03: Aggregation

Aggregating data is at the heart of where SQL's utility will shine. Crafting aggregation and calculation queries in SQL allows you to also combine in the robust filtering and selection into a single query. This sounds like a lot of information going into a single query, but the compartmentalized structure of SQL queries maintains a aood amount of readability.

Having all your selection, filtering, aggretation, and other transformation code all in one place helps you document you analysis with more certainty than working in just a spreadsheet. Most of the time, each query will presume that you are working from the same base set of raw data and transforming it each time to what you need.

At the core of aggregation is to tell you more than just your raw data, and is where you can start answering a lot of questions that you may have.

`COUNT()`

```
SELECT COUNT(*) FROM pettigrew;
```

This returns a single result: `601`. Knowing the rest of our data, can we take a guess as to what happened?

`COUNT()` is a function that will count the number of rows returned from your query. In this case, we've just passed it the `*` wildcard because it doesn't matter which columns we select -- it is only counting the rows.

Most of the aggregation functions will want to work with continuous data, such as measurements. But if what you have is mostly text codes or categories, you might be sticking with `COUNT()` and doing other aggregation activities on categorical or group columns.

You can combine `COUNT()` with other keywords. For example, how many unique BoxNumber values are there? We know there are 601 total records, but we want to count the number of BoxNumbers mentioned.

This query involves two actions:

1. Construct a query that displays the unique names for BoxNumber.
2. Add `COUNT()` in there to count how many there are.

For step 1:

This query will select all the unique values returned from a query only looking at the BoxNumber column.

```
SELECT DISTINCT BoxNumber FROM pettigrew;
```

For step 2:

The question here is where should COUNT() go. In our previous example, we had COUNT(*) so our first instinct might be to put it just around BoxNumber. Let's go ahead and try that:

```
SELECT DISTINCT COUNT(BoxNumber) FROM pettigrew;
```

This gives us a result of 601, which we know to be the total count of all our rows. What can this tell us?

We know that something went wrong but SQL isn't being noisy about it. We only know that something has gone wrong because we know our data very well.

Recall that DISTINCT operates on the row level. As in, it only returns the unique rows. We also know that COUNT returns back however many rows were returned.

So putting this all together, we can take a guess that COUNT() operated first and returned a count of all of BoxNumber. Since we had no other filters in place, it spat back all 601 rows. So then DISTINCT had really nothing to operate on.

Let's try moving DISTINCT in with BoxNumber.

```
SELECT COUNT(DISTINCT BoxNumber) FROM pettigrew;
```

Good, we have a count of 13 in there, and we know that there are 13 box numbers.

`GROUP BY`

You won't always want to count the number of all results coming up from a single column. The `GROUP BY` keyword allows us to refine our `COUNT()` or other aggregation statements to create groups or subsets that are further counted. This allows us some similar tools like a PivotTable does.

Whereas `COUNT()` can exist without `GROUP BY` and acts on the entire set of results, `GROUP BY` cannot exist without some aggregation function. Well, it will technically work, but do silly things.

```
SELECT * FROM pettigrew GROUP BY BoxNumber;
```

The results of this query are composed of the last result of each row in each of the BoxNumber categories. You can visualize this in your head, of imagine running a `WHERE` clause for each of the BoxNumbers. Then take the last row of each of those results. That's what we're seeing here.

SQL is a very forgiving language, where it will desperately attempt to execute what you tell it, even if the results are nonsense. This means that you need to know your data very well so you can keep an eye out for these kinds of issues.

So let's get back to making an actual aggregation statement that works. Remember that it first performs the grouping, and then it performs the aggregation on each of those groups.

```
SELECT COUNT(*) as NumLettersInEachBox FROM pettigrew GROUP BY BoxNumber;
```

This does a few actions:

1. It selects all the columns, which is fine.
2. It groups all the data by the BoxNumber value. So knowing the data, you should have 13 groups.
3. It counts how many results are inside of each group and gives you that number for each value.

