Basic and Aggregate Functions

* In the previous unit we learned all about SELECT statements. Now that you’ve learned how to segment the data in various ways, you want to be able to aggregate the data and perform calculations on it. SQL provides many aggregation and other advanced query functions to do that
* Being able to aggregate your data is essential to the analysis workflow. It allows you to summarize and generally get a better idea of your data. It also allows you to track metrics over time or for different segments of your data. Aggregate functions are used to compute against a "returned column of numeric data" from your SELECT statement. They basically summarize the results of a particular column of selected data. We are covering these first since they are required by the next lesson, GROUP BY. Although they are required for the GROUP BY clause, these functions can be used without the GROUP BY clause.
* Before we start learning new functions, let’s review some nomenclature you should be familiar with:
  + Operators are the math of SQL
    - Operators are divided into 4 categories: arithmetic, comparison, logical, and string. All categories follow the same rules found in math



* + Delimiters are the grammar of SQL
    - Delimiters are used to separate, or mark the start and end of items of data, similar to the punctuation found in grammar
    - Be careful not to omit delimiters where they are needed. These errors can be hard to trace (e.g. unbalanced parentheses or quotes, unmatched comment breaks, missing semicolons.)
    - 
  + You should become familiar with 2 other terms
    - Expressions
      * **Expression operators** create an expression, used ot change or modify the values that are returned.
      * Expressions in SQL generally fall into one of 4 categories:
        + Boolean
        + Numeric
        + Character
        + Date Expression
      * **Arguments** can be a literal value, a variable, or an expression in a SQL statement
  + Basic (Non-Aggregate) Calculations
    - You’ve learned how to SELECT data from a database using SQL – but what if the data isn’t structured exactly the way you’d like? That’s where basic (non-aggregate) calculations can help.
    - Let’s look at this sample data set.
      * What could you do if you wanted these track times in minutes, rather than seconds?



* + - * Or the full names of your employees, rather than their first and last names?
      * 
      * There are a few ways to customize your SQL results. Remember how the SELECT keyword is followed by a field name or a list of field names? Well, you can manipulate these fields using operators like the ones we learned in the last unit.
        + For example, we could **concatenate** (combine) two fields into one using the + operator.
        + So instead of writing:

SELECT first\_name, last\_name, job\_title

FROM employees

LIMIT 3;

* + - * + Which would return:



* + - * + We could **concatenate** (combine) two text fields into one using the + operator.
        + That means we would write:

SELECT first\_name + last\_name, job\_title

FROM employees

LIMIT 3;

* + - * + And it would return:



* + - * Notice how the names are spliced together using the addition sysmbol. Adding (+) text fields puts them together
      * TO add a space between these two values, we would have to write:

SELECT first\_name + ' ' + last\_name, job\_title

FROM employees

LIMIT 3;

* + - * 
    - You can also perform calculation in a SELECT clause.
      * For example, let’s say we wanted to know how many of their 21 vacation days each employee has left. We would write:

SELECT first\_name + ' ' + last\_name, job\_title, 21 - vacation\_days

FROM employees

LIMIT 3;

* + - * + 
    - Our field names are getting complicated and long, too. SQL has a useful feature called **aliases**, which will help make our table more readable
      * TO create an alias, we use the AS keyword followed by a temporary name for our table or column.
      * So we could write our last query as:

SELECT first\_name + ' ' + last\_name AS full\_name,

job\_title,

21 - vacation\_days AS days\_left

FROM employees

LIMIT 3;



* Summary of Algebraic Functions in SQL
  + Algebraic functions help calculate values and can be used on fields like we just did:
    - “+” Adds two columns together
      * Syntax: column\_a + column\_b
      * Example: sales\_jan + sales\_feb gives the total number of sales for January and February
    - “-“ Subtracts one column from another.
      * Syntax: column\_a – column\_b
      * Example: sales\_jan – sales\_feb gives the difference in sales between January and February.
    - “\*” Multiplies two columns together
      * Syntax: column\_a \* column\_b
      * Example: number\_of\_products\_purchased \* product\_price would give the total cost of a customer’s order
    - “/” Divides one column by another.
      * Syntax: column\_a / column\_b
      * Example: total\_cost / number\_of\_products\_purchased would give the cost per product
  + Now looking back at our example, instead of calculating remaining vacation days, we could find the percentage of vacation days used of those allotted
    - To find the percentage of vacation days taken by each employee, we would write:

SELECT first\_name + ' ' + last\_name AS full\_name,

job\_title,

vacation\_days / 21 AS vacation\_used

FROM employees

LIMIT 3;

* + - 
    - But is 0.23809524 really a useful percentage for understanding how much vacation Bianca Bryant has used? Not really.
      * For that, just like Excel, SQL allows us to ROUND long decimals
        + The ROUND function in SQL takes two arguments:

The field to round

The number of decimals to be returned

* + - * So this query:

