiles\Lab09\Starter\Project\Project.ssmssln** and the T-SQL script **51 - Lab Exercise 1.sql**. Ensure that you are connected to the **TSQL** database.
2. Write a SELECT statement that will return groups of customers who made a purchase. The SELECT clause should include the **custid** column from the **Sales.Orders** table, and the **contactname** column from the **Sales.Customers** table. Group both columns and filter only the orders from the sales employee whose empid equals five.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab09\Solution\52 - Lab Exercise 1 - Task 2 Result.txt.

Detailed Steps ▲

Task 3: Add an Additional Column From the Sales.Customers Table

1. Copy the T-SQL statement in task 1 and modify it to include the **city** column from the **Sales.Customers** table in the SELECT clause.
2. Execute the query.
3. You will get an error. What is the error message? Why?
4. Correct the query so that it will execute properly.
5. Execute the query and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab09\Solution\53 - Lab Exercise 1 - Task 3 Result.txt.

Detailed Steps ▲

Task 4: Write a SELECT Statement to Retrieve the Customers with Orders for Each Year

1. Write a SELECT statement that will return groups of rows based on the **custid** column and a calculated column **orderyear** representing the order year based on the **orderdate** column from the **Sales.Orders** table. Filter the results to include only the orders from the sales employee whose empid equals five.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab09\Solution\54 - Lab Exercise 1 - Task 4 Result.txt.

Detailed Steps ▲

Task 5: Write a SELECT Statement to Retrieve Groups of Product Categories Sold in a Specific Year

1. Write a SELECT statement to retrieve groups of rows based on the **categoryname** column in the **Production.Categories** table. Filter the results to include only the product categories that were ordered in the year 2008.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab09\Solution\55 - Lab Exercise 1 - Task 5 Result.txt.

Result: After this exercise, you should be able to use the GROUP BY clause in the T-SQL statement.

Exercise 2: Writing Queries That Use Aggregate Functions

Scenario

The marketing department wants to launch a new campaign, so the staff need to gain a better insight into the existing customers' buying behavior. You should create different sales reports, based on the total and average sales amount per year and per customer.

The main tasks for this exercise are as follows:

1. Write a SELECT statement to Retrieve the Total Sales Amount Per Order
2. Add Additional Columns
3. Write a SELECT Statement to Retrieve the Sales Amount Value Per Month
4. Write a SELECT Statement to List All Customers with the Total Sales Amount and Number of Order Lines Added

Detailed Steps ▲

Task 1: Write a SELECT statement to Retrieve the Total Sales Amount Per Order

1. Open the T-SQL script **61 - Lab Exercise 2.sql**. Ensure that you are connected to the **TSQL** database.
2. Write a SELECT statement to retrieve the **orderid** column from the **Sales.Orders** table and the total sales amount per orderid. (Hint: multiply the **qty** and **unitprice** columns from the **Sales.OrderDetails** table.) Use the alias **salesamount** for the calculated column. Sort the result by the total sales amount in descending order.
3. Execute the written statement and compare the results that you achieved with the desired results shown in

the file D:\Labfiles\Lab09\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.

Detailed Steps ▲

Task 2: Add Additional Columns

1. Copy the T-SQL statement in task 1 and modify it to include the total number of order lines for each order and the average order line sales amount value within the order. Use the aliases **nooforderlines** and **avgsalesamountperorderline**, respectively.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\63 - Lab Exercise 2 - Task 2 Result.txt.

Detailed Steps ▲

Task 3: Write a SELECT Statement to Retrieve the Sales Amount Value Per Month

1. Write a select statement to retrieve the total sales amount for each month. The SELECT clause should include a calculated column named **yearmonthno** (YYYYMM notation), based on the **orderdate** column in the **Sales.Orders** table and a total sales amount (multiply the **qty** and **unitprice** columns from the **Sales.OrderDetails** table). Order the result by the **yearmonthno** calculated column.
2. Execute the written statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab09\Solution\64 - Lab Exercise 2 - Task 3 Result.txt.

Detailed Steps ▲

Task 4: Write a SELECT Statement to List All Customers with the Total Sales Amount and Number of Order Lines Added

1. Write a select statement to retrieve all the customers (including those who did not place any orders) and their total sales amount, maximum sales amount per order line, and number of order lines.
2. The SELECT clause should include the **custid** and **contactname** columns from the **Sales.Customers** table and four calculated columns based on appropriate aggregate functions:
    a. **totalsalesamount**, representing the total sales amount per order
    b. **maxsalesamountperorderline**, representing the maximum sales amount per order line
    c. **numberofrows**, representing the number of rows (use * in the COUNT function)
    d. **numberoforderlines**, representing the number of order lines (use the **orderid** column in the COUNT

function)

3. Order the result by the **totalsalesamount** column.

4. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\65 - Lab Exercise 2 - Task 4 Result.txt.

5. Notice that the **custid** 22 and 57 rows have a NULL in the columns with the SUM and MAX aggregate functions. What are their values in the **COUNT** columns? Why are they different?

Exercise 3: Writing Queries That Use Distinct Aggregate Functions

Scenario

The marketing department want to have some additional reports that display the number of customers who made any order in a specific time period and the number of customers based on the first letter in the contact name.

