Aggregate Functions

Aggregate Function	Output data-type	Result
AVG([DISTINCT] *column_values*)	numeric	The average of the non-null columns in the expression
SUM([DISTINCT] *column_values*)	numeric	The total of the non-null columns in the expression
MIN([DISTINCT] *column_values*)	numeric, date, string	The lowest value off the non-null columns in the expression
MAX([DISTINCT] *column_values*)	numeric, date, string	The highest value of the non-null columns in the expression
COUNT([DISTINCT] *column_values*)	numeric	The number of the non-null columns in the expression
COUNT(*)	numeric	The number of rows returned by the query

- Aggregate functions are synonymous with column functions.
- A summary query uses at least on column function.
- **AVG**, **SUM** return numeric values.
- **MIN**, **MAX**, **COUNT** can return numeric, date, or string values
- All values are included in aggregate functions by default unless you specify the **DISTINCT** keyword
- Duplicate rows are excluded in all aggregate functions with the exception of **COUNT(*)**
- ***** IF YOU CODE AN AGGREGATE FUNCTION IN THE SELECT STATEMENT, YOU CANNOT ALSO INCLUDE NON-AGGREGATE FUNCTIONS IN THE SELECT STATEMENT UNLESS THOSE NON-AGGREGATE COLUMNS ARE INCLUDED IN A **GROUP BY** CLAUSE

```
USE bike;
SELECT AVG(list_price), SUM(list_price), MIN(list_price),
     MAX(list_price), COUNT(list_price), COUNT(*)
FROM product;
```

```
USE bike;
SELECT AVG(list_price), SUM(list_price), MIN(list_price),
    MAX(list_price), COUNT(list_price), COUNT(*)
FROM product;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

AVG(list_price)	SUM(list_price)	MIN(list_price)	MAX(list_price)	COUNT(list_price)	COUNT(*)
1520.591402	488109.84	89.99	11999.99	321	321

Grouping Data

Aggregate Function	Order of Execution	Description
GROUP BY	3	Groups rows of a result set based on columns or expressions separated by commas. You cannot

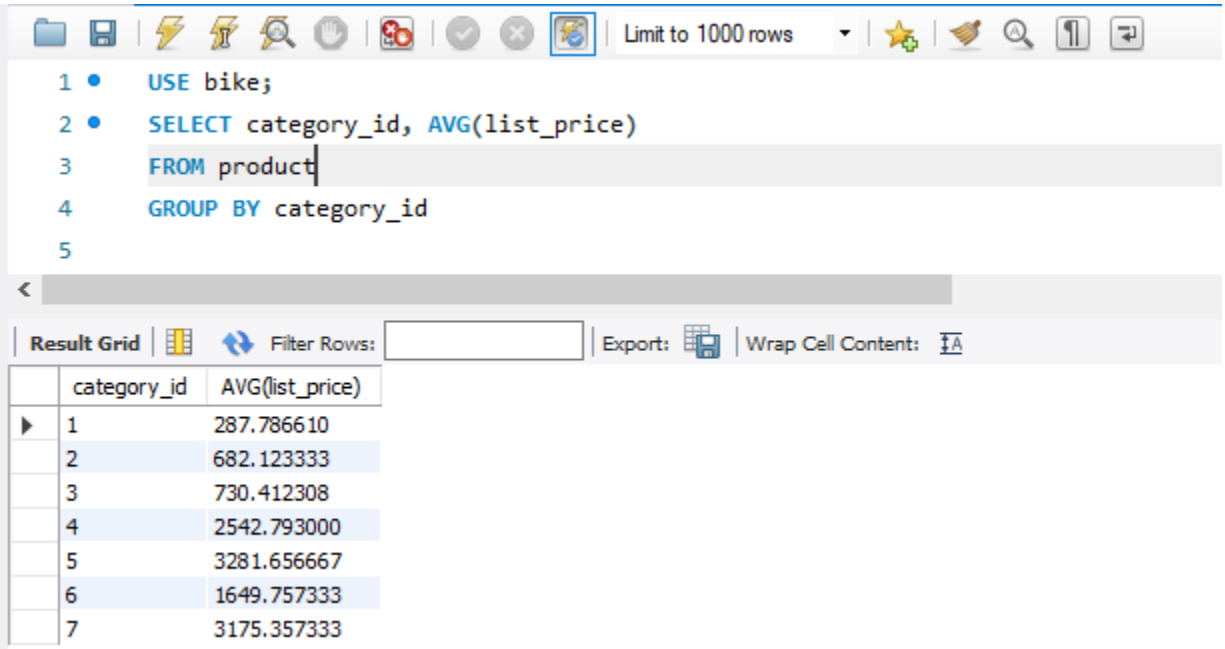
- Group rows based on a column(s) or expression(s).
- If you use an *aggregate function* with a GROUP BY clause, the aggregation is calculated for each group.

Filtering Clause	Order of Execution	Description
WHERE	2	* Executes BEFORE GROUP BY clause * CANNOT use aggregate functions * You can specify columns not defined in the SELECT clause
HAVING	4	* Executes AFTER GROUP BY clause * CAN use aggregate functions * You CANNOT specify columns not defined in the SELECT clause

- Notice the order of execution. **GROUP BY** happens after **WHERE** but before **HAVING**.
- It is possible to use WHERE and HAVING in the same statement. They are not mutually exclusive.

Simple GROUP BY query:

