Let’s do some coding practice to learn more about aggregate functions, taking a group of values and get one summary value from that data.

Let’s look at the art gallery database.

```
USE art_gallery;

SELECT country, fname, lname
FROM artist;
```

Here we get all the artists. We have no filter (or WHERE clause) so everyone is showing up. Now our database is very small so there aren’t many artist here, but it is everyone we have in the database.

Let’s add a filter and get just those artists from France:

```
SELECT country, fname, lname
FROM artist
WHERE country = ‘France’;
```

Now we are getting just those from France. So far we haven’t summarized anything. Just filtered. But what if we wanted to know how many artist were from France. Well that’s easy in our case right? We just add them up. But what if our data had thousands or millions of rows of artists. Our query wouldn’t be very efficient because we’d have to count all the rows.

That’s where aggregate functions come in. One of those functions is the COUNT function. Let’s add that function. Let’s take off the last name and first name for now and I’ll explain why in a minute.

```
SELECT COUNT(country)
FROM artist
WHERE country = ‘France’;
```

Now we see that we get one row of summary data from multiple rows. If we’d had 100 artist from France we’d now see a result set with one value of 100 here. So with the aggregate function we quickly see how many are from France using the COUNT aggregate function.

So why can’t we have first and last name here.

```
SELECT COUNT(country), fname, lname
FROM artist
WHERE country = ‘France’;
```

Because we have summarized a number of rows into one row. So fname and lname go with each individual artist and they don't belong with a summarized value. How can we show both Monet and van Gogh in one row of information? Or for that matter 100 different artist if that's how many we had from France.

We could put the country name though because they are all the same for that aggregate function.

```
SELECT COUNT(country), country
FROM artist
WHERE country = 'France';
```

Let's look at the bike database since we have some bike prices we can work with to do some math type aggregate functions.

```
USE bike;

SELECT list_price
FROM product;
```

Let's add an aggregate function of average this time instead of COUNT.

```
SELECT AVG(list_price)
FROM product;
```

Now we are taking every single bike price and we are getting the average of all the bike prices.

Now that math has been done on the numbers we are getting a larger decimal point. So we can use a scalar function with the results of the aggregate function to get the decimal places according to currency.

```
SELECT FORMAT(AVG(list_price), 2)
FROM product;
```

Let's add up all the prices with a sum aggregate function.
No extra decimals are created with these aggregate functions so we don't need the FORMAT function here.

```
SELECT SUM(list_price)
FROM product;
```

Let’s get the largest and the smallest bike prices with MAX and MIN.

```
SELECT MAX(list_price), MIN(list_price)
FROM product;
```

What if I wanted the average of each model year? I could query something like this:

```
SELECT AVG(list_price)
FROM product
WHERE model_year = 2016;
```

I could run a query like this for each year.

```
SELECT AVG(list_price)
FROM product
WHERE model_year = 2017;
```

But what if I have 50 years’ worth of bike data. How can I speed this up and get them all at once?  Now we are going to add a GROUP BY clause. Now we get the average of each year.

```
SELECT AVG(list_price)
FROM product
GROUP BY model_year;
```

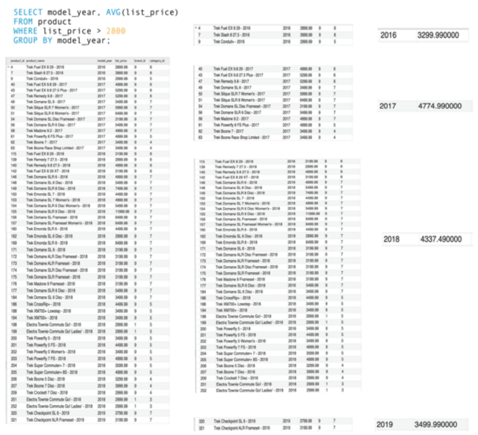
We can add the model_year column to clarify this a bit.

```
SELECT model_year, AVG(list_price)
FROM product
GROUP BY model_year;
```