The main tasks for this exercise are as follows:

1. Modify a SELECT Statement to Retrieve the Number of Customers
2. Write a SELECT Statement to Analyze Segments of Customers
3. Write a SELECT Statement to Retrieve Additional Sales Statistics

Detailed Steps ▲

Task 1: Modify a SELECT Statement to Retrieve the Number of Customers

1. Open the T-SQL script **71 - Lab Exercise 3.sql**. Ensure that you are connected to the **TSQL** database.
2. A junior analyst prepared a T-SQL statement to retrieve the number of orders and the number of customers for each order year. Observe the provided T-SQL statement and execute it:

```
SELECT
YEAR(orderdate) AS orderyear,
COUNT(orderid) AS nooforders,
COUNT(custid) AS noofcustomers
FROM Sales.Orders
GROUP BY YEAR(orderdate);
```

3. Observe the results. Notice that the number of orders is the same as the number of customers. Why?
4. Amend the T-SQL statement to show the correct number of customers who placed an order for each year.

5. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

Detailed Steps ▲

Task 2: Write a SELECT Statement to Analyze Segments of Customers

1. Write a SELECT statement to retrieve the number of customers based on the first letter of the values in the **contactname** column from the **Sales.Customers** table. Add an additional column to show the total number of orders placed by each group of customers. Use the aliases **firstletter**, **noofcustomers** and **nooforders**. Order the result by the **firstletter** column.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

Detailed Steps ▲

Task 3: Write a SELECT Statement to Retrieve Additional Sales Statistics

1. Copy the T-SQL statement in exercise 1, task 5, and modify to include the following information about each product category—total sales amount, number of orders, and average sales amount per order. Use the aliases **totalsalesamount, nooforders**, and **avgsalesamountperorder**, respectively.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\74 - Lab Exercise 3 - Task 3 Result.txt.

Result: After this exercise, you should have an understanding of how to apply a DISTINCT aggregate function.

Exercise 4: Writing Queries That Filter Groups with the HAVING Clause

Scenario

The sales and marketing departments were satisfied with the reports you provided to analyze customers' behavior. Now they would like to have the results filtered, based on the total sales amount and number of orders. So, in the final exercise, you will learn how to filter the result, based on aggregated functions, and learn when to use the WHERE and HAVING clauses.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve the Top 10 Customers

2. Write a SELECT Statement to Retrieve Specific Orders
3. Apply Additional Filtering
4. Retrieve the Customers with More Than 25 Orders

Detailed Steps ▲

Task 1: Write a SELECT Statement to Retrieve the Top 10 Customers

1. Open the T-SQL script **81 - Lab Exercise 4.sql**. Ensure that you are connected to the **TSQL** database.
2. Write a SELECT statement to retrieve the top 10 customers (by total sales amount) who spent more than $10,000. Display the **custid** column from the **Orders** table and a calculated column that contains the total sales amount, based on the **qty** and **unitprice** columns from the **Sales.OrderDetails** table. Use the alias **totalsalesamount** for the calculated column.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.

Detailed Steps ▲

Task 2: Write a SELECT Statement to Retrieve Specific Orders

1. Write a SELECT statement against the **Sales.Orders** and **Sales.OrderDetails** tables, and display the **empid** column and a calculated column representing the total sales amount. Filter the results to group only the rows with an order year 2008.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\83 - Lab Exercise 4 - Task 2 Result.txt.

Detailed Steps ▲

Task 3: Apply Additional Filtering

1. Copy the T-SQL statement in task 2 and modify it to apply an additional filter to retrieve only the rows that have a sales amount higher than $10,000.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\84 - Lab Exercise 4 - Task 3_1 Result.txt.
3. Apply an additional filter to show only employees with empid equal to 3.

4. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\85 - Lab Exercise 4 - Task 3_2 Result.txt.

5. Did you apply the predicate logic in the WHERE clause or the HAVING clause? Which do you think is better? Why?

Detailed Steps ▲

Task 4: Retrieve the Customers with More Than 25 Orders

1. Write a SELECT statement to retrieve all customers who placed more than 25 orders and add information about the date of the last order and the total sales amount. Display the **custid** column from the **Sales.Orders** table and two calculated columns— **lastorderdate** based on the **orderdate** column, and **totalsalesamount** based on the **qty** and **unitprice** columns in the **Sales.OrderDetails** table.

2. Execute the written statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab09\Solution\86 - Lab Exercise 4 - Task 4 Result.txt.

3. Close SQL Server Management Studio without saving any files.

Result: After this exercise, you should have an understanding of how to use the HAVING clause.

Module Review and Takeaways

In this lesson, you have learned how to:

- List the built-in aggregate functions provided by SQL Server.
- Write queries that use aggregate functions in a SELECT list to summarize all the rows in an input set.
- Describe the use of the DISTINCT option in aggregate functions.
- Write queries using aggregate functions that handle the presence of NULLs in source data.

Review Question(s)

Check Your Knowledge

Discovery

What is the difference between the COUNT function and the COUNT_BIG function?

Show solution Reset

COUNT returns an int; COUNT_BIG returns a big_int.

Check Your Knowledge

Discovery

Can a GROUP BY clause include more than one column?

Show solution Reset

Yes, separated by commas.

Check Your Knowledge

Discovery

In a query, can a WHERE clause and a HAVING clause filter on the same column?

Show solution Reset

Yes.