**Daily XP700**

**Exercise**

**Aggregate functions and data types**

Aggregate functions are another valuable tool for the SQL programmer. They are used extensively across businesses to calculate important metrics, such as the average cost of making a film.

You know five different aggregate functions:

-AVG()

-SUM()

-MIN()

-MAX()

-COUNT()

Test your knowledge of what data types they are compatible with.

**Instructions**

**100XP**

* Classify the function based on what data type it is compatible with.

**Daily XP800**

##### Exercise

##### Exercise

# Practice with aggregate functions

Now let's try extracting summary information from a table using these new aggregate functions. Summarizing is helpful in real life when extracting top-line details from your dataset. Perhaps you'd like to know how old the oldest film in the films table is, what the most expensive film is, or how many films you have listed.

Now it's your turn to get more insights about the films table!

##### Instructions 1/4

**25 XP**

* [1](javascript:void(0))
  + Use the SUM() function to calculate the total duration of all films and alias with total\_duration.

 [2](javascript:void(0))

* Calculate the average duration of all films and alias with average\_duration.

 [3](javascript:void(0))

* Find the most recent release\_year in the films table, aliasing as latest\_year.

 [4](javascript:void(0))

* Find the duration of the shortest film and use the alias shortest\_film.
* -- Query the sum of film durations
* SELECT SUM(duration) AS total\_duration
* FROM films;

| **total\_duration** |
| --- |
| 534882 |

-- Calculate the average duration of all films

SELECT AVG(duration) AS average\_duration

FROM films;

| **average\_duration** |
| --- |
| 107.9479313824419778 |

-- Find the latest release\_year

SELECT MAX(release\_year) AS latest\_year

FROM films;

| **latest\_year** |
| --- |
| 2016 |

-- Find the duration of the shortest film

SELECT MIN(duration) AS shortest\_film

FROM films;

| **shortest\_film** |
| --- |
| 7 |

Well done! You'll find yourself using aggregate functions over and over again to get a quick grasp of the data in a SQL database.

**Daily XP900**

# Summarizing subsets

**50 XP**

## 1. Summarizing subsets

Well done! Now we will combine our filtering skills with our new summarizing skills.

## 2. Using WHERE with aggregate functions

We can combine aggregate functions with the WHERE clause to gain further insights from our data. That's because the WHERE clause executes before the SELECT statement. For example, to get the average budget of movies made in 2010 or later, we would select the average of the budget field from the films table where the release year is greater than or equal to 2010.

## 3. Using WHERE with aggregate functions

Here are a few more examples using the other functions: we find the total budget of movies made in 2010 using the SUM function, that's over 8.9 billion! Next, we get the smallest budget using the MIN function, which is 65,000.

## 4. Using WHERE with aggregate functions

Here, we query the highest budget using the MAX function. 600 million feels like a lot again for a movie budget, but this is in Indian Rupees for the movie "Kites". Finally, we query the count of the number of budgets using the COUNT function, which gives us the total number of non-missing values in the budget field, meaning there are 194 budgets recorded for the year 2010 in the films table.

## 5. ROUND()

Now that we are doing all sorts of things with our numerical values, we'll likely want to clean up some of the crazy decimals that might appear. In SQL, we can use ROUND() to round our number to a specified decimal. There are two parameters for ROUND(): the number we want to round and the decimal place we want to round to. Here we have re-calculated the same average budget as before, but this time we have included ROUND() and specified we want to round to two decimal places because we are dealing with currency.

## 6. ROUND() to a whole number

The second parameter in our ROUND() function is optional, so we can leave it out if we want to round to a whole number. We would get the same result if we passed zero as the second argument, as it is the default when no number is given.

## 7. ROUND() using a negative parameter

Here is a tricky one: we could also pass a negative number as the second parameter and still get a result. Here, the function is rounding to the left of the decimal point instead of the right. Using negative five as the decimal place parameter will cause the function to round to the hundred thousand or five places to the left. ROUND() can only be used with numerical fields.

## 8. Let's practice!

Let's practice!

**Daily XP950**

##### Exercise

##### Exercise

# Combining aggregate functions with WHERE

When combining aggregate functions with WHERE, you get a powerful tool that allows you to get more granular with your insights, for example, to get the total budget of movies made from the year 2010 onwards.

This combination is useful when you only want to summarize a subset of your data. In your film-industry role, as an example, you may like to summarize each certification category to compare how they each perform or if one certification has a higher average budget than another.

Let's see what insights you can gain about the financials in the dataset.