When performing these types of actions, it can be helpful to also report back the value of the column you've grouped by. Remember that it'll take the last value for each group, but the contents should all be the same, so that's fine.

Here's our final statement for this:

```
SELECT BoxNumber, COUNT(*) as NumLettersInEachBox FROM pettigrew GROUP BY BoxNumber;
```

Again, remember that you can do silly things here.

```
SELECT FolderNumber, COUNT(*) as NumLettersInEachBox FROM pettigrew GROUP BY BoxNumber;
```

This results in the last folder number for each group. Maybe that's what you want, but likely not.

Our basic formula is:

1. Pick the column you want to do math to. This will likely the name of the column you're going to group by if you're just counting instances.
2. Pick the aggregation function you want to do. Be sure to read the documentation and understand how that function will operate on the data type of the column you're going to give it.
3. Pick the column that you want to aggregate by (this is usually some categorical variable). Do remember that this will group by unique content, so if you have typos or uncontrolled values, each unique value will be treated as a unique instance.

Thus our aggregation madlib is `SELECT AGGREGATION_FUNCTION(math_column) FROM database GROUP BY categorical_column;`.

When you're preparing for analysis or even just setting up your data, it can be very helpful to mark up or make a note of the measurement variables and categorical variables within the question that you're trying to answer.

Adding more functions in

Now that we can group our data around based on the BoxNumber, let's learn more about our folder structures.

As always, start with a broader SQL statement that you know works before adding in more detail. In this case, let's set up a basic group by statement that counts the number of letters in each Box.

```
SELECT BoxNumber, COUNT(*) FROM pettigrew GROUP BY CAST(BoxNumber as numeric);
```

The `CAST()` statement is in there on the BoxNumber grouping so that it sorts correctly.

Since the FolderNumber appears to have meaning within each folder, let's look at the spread of the folder numbers within each box. Meaning, let's look at what the `min` and `max` values are for each group of box letters.

Challenge: add `MIN()` AND `MAX()` into your query. Leave the `COUNT()` column in there. You should have a table with 13 rows and 4 columns.

```
SELECT BoxNumber, COUNT(*), MIN(CAST(FolderNumber as numeric)), MAX(CAST(FolderNumber as numeric)) FROM pettigrew GROUP BY CAST(BoxNumber as
```

Subqueries

There are many calculation functions available, but may not always work for your data. For example:

```
SELECT AVG(CAST(FolderNumber as numeric)) FROM pettigrew;
```

This gives a result of `304.02` (rounded), but the meaning is kind of silly. These are just numerical codes for folder numbers, so the average isn't a helpful statistic.

However, what about the average number of letters inside of each box? That's something a bit more interesting.

This is where sometimes our logic of what's happening doesn't always mesh up with SQL. We've got a column we're computing the number of letters in each bin. So your natural instinct

might be to put `AVG(COUNT(*))` in there, but that's not exactly what `AVG()` wants.

There are several other ways to do this, but we're going to highlight the subquery method.

Remember how you can use `()` to declare pieces of boolean queries to be executed before other parts of the query? You can extend this by embedding other SQL queries into your statement. For example, if you want to look up the IDs of movies an actor is in, and then look up the earnings for each. You can use a subquery for that.

In this case, we want to construct a subquery that produces the column(s) that we want to compute on.

```
SELECT count(BoxNumber) AS boxCount FROM pettigrew group by BoxNumber
```

The `AS` keyword is here so we can directly reference this column name in our next query.

Imagine if this result is our table and we want to compute the average? We'd want `AVG(boxCount)` in this case. We can then replace our database name with our subquery.

So in our template: `SELECT AVG(boxCount) from (subquery);` we end up with this:

```
select AVG(boxCount) from (SELECT count(BoxNumber) as boxCount FROM pettigrew group by BoxNumber);
```

Which yields `46.23` (rounded).