```
USE bike;
SELECT category_id, AVG(list_price)
FROM product
GROUP BY category_id
```



category_id	AVG(list_price)
1	287.786610
2	682.123333
3	730.412308
4	2542.793000
5	3281.656667
6	1649.757333
7	3175.357333

USE bike:

- Set the bike database to be the default

SELECT category_id, AVG(list_price):

- Select the category_id from the base table
- Calculate the Average of the list price for all rows in the table

FROM product:

- Product is the base table from which data will be returned

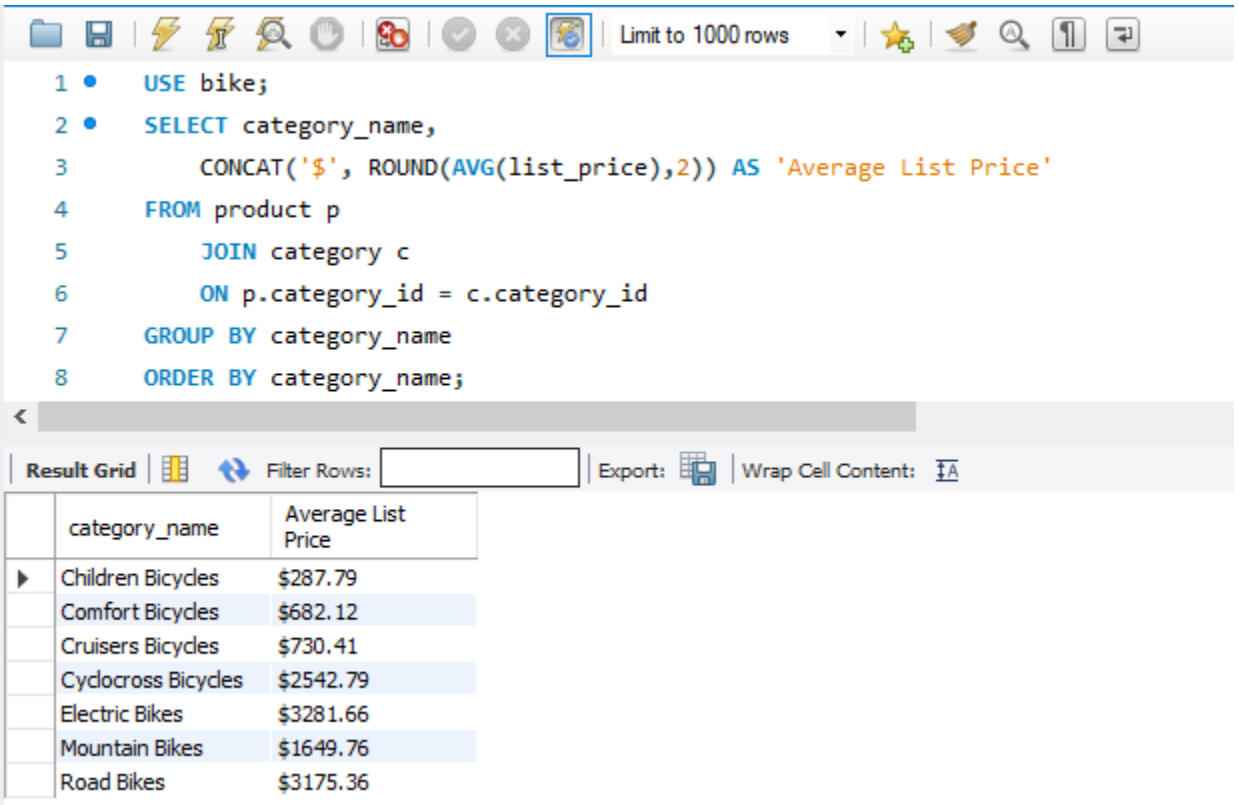
GROUP BY category_id:

- Instead of returning a single value that is the average of all list_price items in the product table, return an average list_price for each category
- Without the **GROUP BY** clause we see from our first example only a single row is returned with an average list_price of 1520.591402.
- With the **GROUP BY** clause, we return an average for each category_id.

Improving the GROUP BY query

- The report would be nicer if we showed the category name instead of the category_id. This will require joining the product table to the category table.
- We can **ROUND** the **AVG** list price by category to TWO decimals points.
- We can **CONCAT** the dollar sign to the left of the list_price.