##### Instructions 1/4

**25 XP**

* [1](javascript:void(0))
  + Use SUM() to calculate the total gross for all films made in the year 2000 or later, and use the alias total\_gross.

 [2](javascript:void(0))

* Calculate the average amount grossed by all films whose titles start with the letter 'A' and alias with avg\_gross\_A.

 [3](javascript:void(0))

* Calculate the lowest gross film in 1994 and use the alias lowest\_gross.

 [4](javascript:void(0))

* Calculate the highest gross film between 2000 and 2012, inclusive, and use the alias highest\_gross.
* -- Calculate the sum of gross from the year 2000 or later
* SELECT SUM(gross) AS total\_gross
* FROM films
* WHERE release\_year >= 2000;

| **total\_gross** |
| --- |
| 150900926358 |

-- Calculate the average gross of films that start with A

SELECT AVG(gross) AS avg\_gross\_A

FROM films

WHERE title LIKE 'A%';

| **avg\_gross\_a** |
| --- |
| 47893236.422480620155 |

-- Calculate the lowest gross film in 1994

SELECT MIN(gross) AS lowest\_gross

FROM films

WHERE release\_year = 1994;

| **lowest\_gross** |
| --- |
| 125169 |

-- Calculate the highest gross film released between 2000-2012

SELECT MAX(gross) AS highest\_gross

FROM films

WHERE release\_year BETWEEN 2000 AND 2012;

| **highest\_gross** |
| --- |
| 760505847 |

Nice. SQL provides us with several building blocks that we can combine in all kinds of ways, hence the name: Structured Query Language.

**Daily XP1050**

##### Exercise

##### Exercise

# Using ROUND()

Aggregate functions work great with numerical values; however, these results can sometimes get unwieldy when dealing with long decimal values. Luckily, SQL provides you with the ROUND() function to tame these long decimals.

If asked to give the average budget of your films, ten decimal places is not necessary. Instead, you can round to two decimal places to create results that make more sense for currency.

Now you try!

##### Instructions

**100 XP**

* Calculate the average facebook\_likes to one decimal place and assign to the alias, avg\_facebook\_likes.
* -- Round the average number of facebook\_likes to one decimal place
* SELECT ROUND(AVG(facebook\_likes), 1) AS avg\_facebook\_likes
* FROM reviews;

| **avg\_facebook\_likes** |
| --- |
| 7802.9 |

Well done! The average, rounded facebook\_likes is 7802.9. This insight can be used as a benchmark to measure film reviews; any film with over 7802.9 likes can be considered popular.

**Daily XP1150**

##### Exercise

##### Exercise

# ROUND() with a negative parameter

A useful thing you can do with ROUND() is have a negative number as the decimal place parameter. This can come in handy if your manager only needs to know the average number of facebook\_likes to the hundreds since granularity below one hundred likes won't impact decision making.

Social media plays a significant role in determining success. If a movie trailer is posted and barely gets any likes, the movie itself may not be successful. Remember how 2020's "Sonic the Hedgehog" movie got a revamp after the public saw the trailer?

Let's apply this to other parts of the dataset and see what the benchmark is for movie budgets so, in the future, it's clear whether the film is above or below budget.

##### Instructions

**100 XP**

* Calculate the average budget from the films table, aliased as avg\_budget\_thousands, and round to the nearest thousand.
* -- Calculate the average budget rounded to the thousands
* SELECT ROUND(AVG(budget), -3) AS avg\_budget\_thousands
* FROM films;

| **avg\_budget\_thousands** |
| --- |
| 39903000 |

ROUND() of applause! The ROUND() function is very handy when making financial calculations to get a top-level view or specify to the penny or cent.

**Daily XP1250**

# Aliasing and arithmetic

**50 XP**

## 1. Aliasing and arithmetic

Welcome back! In this lesson, we will learn about using more arithmetic in our queries and take a closer look at aliasing with AS.

## 2. Arithmetic

We can perform basic arithmetic with symbols like plus, minus, multiply, and divide. Using parentheses with arithmetic indicates to the processor when the calculation needs to execute. Here are some basic examples of how we can use arithmetic in SQL. We can add, subtract, multiply, and divide as follows. In these examples, the parentheses are not required as only one calculation takes place; however, they provide more clarity to the code. But, the division gives a result of one; why is that?