I’m going to add a WHERE here to filter it just a bit so we can visual this with a few less rows.

```
SELECT model_year, AVG(list_price)
FROM product
WHERE list_price > 2800
GROUP BY model_year;
```

Let’s look at what just happened.



```
SELECT model_year, AVG(list_price)
FROM product
WHERE list_price > 2800
GROUP BY model_year;
```

| product_id | product_name | model_year | list_price | brand_id | category_id |
|---|---|---|---|---|---|
| 4 | Trek Fuel EX 8 29 - 2016 | 2016 | 2899.99 | 9 | 6 |
| 7 | Trek Slash 8 27.5 - 2016 | 2016 | 3999.99 | 9 | 6 |
| 9 | Trek Conduit+ - 2016 | 2016 | 2999.99 | 9 | 5 |
| 40 | Trek Fuel EX 9.8 29 - 2017 | 2017 | 4999.99 | 9 | 6 |
| 43 | Trek Fuel EX 9.8 27.5 Plus - 2017 | 2017 | 5299.99 | 9 | 6 |
| 47 | Trek Remedy 9.8 - 2017 | 2017 | 5299.99 | 9 | 6 |
| 49 | Trek Domane SL 6 - 2017 | 2017 | 3499.99 | 9 | 7 |
| 50 | Trek Silque SLR 7 Women's - 2017 | 2017 | 5999.99 | 9 | 7 |
| 51 | Trek Silque SLR 8 Women's - 2017 | 2017 | 6499.99 | 9 | 7 |
| 54 | Trek Domane SL Disc Frameset - 2017 | 2017 | 3199.99 | 9 | 7 |
| 56 | Trek Domane SLR 6 Disc - 2017 | 2017 | 5499.99 | 9 | 7 |
| 58 | Trek Madone 9.2 - 2017 | 2017 | 4999.99 | 9 | 7 |
| 61 | Trek Powerfly 8 FS Plus - 2017 | 2017 | 4999.99 | 9 | 5 |
| 62 | Trek Boone 7 - 2017 | 2017 | 3499.99 | 9 | 4 |
| 63 | Trek Boone Race Shop Limited - 2017 | 2017 | 3499.99 | 9 | 4 |
| 115 | Trek Fuel EX 8 29 - 2018 | 2018 | 3199.99 | 9 | 6 |
| 139 | Trek Remedy 7 27.5 - 2018 | 2018 | 2999.99 | 9 | 6 |
| 140 | Trek Remedy 9.8 27.5 - 2018 | 2018 | 4999.99 | 9 | 6 |
| 142 | Trek Fuel EX 8 29 XT - 2018 | 2018 | 3199.99 | 9 | 6 |
| 146 | Trek Domane SLR 6 - 2018 | 2018 | 4999.99 | 9 | 7 |
| 148 | Trek Domane SL 8 Disc - 2018 | 2018 | 5499.99 | 9 | 7 |
| 149 | Trek Domane SLR 8 Disc - 2018 | 2018 | 7499.99 | 9 | 7 |
| 150 | Trek Emonda SL 7 - 2018 | 2018 | 4499.99 | 9 | 7 |
| 153 | Trek Domane SL 7 Women's - 2018 | 2018 | 4999.99 | 9 | 7 |
| 154 | Trek Domane SLR 6 Disc Women's - 2018 | 2018 | 5499.99 | 9 | 7 |
| 155 | Trek Domane SLR 9 Disc - 2018 | 2018 | 11999.99 | 9 | 7 |