```
USE bike;
SELECT category_name,
      CONCAT('$', ROUND(AVG(list_price),2)) AS 'Average List Price'
FROM product p
      JOIN category c
    ON p.category_id = c.category_id
GROUP BY category_name
ORDER BY category_name;
```



category_name	Average List Price
Children Bicycles	$287.79
Comfort Bicycles	$682.12
Cruisers Bicycles	$730.41
Cyclocross Bicycles	$2542.79
Electric Bikes	$3281.66
Mountain Bikes	$1649.76
Road Bikes	$3175.36

```
USE bike:
```

- Set the bike database to be the default

```
SELECT category_name,
      CONCAT('$', ROUND(AVG(list_price),2)) AS 'Average List Price'
```

- Return the category_name from the category table.
- You do not have to qualify the column name with the table name because category_name only exists in one table of the join.
- Return the list price with the '$' followed by the list_price rounded to the 2nd decimal and assigned a column alias of 'Average List Price'.
- You do not have to qualify the column name of list_price because it exists in only one table of the join.

```
FROM product p
      JOIN category c
    ON p.category_id = c.category_id
```

- JOIN the product table to the category table
- Assign a table alias of "p" to product and "c" to category
- The join condition is the primary key of category_id from the category table equal to the foreign key of category_id in the product table.

```
GROUP BY category_name
```

- Instead of retrieving a single value with the average price of all products, return a list of average prices by category name.

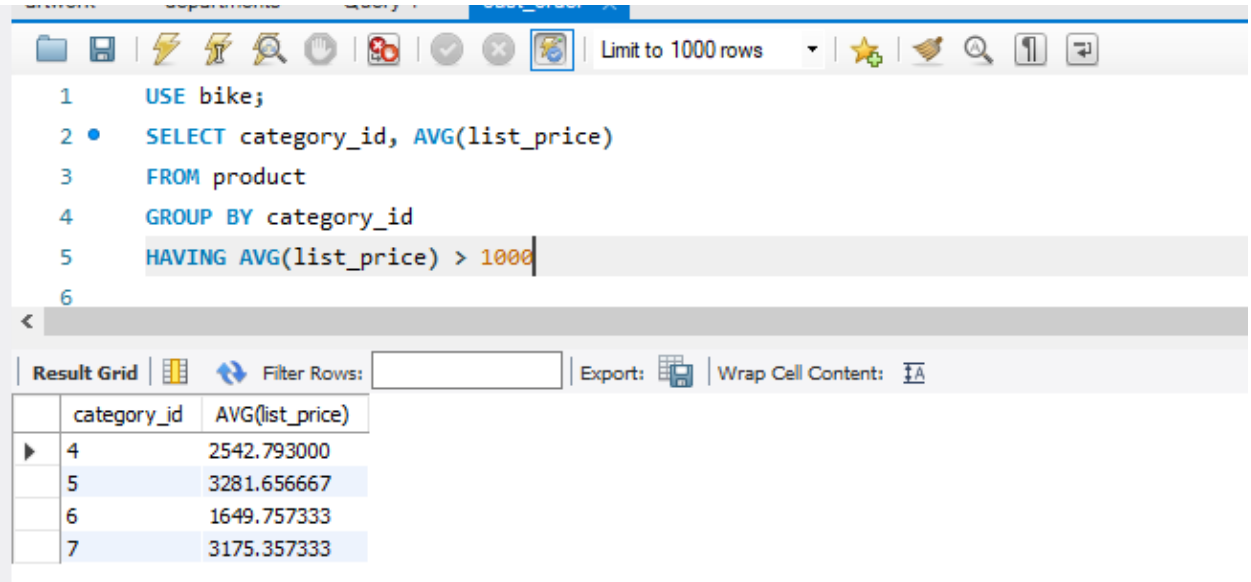
```
ORDER BY category_name;
```

- Sort the results by category_name

Using the HAVING Clause

- The HAVING CLAUSE allows you to use an aggregate function as a filter. This is not allowed in a WHERE clause.
- Any columns or expressions you want to use in a HAVING clause, MUST BE DEFINED IN THE SELECT CLAUSE as well.