## 3. Arithmetic

Similar to other programming languages, SQL assumes that we want to get an integer back if we divide an integer by an integer. So be careful! When dividing, we can add decimal places to our numbers if we want more precision. For example, SELECT four-point-zero divided by three-point-zero gives us the result we would expect: 1-point-3 repeating.

## 4. Aggregate functions vs. arithmetic

What's the difference between using aggregate functions and arithmetic? The key difference is that aggregate functions, like SUM, perform their operations on the fields vertically while arithmetic adds up the records horizontally.

## 5. Aliasing with arithmetic

Before we move on, let's run through an arithmetic example using our database. Here we have selected the gross, how much the movie made, minus the budget, how much the movie cost, from our films table. The result is the amount of profit. Notice that the query's result doesn't give us a defined field name. We will always need to use an alias when summarizing data with aggregate functions and arithmetic.

## 6. Aliasing with functions

As we progress and learn how to manipulate our data, it will be even more important to keep our field names clear. For example, if we're using multiple MAX functions in one query, we'll have two fields named max, which isn't very useful! This is a situation when it's especially important to alias like we do here.

## 7. Order of execution

Let's explore how using an alias fits into the SQL execution order. Here is a reminder of the order of execution we know so far: SQL will process the FROM statement first, followed by the WHERE clause, then the SELECT statement, and finally, LIMIT. When adding an alias for a field name in the SELECT clause, we might assume we could use it later in our query with the WHERE clause. Unfortunately, that is not possible; as we can see by the order of execution, the query would not have created the alias yet, and our code would generate an error.

## 8. Let's practice!

Time for more practice.

**Daily XP1300**

##### Exercise

##### Exercise

# Using arithmetic

SQL arithmetic comes in handy when your table is missing a metric you want to review. Suppose you have some data on movie ticket sales, but the table only has fields for ticket price and discount. In that case, you could combine these by subtracting the discount from the ticket price to get the amount the film-goer paid.

You have seen that SQL can act strangely when dividing integers. What is the result if you divide a discount of two dollars by the paid\_price of ten dollars to get the discount percentage?

##### Instructions

**50 XP**

##### Possible Answers

* 

2

* 

0.222

* Correct! SQL thinks we want the answer to be an integer since we are dividing two integers. 0 is the closest integer to 0.2.

**0**

* 

0.2

**Daily XP1350**

##### Exercise

##### Exercise

# Aliasing with functions

Aliasing can be a lifesaver, especially as we start to do more complex SQL queries with multiple criteria. Aliases help you keep your code clean and readable. For example, if you want to find the MAX() value of several fields without aliasing, you'll end up with the result with several columns called max and no idea which is which. You can fix this with aliasing.

Now, it's over to you to clean up the following queries.

##### Instructions 1/3

**35 XP**

* [1](javascript:void(0))
  + Select the title and duration in hours for all films and alias as duration\_hours; since the current durations are in minutes, you'll need to divide duration by 60.0.

 [2](javascript:void(0))

* Calculate the percentage of people who are no longer alive and alias the result as percentage\_dead.

 [3](javascript:void(0))

* Find how many decades the films table covers by using MIN() and MAX() and alias as number\_of\_decades.
* -- Calculate the title and duration\_hours from films
* SELECT title, \_\_\_
* FROM films;
* -- Calculate the title and duration\_hours from films
* SELECT  title, (duration/60.0) AS duration\_hours
* FROM films;

| **title** | **duration\_hours** |
| --- | --- |
| Intolerance: Love's Struggle Throughout the Ages | 2.0500000000000000 |
| Over the Hill to the Poorhouse | 1.8333333333333333 |
| The Big Parade | 2.5166666666666667 |
| Metropolis | 2.4166666666666667 |
| Pandora's Box | 1.8333333333333333 |
| The Broadway Melody | 1.6666666666666667 |
| Hell's Angels | 1.6000000000000000 |
| A Farewell to Arms | 1.3166666666666667 |
| 42nd Street |  |

-- Calculate the percentage of people who are no longer alive

SELECT \_\_\_ \* 100.0 / \_\_\_ AS percentage\_dead

FROM \_\_\_;

| **percentage\_dead** |
| --- |
| 12.7925877763328999 |

-- Find the number of decades in the films table

SELECT \_\_\_ / 10.0 AS \_\_\_

FROM films;

-- Find the number of decades in the films table

SELECT (MAX(release\_year) - MIN(release\_year)) / 10.0 AS number\_of\_decades

FROM films;

| **number\_of\_decades** |
| --- |
| 10.0000000000000000 |

Amazing work mastering arithmetic, aggregate functions, and aliasing! Now you know that our films table covers films released over ten decades, or a hundred years!