| 156 | Trek Domane SL Frameset - 2018 | 2018 | 6499.99 | 9 | 7 |
| 157 | Trek Domane SL Frameset Women's - 2018 | 2018 | 6499.99 | 9 | 7 |
| 160 | Trek Emonda SLR 6 - 2018 | 2018 | 4499.99 | 9 | 7 |
| 162 | Trek Emonda SL 6 Disc - 2018 | 2018 | 2999.99 | 9 | 7 |
| 169 | Trek Emonda SLR 8 - 2018 | 2018 | 6499.99 | 9 | 7 |
| 171 | Trek Domane SL 6 - 2018 | 2018 | 3199.99 | 9 | 7 |
| 172 | Trek Domane ALR Disc Frameset - 2018 | 2018 | 3199.99 | 9 | 7 |
| 173 | Trek Domane ALR Frameset - 2018 | 2018 | 3199.99 | 9 | 7 |
| 174 | Trek Domane SLR Disc Frameset - 2018 | 2018 | 3199.99 | 9 | 7 |
| 175 | Trek Domane SLR Frameset - 2018 | 2018 | 3199.99 | 9 | 7 |
| 176 | Trek Madone 9 Frameset - 2018 | 2018 | 3199.99 | 9 | 7 |
| 177 | Trek Domane SLR 6 Disc - 2018 | 2018 | 5499.99 | 9 | 7 |
| 184 | Trek Domane SL 6 Disc - 2018 | 2018 | 3499.99 | 9 | 7 |
| 186 | Trek CrossRip+ - 2018 | 2018 | 4499.99 | 9 | 5 |
| 188 | Trek XM700+ Lowstep - 2018 | 2018 | 3499.99 | 9 | 5 |
| 194 | Trek XM700+ - 2018 | 2018 | 3499.99 | 9 | 5 |
| 198 | Electra Townie Commute Go! - 2018 | 2018 | 2999.99 | 1 | 5 |
| 199 | Electra Townie Commute Go! Ladies' - 2018 | 2018 | 2999.99 | 1 | 5 |
| 200 | Trek Powerfly 5 - 2018 | 2018 | 3499.99 | 9 | 5 |
| 201 | Trek Powerfly 5 FS - 2018 | 2018 | 4499.99 | 9 | 5 |
| 202 | Trek Powerfly 5 Women's - 2018 | 2018 | 3499.99 | 9 | 5 |
| 203 | Trek Powerfly 7 FS - 2018 | 2018 | 4999.99 | 9 | 5 |
| 204 | Trek Super Commuter+ 7 - 2018 | 2018 | 3599.99 | 9 | 5 |
| 205 | Trek Super Commuter+ 8S - 2018 | 2018 | 4999.99 | 9 | 5 |
| 206 | Trek Boone 5 Disc - 2018 | 2018 | 3299.99 | 9 | 4 |
| 207 | Trek Boone 7 Disc - 2018 | 2018 | 3999.99 | 9 | 4 |
| 209 | Trek Crockett 7 Disc - 2018 | 2018 | 2999.99 | 9 | 4 |
| 251 | Electra Townie Commute Go! - 2018 | 2018 | 2999.99 | 1 | 3 |
| 252 | Electra Townie Commute Go! Ladies' - 2018 | 2018 | 2999.99 | 1 | 3 |
| 320 | Trek Checkpoint SL 6 - 2019 | 2019 | 3799.99 | 9 | 7 |
| 321 | Trek Checkpoint ALR Frameset - 2019 | 2019 | 3199.99 | 9 | 7 |

```
2016    3299.990000
2017    4774.990000
2018    4337.490000
2019    3499.990000
```