```
USE bike;
SELECT category_id, AVG(list_price)
FROM product
GROUP BY category_id
HAVING AVG(list_price) > 1000
```



We previously discussed the preceding lines of code for this query so we will focus solely on the HAVING clause.

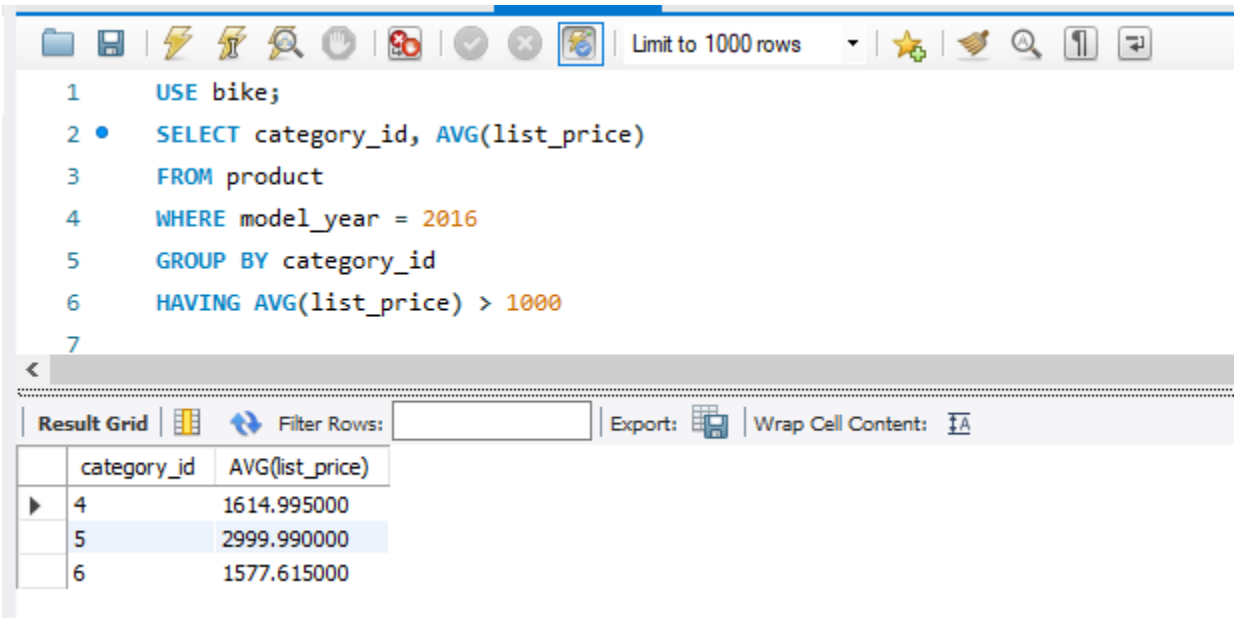
```
HAVING AVG(list_price) > 1000
```

- The **HAVING** clause executes after the **GROUP BY** clause but before the **SELECT** clause.
- If you use an aggregate function in the **HAVING** clause, you must include the same aggregate function in the **SELECT** clause.
- If you reference a column or expression in the **HAVING** clause, you must include the same column or expression in the **SELECT** clause.
- You cannot use aggregate functions in a **WHERE** clause

Using HAVING and WHERE Clause Together

Below is an example of a statement that includes both the HAVING and WHERE clause in the same SQL statement.

```
USE bike;
SELECT category_id, AVG(list_price)
FROM product
WHERE model_year = 2016
GROUP BY category_id
HAVING AVG(list_price) > 1000
```