**Daily XP1450**

##### Exercise

##### Exercise

# Rounding results

You found some valuable insights in the previous exercise, but many of the results were inconveniently long. We forgot to round! We won't make you redo them all; however, you'll update the worst offender in this exercise.

##### Instructions

**100 XP**

* Update the query by adding ROUND() around the calculation and round to two decimal places.
* -- Round duration\_hours to two decimal places
* SELECT title, duration / 60.0 AS duration\_hours
* FROM films;
* - Round duration\_hours to two decimal places
* SELECT title, ROUND(duration / 60.0 , 2) AS duration\_hours
* FROM films;

| **title** | **duration\_hours** |
| --- | --- |
| Intolerance: Love's Struggle Throughout the Ages | 2.05 |
| Over the Hill to the Poorhouse | 1.83 |
| The Big Parade | 2.52 |
| Metropolis | 2.42 |
| Pandora's Box | 1.83 |
| The Broadway Melody | 1.67 |
| Hell's Angels | 1.60 |
| A Farewell to Arms |  |

That's better! Now you can clearly see how long a movie is.

**Daily XP1550**

# Sorting results

**50 XP**

## 1. Sorting results

Congratulations on making it this far! We'll now learn how to sort and group results to gain further insight.

## 2. Sorting results

Sorting results means we want to put our data in a specific order. It's another way to make our data easier to understand by quickly seeing it in a sequence. Let's say we wanted to extract our three longest coats; if our closet were messy, it would take a long time to find. However, if we sorted our closet by garment type and length, we could quickly grab them!

## 3. ORDER BY

In SQL, the ORDER BY keyword is used to sort results of one or more fields. When used on its own, it is written after the FROM statement, as shown here. ORDER BY will sort in ascending order by default. This can mean from smallest to biggest or from A to Z. In this case, we have one query sorting the budget from smallest to biggest and a second query sorting the titles alphabetically. Our database contains film titles that start with symbols and numbers; these come before the letter A.

## 4. ASCending

We could also add the ASC keyword to our query to clarify that we are sorting in ascending order. The results are the same, and our code is more readable.

## 5. DESCending

We can use the DESC keyword to sort the results in descending order. This query gives us the film titles sorted by budget from biggest to smallest. However, our data contains a lot of null values. We can add a WHERE clause before ORDER BY to filter the budget field for only non-null values and improve our results.

## 6. Sorting fields

Notice that we don't have to select the field we are sorting on. For example, here's a query where we sort by release year and only look at the title. However, it is a good idea to include the field we are sorting on in the SELECT statement for clarity.

## 7. ORDER BY multiple fields

ORDER BY can also be used to sort on multiple fields. It will sort by the first field specified, then sort by the next, etc. To specify multiple fields, we separate the field names with a comma. The second field we sort by can be thought of as a tie-breaker when the first field is not decisive in telling the order. Here is an example. Let's say we wanted to find the best movie. In the first query, we are only sorting the films by the number of Oscar wins and getting a tie. We can break that tie by adding a second sorting field by seeing which film has the most wins and the highest imdb\_score.

## 8. Different orders

We can also select a different order for each field we are sorting. For example, here, we are sorting birthdate in ascending order and name in descending order.

## 9. Order of execution

ORDER BY falls towards the end of the order of execution we already know, coming in just before limit. The FROM statement will execute first, then WHERE, followed by SELECT, ORDER BY, and finally, LIMIT.

## 10. Let's practice!

Time to practice our new sorting skills!

# Sorting text

SQL provides you with the ORDER BY keyword to sort one or more fields from your data. It can do this multi-directionally and helps make results easy to interpret.

How does ORDER BY sort a column of text values by default?

##### Answer the question

**50XP**

#### Possible Answers

* Correct! Because ascending is the default, indicating ASC is helpful for clarity but not required!.

**Alphabetically (A-Z)**

press1

* 

Reverse alphabetically (Z-A)

press2

* 

There's no natural ordering to text data

press3

* 

By number of characters (fewest to most)

press4

##### Exercise

# Sorting single fields

Now that you understand how ORDER BY works, you'll put it into practice. In this exercise, you'll work on sorting single fields only. This can be helpful to extract quick insights such as the top-grossing or top-scoring film.

The following exercises will help you gain further insights into the film database.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
  + Select the name of each person in the people table, sorted alphabetically.

