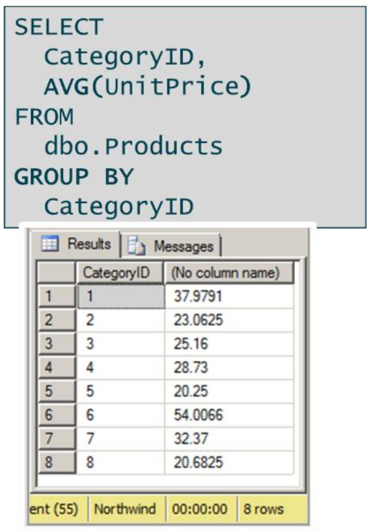
GROUP BY

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

The **GROUP BY** statement groups rows that have the same values into summary rows, e.g. How many orders did we have by Country?

The **GROUP BY** statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

Effectively *GROUP BY* tells SQL to generate a unique list of the values in the *GROUP BY* column(s) using a DISTINCT, and then to calculate the aggregate for all of the rows with that value using a WHERE.



```
SELECT
  CategoryID,
  AVG(UnitPrice)
FROM
  dbo.Products
GROUP BY
  CategoryID
```

Results | Messages

	CategoryID	(No column name)
1	1	37.9791
2	2	23.0625
3	3	25.16
4	4	28.73
5	5	20.25
6	6	54.0066
7	7	32.37
8	8	20.6825

ent (55) | Northwind | 00:00:00 | 8 rows

HAVING

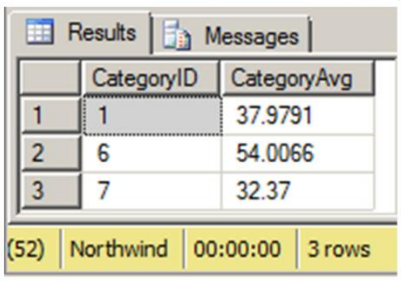
The **HAVING** clause was added to SQL because the **WHERE** keyword cannot be used with aggregate functions. You can use the **HAVING** clause on any of the columns in the select list.

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

A **HAVING** clause is like a **WHERE** clause, but applies only to groups as a whole (that is, to the rows in the result set representing groups), whereas the **WHERE** clause applies to individual rows.

1. The **WHERE** clause is applied first to the individual rows in the tables.
2. Only the rows that meet the conditions in the **WHERE** clause are grouped
3. The **HAVING** clause is then applied to the rows in the result set.
4. Only the groups that meet the **HAVING** conditions appear in the query output.

```
SELECT
  CategoryID,
  AVG(UnitPrice)
FROM
  dbo.Products
GROUP BY
  CategoryID
HAVING
  AVG(UnitPrice) > 30
```



	CategoryID	CategoryAvg
1	1	37.9791
2	6	54.0066
3	7	32.37

```
(52) | Northwind | 00:00:00 | 3 rows
```

Exercise 1: Basic aggregates

You want to learn more about the data stored in the dbo.Orders table, specifically those placed by Nancy Davolio.

In this exercise, you will use basic aggregates.

Task 1: Create a report that selects a count of rows

1. Create a new query and save it with a name of “OrderAnalysis.sql”.
2. Write a report that uses the Northwind database and displays a count of all the rows in the dbo.Orders table. Alias the count to call it “NumberOfOrders”.
3. Execute the query and verify that the result set contains a single value of 830.

Task 2: Write a report that selects maximums and minimums

1. Modify your existing report.
2. Add an aggregate to the select list that calculates the minimum OrderDate value. Alias it as “EarliestOrder”.
3. Add another aggregate that calculates the maximum OrderDate value. Alias it as “LatestOrder”.
4. Execute the query and verify that the answers are 830, 4 July 1996 and 6 May 1998.

Task 3: Write a report that selects aggregates for only one employee

1. Modify your existing report.
2. Add a WHERE clause to your query so that it only includes rows with an EmployeeID equal to 1.
3. Execute the query and verify that the answers are now 123, 17 July 1996
4. and 6 May 1998.

Exercise 2: Grouping aggregates

You have been asked to create a report on the number of orders placed by each customer. The report is to be sorted by the number of orders placed, from highest to lowest.

In this exercise, you will use the GROUP BY clause and an ORDER BY.

Task 1: Create a query that counts orders

1. Create a new query and save it with a name of “CustomerOrders.sql”.
2. Write a report that uses the Northwind database and displays a count of all the OrderIDs in the dbo.Orders table. Alias the aggregate as “NumberOfOrders”.
3. Execute the query and verify that it returns a value of 830.

Task 2: Write a report that groups orders based on the customer’s ID

1. Modify the existing report.
2. Add the CustomerID column to the report’s select list.
3. Note: At this point, the query won’t work.
4. Add a GROUP BY clause to the report that groups the results based on the CustomerID column.
5. Execute the query and verify that it returns 89 rows, the top one being ALFKI with a NumberOfOrders of 6.

Task 3: Write a report that sorts order counts in descending order

1. Modify the existing report.
2. Add an ORDER BY clause to the report that sorts on the NumberOfOrders column in descending order.
3. Execute the query and verify that it returns 89 rows, the top one now being SAVEA with a count of 31.

Exercise 3: Aggregating calculated values and filtering aggregates

Northwind Traders are now trying to see exactly how much customers are spending on products.

In this exercise, you will help them to do so by using aggregate functions on calculated columns.

Task 1: Create a report that sums quantities of products sold

1. Create a new query and save it with a name of "ProductSales.sql".
2. Write a report that uses the Northwind database and selects the ProductID and the sum of the Quantity columns from the dbo.[Order Details] table. Alias the aggregate column as "TotalSold".
3. Group the results on the ProductID column.
4. Execute the query and verify that it returns 77 rows, the first row being product id 23, with a TotalSold of 580.

Task 2: Create a report that sums a calculation

1. Modify the existing report.
2. Modify the SUM aggregate so that it adds up the value of the Quantity column multiplied by the UnitPrice column for each product. Change the alias name to "TotalValue".
3. Sort the results on the TotalValue column, in descending order.
4. Execute the query and verify that the top-selling product is productid 38, with a total sales value of 149984.20.

Task 3: Create a report that filters aggregate values

1. Modify the existing report.
2. Add a HAVING clause to the report to only return the rows with a TotalValue of less than or equal to 5000.
3. REMEMBER: just like with a WHERE clause, the actual column named TotalValue doesn't exist yet in your HAVING, so you'll need to re-use the calculation.
4. Execute the query and verify that you now see only 16 rows, the first of which is product 23 with a value of 4840.20.