```
WHERE model_year = 2016
```

- The **WHERE** clause executes before the **GROUP BY** clause.
- You can refer to columns not defined in the **SELECT** clause.
- You cannot use aggregate functions in the **WHERE** clause.

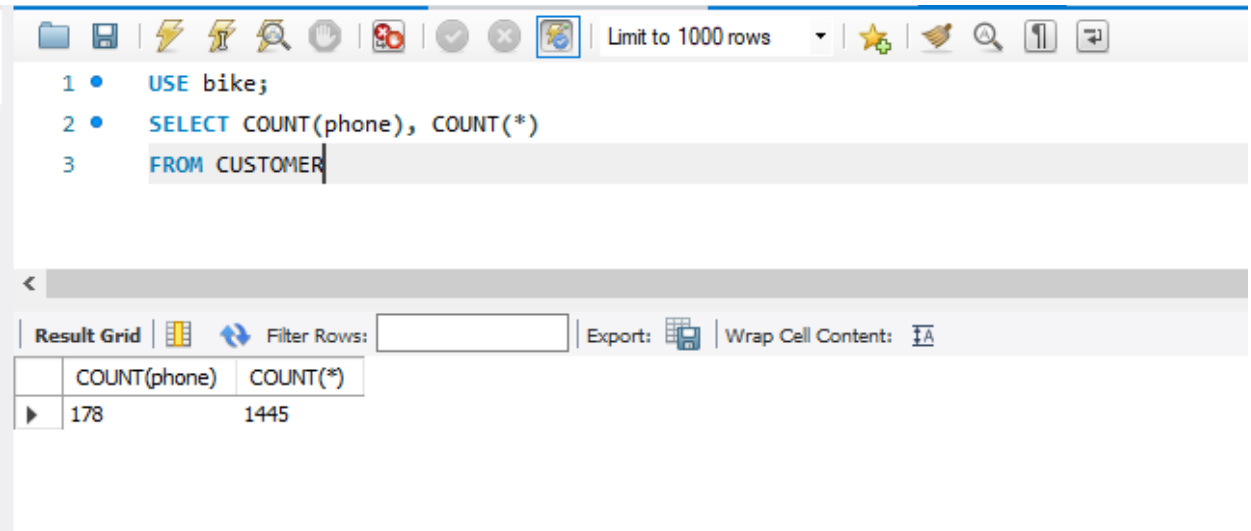
```
HAVING AVG(list_price) > 1000
```

- The **HAVING** clause executes after the **GROUP BY** clause but before the **SELECT** clause.
- If you use an aggregate function in the **HAVING** clause, you must include the same aggregate function in the **SELECT** clause.
- If you reference a column or expression in the **HAVING** clause, you must include the same column or expression in the **SELECT** clause.
- You cannot use aggregate functions in a **WHERE** clause.

The Difference Between Count(*column_name*) and COUNT(*)

- COUNT(*) is the only aggregate function that counts rows with null values.
- When you specify a count based on a specific column, null values will not be counted.

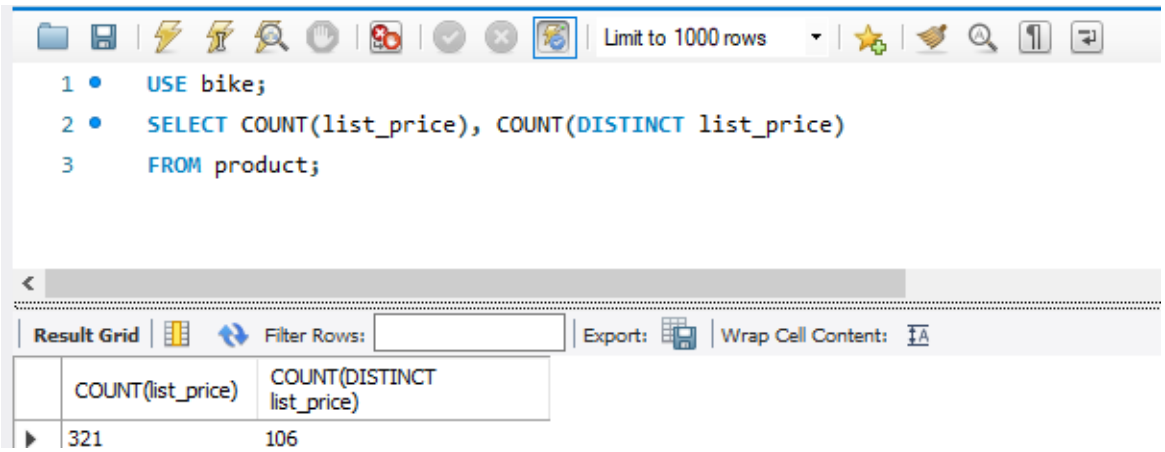
```
USE bike;
SELECT COUNT(phone), COUNT(*)
FROM CUSTOMER
```



Using the DISTINCT Keyword

- The DISTINCT keyword allows you to eliminate duplicate rows in aggregate functions.
- You may also use the DISTINCT keyword with columns of the base table in a SELECT statement.

```
USE bike;
SELECT COUNT(list_price), COUNT(DISTINCT list_price)
FROM product;
```



- COUNT(list_price) counts all the rows in the product table that have a list price.
- COUNT(DISTINCT list_price) eliminates duplicate values in the list_price.