 [2](javascript:void(0))

* Select the title and duration for every film, from longest duration to shortest.
* -- Select name from people and sort alphabetically
* SELECT name
* FROM people
* ORDER BY name ASC;

| **name** |
| --- |
| 50 Cent |
| A. Michael Baldwin |
| A. Raven Cruz |
| A.J. Buckley |
| A.J. DeLucia |
| A.J. Langer |
| AJ Michalka |
| Aaliyah |

-- Select the title and duration from longest to shortest film

SELECT title, duration

FROM films

ORDER BY duration DESC;

| **title** | **duration** |
| --- | --- |
| Destiny | null |
| Should've Been Romeo | null |
| Hum To Mohabbat Karega | null |
| Harry Potter and the Deathly Hallows: Part I | null |
| Barfi | null |
| Romantic Schemer | null |
| Wolf Creek | null |
| Dil Jo Bhi Kahey... | null |
| The Naked Ape |  |

**Daily XP200**

##### Exercise

##### Exercise

# Sorting single fields

Now that you understand how ORDER BY works, you'll put it into practice. In this exercise, you'll work on sorting single fields only. This can be helpful to extract quick insights such as the top-grossing or top-scoring film.

The following exercises will help you gain further insights into the film database.

##### Instructions 2/2

**50 XP**

* + Select the name of each person in the people table, sorted alphabetically.
  + Select the title and duration for every film, from longest duration to shortest.

+100 XP

Superb sorting! ORDER BY is another simple yet effective way to gain intelligence about your business and data. You now know how to extract your best and worst-performing assets with only a few lines of code.

##### ise

##### Exercise

# Sorting multiple fields

ORDER BY can also be used to sort on multiple fields. It will sort by the first field specified, then sort by the next, and so on. As an example, you may want to sort the people data by age and keep the names in alphabetical order.

Try using ORDER BY to sort multiple columns.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
  + Select the release\_year, duration, and title of films ordered by their release year and duration, in that order.

 [2](javascript:void(0))

* Select the certification, release\_year, and title from films ordered first by certification (alphabetically) and second by release year, starting with the most recent year.
* -- Select the release year, duration, and title sorted by release year and duration
* SELECT release\_year, duration, title
* FROM films
* ORDER BY release\_year, duration;

| **release\_year** | **duration** | **title** |
| --- | --- | --- |
| 1916 | 123 | Intolerance: Love's Struggle Throughout the Ages |
| 1920 | 110 | Over the Hill to the Poorhouse |
| 1925 | 151 | The Big Parade |
| 1927 | 145 | Metropolis |
| 1929 | 100 | The Broadway Melody |
| 1929 | 110 | Pandora's Box |
| 1930 | 96 | Hell's Angels |
| 1932 | 79 | A Farewell to Arms |

-- Select the certification, release year, and title sorted by certification and release year

SELECT certification, release\_year, title

FROM films

ORDER BY certification, release\_year;

| **certification** | **release\_year** | **title** |
| --- | --- | --- |
| Approved | 1933 | She Done Him Wrong |
| Approved | 1935 | Top Hat |
| Approved | 1936 | The Charge of the Light Brigade |
| Approved | 1937 | Snow White and the Seven Dwarfs |
| Approved | 1937 | The Prisoner of Zenda |
| Approved | 1938 | You Can't Take It with You |
| Approved | 1938 | Alexander's Ragtime Band |
| Approved | 1940 | Pinocchio |
| Approved | 1940 | The Blue Bird |
| Approved | 1941 | How Green Was My Valley |
| Approved | 1942 | Bambi |
| Approved | 1945 | State Fair |
| Approved | 1947 | Gentleman's Agreement |
| Approved | 1947 | Tycoon |
| Approved | 1948 | The Pirate |
| Approved | 1948 | Open Secret |
| Approved | 1948 | Red River |
| Approved | 1949 | Sands of Iwo Jima |
| Approved | 1951 | Show Boat |
| Approved | 1952 | Singin' in the Rain |
| G | 1936 | Modern Times |
| G | 1939 | Gone with the Wind |
| G | 1940 | Fantasia |
| G | 1962 | The Longest Day |
| G | 1965 | The Sound of Music |
| G | 1965 | The Greatest Story Ever Told |
| G | 1968 | 2001: A Space Odyssey |
| G | 1968 | Oliver! |

Nicely done! The second column you order on only steps in when the first column has been ordered.

# Grouping data

**50 XP**

## 1. Grouping data

We've learned how to sort data. Next, we'll look at grouping our results.