Order of execution runs the FROM and WHERE first and we get all the products. Then we get a group of data for each model_year with GROUP BY. Lastly the SELECT runs and gets the model year (and at this point we know that all model_years are the same in each group and we get the AVG of each model_year group on their own row. So we have a row for each year, because that is how we grouped them.

Let’s also introduce here the WITH ROLLUP clause. Let’s add it to the end of our GROUP BY since it is a sub clause of GROUP BY.

```
SELECT model_year, AVG(list_price)
FROM product
WHERE list_price > 2800
GROUP BY model_year WITH ROLLUP;
```

Now we see the average of all the averages.

I like to use it with SUM to get a grand total row.

```
SELECT model_year, SUM(list_price)
FROM product
WHERE list_price > 2800
GROUP BY model_year WITH ROLLUP;
```

We’ve used DISTINCT before when we want to only see unique results. We can also use it with aggregate functions as well.  Let’s put AVG back on the list_price and add the DISTINCT keyword. Now if there are list prices that are exactly the same they will only be included in the average one time.

```
SELECT model_year, AVG(DISTINCT list_price)
FROM product
WHERE list_price > 2800
GROUP BY model_year WITH ROLLUP;
```

This does change the 4 last years' averages. The default is *all* values like we have been using. The ALL keyword is assumed.

```
SELECT model_year, AVG(ALL list_price)
FROM product
WHERE list_price > 2800
GROUP BY model_year WITH ROLLUP;
```

It works well with count. Here we can count all the list prices.

```
SELECT model_year, COUNT(ALL list_price)
FROM product
WHERE list_price > 2800
GROUP BY model_year WITH ROLLUP;
```

It's the same thing without the keyword ALL since that is assumed.

```
SELECT model_year, COUNT(list_price)
FROM product
WHERE list_price > 2800
GROUP BY model_year WITH ROLLUP;
```

And if we add DISTINCT we count only the unique list prices for each year.

```
SELECT model_year, COUNT(DISTINCT list_price)
FROM product
WHERE list_price > 2800
GROUP BY model_year WITH ROLLUP;
```

OK so one more new clause to introduce here. The HAVING clause. The HAVING clause is also a filter but it will filter after the groups have been made. So sometimes you can filter before we even group and that would be with a WHERE clause. But if we want to filter after the data has been grouped we use a HAVING clause.

```
SELECT model_year, AVG(list_price)
FROM product
WHERE list_price > 2800
GROUP BY model_year
HAVING AVG(list_price) > 4000;
```

We could filter by another column like product_name that is not included in the SELECT list in the WHERE but not the HAVING. It is limited to only what is in the SELECT list. Let's look at our image again. This HAVING would eliminate two rows because the average list price of 2016 and 2019 are not greater than 4000. That leave us with the two rows that are over that amount.

We can make our average number look more like currency using FORMAT again in the SELECT listing. But notice if I try and use the FORMAT function with the nested AVG inside in the HAVING it's not working right and we don't get any results

```
SELECT model_year, FORMAT(AVG(list_price), 2)
FROM product
WHERE list_price > 2800
GROUP BY model_year
HAVING FORMAT(AVG(list_price), 2) > 4000;
```

That's because FORMAT change the data type of decimal to a string with a comma in the amounts over 1000. So, when you compare a string to greater than 4000 it gets no results. So, let's take that FORMAT out of the HAVING.

Remember, the HAVING clause can only use what's listed in the SELECT statement because those are the values or aggregates that are part of the groups. But the WHERE clause can refer to any column in the base table even if it's not listed in the SELECT. Here we can filter with the WHERE by product_name even if it's not in the SELECT list of columns or aggregates. But I couldn't refter to product_name in the HAVING. Only model year or the AVG(list_price aggregate.

Also, you wouldn't be able to use the aggregate function in a WHERE clause, only non-aggregates. Because at that point in the order of execution the aggregate hasn't even run yet.

```
SELECT model_year, FORMAT(AVG(list_price), 2)
FROM product
WHERE product_name LIKE '%speed%'
GROUP BY model_year
HAVING AVG(list_price) > 300;
```

What if I wanted to get the average of each brand? So I want to see the brand name and then the average of each brand.

```
SELECT brand.brand_name, FORMAT(AVG(list_price), 2)
FROM product
  JOIN brand
          ON product.brand_id = brand.brand_id
GROUP BY brand.brand_name WITH ROLLUP;
```

Let’s try one that includes a WHERE and a HAVING. We want a result set that shows the brand name and the average list price for each of those brand names. We don’t want to include any 2016 model year bikes in this average. We also want to only see average list price values over $2000.

```
SELECT brand.brand_name, FORMAT(AVG(list_price), 2) AS average
FROM product
  JOIN brand
            ON product.brand_id = brand.brand_id
WHERE model_year > '2016'
GROUP BY brand.brand_name WITH ROLLUP
HAVING AVG(list_price) > 2000;
```