SELECT first\_name + ' ' + last\_name AS full\_name,

job\_title,

ROUND(vacation\_days / 21, 2) AS vacation\_used

FROM employees

LIMIT 3;

* + - * Would yield:
      * 
  + Additional SQL mathematical functions can be used with aliases (AS) to modify returning values when the field has a numerical data type. Here’s a list of some of the most common:
    - ABS(): Find the absolute value
    - SIGN(): Return the sign for the numeric expression
      * -1 for negative expressions
      * 0 for zero expressions
      * +1 for positive expressions
      * NULL then the value is also NULL
    - MOD(numerator, divisor): Return the remainder from dividing two numbers (only the integer component)
      * Numerator: the number for which you want to find the remainder after the division is performed
      * Divisor: the number by which you want to divide
    - FLOOR(): Drop the decimal, just keep the integer component
    - CEILING(): Round the number up to the next-closest integer
    - SQRT(): Take the square root of the number
* Aggregate Functions
  + Instead of SELECT-ing large swathes of data, we’re sometimes going to want SQL to return a single value that describes the whole set
    - The functions we use to do this are called **aggregate functions** because they provide a summary of the data in a single row (aka an aggregate)
    - For example, say we have a table full of over 500 rows of data on different films. One way to describe this table is with the highest-grosing film
      * To find that out, we could use the MAX() function:

SELECT MAX(worldwide\_gross)

FROM films;

* + - * To find the lowest-grossing film, we could use the MIN() function:

SELECT MIN(worldwide\_gross)

FROM films;

* + - * Another way to summarize the data would be to add all the worldwide revenue together

SELECT SUM(worldwide\_gross)

FROM films;

* + - * To find the average gross revenue of all the films in our table, we would write:

SELECT AVG(worldwide\_gross)

FROM films;

* + - * What if we want to know how many films grossed above average?
        + We could use the COUNT function, which returns the number of rows in a given column, or using the star, the entire table

SELECT COUNT(\*)

FROM films

WHERE worldwide\_gross > 406349685.87;

* + - * But what if our data set contains duplicates?
        + Say, a row for every film release?
        + Steven Spielberg's E.T. would have three rows: one for its original release in 1982, a second for its 1985 re-release, and a third for the 20th anniversary re-release in 2002:



* + - * + Given this table, COUNT(title) would return three entries for E.T.
      * If we wanted to count the number of unique films in the table, we’d only want E.T> to count once.
        + We can do this by applying the keyword DISTINCT in our query, like so:

SELECT COUNT(DISTINCT title)

FROM films;

* + - * + Now COUNT includes just one entry for E.T.
    - Using these aggregate functions, we’ve just reduced 569 points of data into a simpler summary:



* + SUM
    - This function adds together the value for each row of a specified column. SUM is extremely useful for adding together the values of a numberic column
    - Syntax: SELECT SUM(<column\_name>) FROM table\_a
  + COUNT
    - This function counts the number of rows returned by a query.
    - COUNT is useful for counting the number of rows returned by your query. This is especially useful when combined with a WHERE clause that gives specific conditions. Then, you are getting a COUNT of occurrences of that specific condition
    - Syntax options:
      * SELECT COUNT (<column\_name>) FROM table\_a
      * SELECT COUNT (\*) FROM table\_a
  + COUNT DISTINCT
    - This function counts the number of DISTINCT rows returned by a query, not considering duplicate rows
    - Syntax: SELECT COUNT (DISTINCT <column\_name>) FROM table\_a
  + COUNT DISTINCT vs COUNT
    - COUNT DISTINCT is particularly useful for determining the number of unique occurrences of something. Like COUNT, it is especially useful when combined with a WHERE clause
  + MIN
    - This function returns the minimum value in a given column. It’s often useful to know the minimum value of a particular column in your data set. It gives you an idea of the lower limit of your data set.
    - Syntax: MIN(<column\_name>)
  + MAX
    - This function returns the maximum value in a given column. It’s often useful to know the maximum value of a particular column in your data set. It gives you an idea of the upper limit of your data set
    - Syntax: MAX(<column\_name>)
  + AVG
    - This function returns the average of all the values in a given column. It’s often useful to know the average of a particular column in your data set. It gives you a good value for comparison of the rest of the values in your data set
    - Syntax: AVG(<column\_name>)
* Knowing the minimum, maximum, and average values in a column can be very useful for identifying potential outliers.
  + If the majority of the values fall within a specific range, for example 0 to 100, but the maximum value is 10,000, that maximum might represent an outlier.
  + Similarly, if the data generally ranges from 0 to 100, but the minimum is -1,000, this might also represent a recording error (can you even have negative values?) or an outlier.
  + It can be difficult to decide whether to include or exclude outliers, and often domain expertise will help decide. But whichever you choose, have a logical reason to support your choice. Don’t just arbitrarily eliminate data to support your analysis.
* Rather than thinking about what function to use, first think about what you want to accomplish. Once you have a clear picture of what results you want to return, think about or search for what function to use to accomplish the task