## 2. Grouping data

In the real world, we'll often need to summarize data for a particular group of results. For example, we might want to see the film data grouped by certification and make calculations on those groups, such as the average duration for each certification.

## 3. GROUP BY single fields

SQL allows us to group with the GROUP BY clause. Here it is used in a query where we have grouped by certification. GROUP BY is commonly used with aggregate functions to provide summary statistics, particularly when only grouping a single field, certification, and selecting multiple fields, certification and title. This is because the aggregate function will reduce the non-grouped field to one record only, which will need to correspond to one group.

## 4. Error handling

SQL will return an error if we try to SELECT a field that is not in our GROUP BY clause. We'll need to correct this by adding an aggregate function around title.

## 5. GROUP BY multiple fields

We can use GROUP BY on multiple fields similar to ORDER BY. The order in which we write the fields affects how the data is grouped. The query here selects and groups certification and language while aggregating the title. The result shows that we have five films that have missing values for both certification and language, two films that are unrated and in Japanese, two films that are rated R and in Norwegian, and so on.

## 6. GROUP BY with ORDER BY

We can combine GROUP BY with ORDER BY to group our results, make a calculation, and then order our results. For example, we can clean up one of our previous queries by sorting the results by the title count in descending order. Here is that query without ORDER BY, and this is the same query with ordering added. ORDER BY is always written after GROUP BY, and notice that we can refer back to the alias within the query. That is because of the order of execution. It looks like movies rated R are most common in our database.

## 7. Order of execution

GROUP BY fits into our order after FROM and before all other clauses. Our updated queries will begin with FROM, followed by grouping, selecting the data and creating the alias, sorting the results, and limiting them to the desired number.

## 8. Let's practice!

In the following exercises, we'll examine our film database to find out about release year, review, and budget patterns. Let's practice!

**Daily XP350**

##### Exercise

##### Exercise

# GROUP BY single fields

GROUP BY is a SQL keyword that allows you to group and summarize results with the additional use of aggregate functions. For example, films can be grouped by the certification and language before counting the film titles in each group. This allows you to see how many films had a particular certification and language grouping.

In the following steps, you'll summarize other groups of films to learn more about the films in your database.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
  + Select the release\_year and count of films released in each year aliased as film\_count.

 [2](javascript:void(0))

* Select the release\_year and average duration aliased as avg\_duration of all films, grouped by release\_year.
* -- Find the release\_year and film\_count of each year
* SELECT release\_year, COUNT(title) AS film\_count
* FROM films
* GROUP BY release\_year;

| **release\_year** | **film\_count** |
| --- | --- |
| 1954 | 5 |
| 1988 | 31 |
| 1959 | 3 |
| 1964 | 10 |
| 1969 | 10 |
| null | 42 |
| 2008 |  |

-- Find the release\_year and average duration of films for each year

SELECT release\_year, AVG(duration) AS avg\_duration

FROM films

GROUP BY release\_year;

| **release\_year** | **avg\_duration** |
| --- | --- |
| 1954 | 140.6000000000000000 |
| 1988 | 107.0000000000000000 |
| 1959 | 136.6666666666666667 |
| 1964 | 119.4000000000000000 |
| 1969 | 126.0000000000000000 |
| null | 77.4390243902439024 |
| 2008 | 105.3822222222222222 |
| 1991 | 113.0645161290322581 |

Great job grouping! Using GROUP BY with a time or date field such as release\_year can help us identify trends such as a period of time where movies were really short!

**Daily XP450**

##### Exercise

##### Exercise

# GROUP BY multiple fields

GROUP BY becomes more powerful when used across multiple fields or combined with ORDER BY and LIMIT.

Perhaps you're interested in learning about budget changes throughout the years in individual countries. You'll use grouping in this exercise to look at the maximum budget for each country in each year there is data available.

##### Instructions

**100 XP**

* Select the release\_year, country, and the maximum budget aliased as max\_budget for each year and each country; sort your results by release\_year and country.
* -- Find the release\_year, country, and max\_budget, then group and order by release\_year and country
* SELECT release\_year, country, MAX(budget) AS max\_budget
* FROM films
* GROUP BY release\_year, country
* ORDER BY release\_year, country;

| **release\_year** | **country** | **max\_budget** |
| --- | --- | --- |
| 1916 | USA | 385907 |
| 1920 | USA | 100000 |
| 1925 | USA | 245000 |
| 1927 | Germany | 6000000 |
| 1929 | Germany | null |
| 1929 | USA | 379000 |
| 1930 | USA | 3950000 |
| 1932 | USA |  |

Well done! You can see how building on your SQL queries helps you gain more insights and detect trends in the data, such as how film budgets may change throughout the years.