Let’s practice with the magazine database.

```
  USE magazine;
```

Let’s find out how many people subscribe to Beautiful Birds magazine.

```
SELECT COUNT(subscriberKey) AS subscribers
FROM magazine m
  JOIN subscription sn
            ON m.magazineKey = sn.magazineKey
WHERE magazineName = "Beautiful Birds";
```

I let’s add the magazineName because the WHERE limits it to only one magazine so I am good with that column.

```
SELECT magazineName, COUNT(subscriberKey) AS subscribers
FROM magazine m
  JOIN subscription sn
            ON m.magazineKey = sn.magazineKey
WHERE magazineName = "Beautiful Birds";
```

What if we want how many subscribers each magazine has?

```
SELECT magazineName, COUNT(subscriberKey) AS subscribers
FROM magazine m
  JOIN subscription sn
            ON m.magazineKey = sn.magazineKey
GROUP BY magazineName;
```

For example, say we want just those magazines having 2 or more subscribers.

```
SELECT magazineName, COUNT(subscriberKey) AS subscribers
FROM magazine m
        JOIN subscription sn
                ON m.magazineKey = sn.magazineKey
GROUP BY magazineName
HAVING subscribers >= 2;
```

Again, we will filter after the data after it has been grouped with a HAVING clause.

What would the total revenue be for each magazine?

```
        SELECT magazineName, COUNT(subscriberKey) AS subscribers, SUM(magazinePrice)
        AS 'Total Revenue'
        FROM magazine m
          JOIN subscription sn
                ON m.magazineKey = sn.magazineKey
        GROUP BY magazineName
        ORDER BY magazineName;
```

Let's practice a few more with the bike database.

```
        USE bike;
```

Let's get the brand name and highest and lowest prices of bikes in each brand.

```
        SELECT brand_name, MAX(list_price), MIN(list_price)
        FROM product
          JOIN brand
                ON product.brand_id = brand.brand_id
        GROUP BY brand_name;
```

Let's find out how many of each product we have at each store.

Let's start out with just seeing how many bikes we have total in stock.

```
        SELECT SUM(quantity)
        FROM;
```

Now we could group it by product name to see how many of each bike we have. Because we want the name of the bike we are joining the product table.

```
        SELECT product_name, SUM(quantity)
```

```
FROM stock st
   JOIN product p
   ON st.product_id = p.product_id
GROUP BY product_name WITH ROLLUP;
```

Notice the WITH ROLLUP gives us the same total we got with the last query.

Now let's take it even one step further and find out how many, not only of each bike, but from each store. Again if we want to show the name of the store we join the store table. And the grouping is not only by the product name but also a sub grouping of the store.

```
SELECT store_name, product_name, SUM(quantity)
   FROM store s
            JOIN stock st
                    ON s.store_id = st.store_id
            JOIN product p
                    ON st.product_id = p.product_id
GROUP BY product_name, store_name;
```

With ROLLUP shows a total of each bike at all the stores and if you scroll to the bottom it shows a grand total of all bikes.

```
SELECT store_name, product_name, SUM(quantity)
   FROM store s
            JOIN stock st
                    ON s.store_id = st.store_id
            JOIN product p
                    ON st.product_id = p.product_id
GROUP BY product_name, store_name WITH ROLLUP;
```

If we reverse the grouping with store name as group and product as the sub group we get the total of how many bikes we have at each store.

```
SELECT store_name, product_name, SUM(quantity)
   FROM store s
            JOIN stock st
                    ON s.store_id = st.store_id
            JOIN product p
                    ON st.product_id = p.product_id
GROUP BY product_name, store_name WITH ROLLUP;
```

Now we could easily see how many of one specific bike is at any of our stores by adding a where clause.

```
SELECT store_name, product_name, SUM(quantity)
        FROM store s
                JOIN stock st
                        ON s.store_id = st.store_id
                JOIN product p
                        ON st.product_id = p.product_id
WHERE product_name LIKE "Electra Cruiser 1 Ladies%"
GROUP BY product_name, store_name;
```

So we can see that we can get very detailed and specific information from our data using aggregate functions.