**Daily XP550**

##### Exercise

##### Exercise

# Answering business questions

In the real world, every SQL query starts with a business question. Then it is up to you to decide how to write the query that answers the question. Let's try this out.

Which release\_year had the most language diversity?

Take your time to translate this question into code. We'll get you started then it's up to you to test your queries in the console.

"Most language diversity" can be interpreted as COUNT(DISTINCT \_\_\_). Now over to you.

##### Instructions

**50 XP**

##### Possible Answers

* 

2005

* 

1916

* 

**2006 Well done! The year 2006 had 16 distinct languages, that's more than any other year.**

* 

1990

SELECT release\_year, COUNT(DISTINCT(language)) AS count\_lang

FROM films

GROUP BY release\_year

ORDER BY count\_lang DESC;

| **release\_year** | **count\_lang** |
| --- | --- |
| 2006 | 16 |
| 2015 | 15 |
| 2005 | 14 |
| 2013 | 13 |
| 2008 | 13 |
| 2009 | 12 |
| 2004 | 12 |
| 2007 | 11 |

**Daily XP600**

# Filtering grouped data

**50 XP**

## 1. Filtering grouped data

That was excellent work. We've combined sorting and grouping; next, we will combine filtering with grouping.

## 2. HAVING

In SQL, we can't filter aggregate functions with WHERE clauses. For example, this query attempting to filter the title count is invalid. That means that if we want to filter based on the result of an aggregate function, we need another way. Groups have their own special filtering word: HAVING. For example, this query shows only those years in which more than ten films were released.

## 3. Order of execution

The reason why groups have their own keyword for filtering comes down to the order of execution. We've written a query using many of the keywords we have covered here. This is their written order, starting with SELECT, FROM films, WHERE the certification is G, PG, or PG-13, GROUP BY certification, HAVING the title count be greater than 500, ORDER BY title count, and LIMIT to three. In contrast, the order of execution is: FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY, and LIMIT. By reviewing this order, we can see WHERE is executed before GROUP BY and before any aggregation occurs. This order is also why we cannot use the alias with HAVING, but we can with ORDER BY.

## 4. HAVING vs WHERE

WHERE filters individual records while HAVING filters grouped records. We'll walk through two business questions here to show how to translate them into the correct filter. The first question is "What films were released in the year 2000?". This question does not indicate any sort of grouping. It asks to see only the titles from a specific year and can therefore be written as SELECT title, FROM films, WHERE release year equals 2000. The second question is, "In what years was the average film duration over two hours?". Straight away, we can see this question has a few more layers. Let's break down the question and query into smaller, easier-to-understand steps.

## 5. HAVING vs WHERE

This question requires us to return information about years, so we select the release year from the films table. Next, it asks for the average film duration, which tells us we need to place AVG(duration) somewhere. Since we do not need to provide any additional information around the duration on its own, it is unlikely we need to perform the aggregation within the SELECT clause, so we'll try the HAVING clause instead. The last part of the question indicates we need to filter on the duration. Since we can't filter aggregates with WHERE, this supports our theory about using HAVING! Finally, we need to add a GROUP BY into our query since we have selected a column that has not been aggregated. Recall the aggregate function will convert the duration values into one average value. Going back to the start of our question, we're interested in knowing the average duration per year, so we group it by release year. And there we have it!

## 6. Let's practice!

Let's improve our confidence with a final round of practice.

**Daily XP650**

##### Exercise

##### Exercise

# Filter with HAVING

Your final keyword is HAVING. It works similarly to WHERE in that it is a filtering clause, with the difference that HAVING filters grouped data.

Filtering grouped data can be especially handy when working with a large dataset. When working with thousands or even millions of rows, HAVING will allow you to filter for just the group of data you want, such as films over two hours in length!

Practice using HAVING to find out which countries (or country) have the most varied film certifications.

##### Instructions

**100 XP**

* Select country from the films table, and get the distinct count of certification aliased as certification\_count.
* Group the results by country.
* Filter the unique count of certifications to only results greater than 10.
* -- Select the country and distinct count of certification as certification\_count
* \_\_\_
* -- Group by country
* \_\_\_
* -- Filter results to countries with more than 10 different certifications
* \_\_\_
* -- Select the country and distinct count of certification as certification\_count
* SELECT country, COUNT(DISTINCT(certification)) AS certification\_count
* FROM films
* -- Group by country
* GROUP BY country
* -- Filter results to countries with more than 10 different certifications
* HAVING COUNT(DISTINCT(certification)) >10;

| **country** | **certification\_count** |
| --- | --- |
| USA | 12 |

Great job! The answer is USA with 12 different certifications.

**Daily XP750**

##### Exercise

##### Exercise

# HAVING and sorting

Filtering and sorting go hand in hand and gives you greater interpretability by ordering our results.

Let's see this magic at work by writing a query showing what countries have the highest average film budgets.

##### Instructions

**100 XP**

* Select the country and the average budget as average\_budget, rounded to two decimal, from films.
* Group the results by country.
* Filter the results to countries with an average budget of more than one billion (1000000000).
* Sort by descending order of the average\_budget.
* -- Select the country and average\_budget from films
* \_\_\_
* -- Group by country
* \_\_\_
* -- Filter to countries with an average\_budget of more than one billion
* \_\_\_
* -- Order by descending order of the aggregated budget
* \_\_\_
* -- Select the country and average\_budget from films
* SELECT country, ROUND(AVG(budget), 2) AS average\_budget
* FROM films
* -- Group by country
* GROUP BY country
* -- Filter to countries with an average\_budget of more than one billion
* HAVING AVG(budget) >1000000000
* -- Order by descending order of the aggregated budget
* ORDER BY average\_budget;

| **country** | **average\_budget** |
| --- | --- |
| Hungary | 1260000000.00 |
| South Korea | 1383960000.00 |

You did it! South Korea and Hungary seem to have pricey films... or do they? Actually, these budgets are pretty standard for their local currency.

**Daily XP850**

##### Exercise

##### Exercise

# All together now

It's time to use much of what you've learned in one query! This is good preparation for using SQL in the real world where you'll often be asked to write more complex queries since some of the basic queries can be answered by playing around in spreadsheet applications.

In this exercise, you'll write a query that returns the average budget and gross earnings for films each year after 1990 if the average budget is greater than 60 million.

This will be a big query, but you can handle it!

##### Instructions 1/4

**25 XP**

* [1](javascript:void(0))
  + Select the release\_year for each film in the films table, filter for records released after 1990, and group by release\_year.

 [2](javascript:void(0))

* Modify the query to include the average budget aliased as avg\_budget and average gross aliased as avg\_gross for the results we have so far.

 [3](javascript:void(0))

* Modify the query once more so that only years with an average budget of greater than 60 million are included.

 [4](javascript:void(0))

* Finally, order the results from the highest average gross and limit to one.
* -- Select the budget for films released after 1990 grouped by year
* SELECT release\_year
* FROM films
* GROUP BY release\_year
* HAVING release\_year > 1990;

| **release\_year** |
| --- |
| 2008 |
| 1991 |
| 2009 |
| 2005 |
| 2013 |
| 2003 |
| 2015 |

-- Modify the query to also list the average budget and average gross

SELECT release\_year, AVG(budget) AS avg\_budget, AVG(gross) AS avg\_gross

FROM films

WHERE release\_year > 1990

GROUP BY release\_year;

| **release\_year** | **avg\_budget** | **avg\_gross** |
| --- | --- | --- |
| 2008 | 41804885.572139303483 | 44573509.378109452736 |
| 1991 | 25176548.387096774194 | 53844501.666666666667 |
| 2009 | 37073287.037037037037 | 46207440.200000000000 |
| 2005 | 70323938.231527093596 | 41159143.290640394089 |
| 2013 | 40519044.915492957746 | 56158357.775401069519 |
| 2003 | 37208648.833333333333 | 48727746.723270440252 |
| 2015 | 39298329.217391304348 | 72573302.930555555556 |
| 1993 | 20729787.234042553191 | 45302091.413043478261 |

SELECT release\_year, AVG(budget) AS avg\_budget, AVG(gross) AS avg\_gross

FROM films

WHERE release\_year > 1990

GROUP BY release\_year

-- Modify the query to see only years with an avg\_budget of more than 60 million

HAVING AVG(budget) > 60000000;

| **release\_year** | **avg\_budget** | **avg\_gross** |
| --- | --- | --- |
| 2005 | 70323938.231527093596 | 41159143.290640394089 |
| 2006 | 93968929.577464788732 | 39237855.953703703704 |

SELECT release\_year, AVG(budget) AS avg\_budget, AVG(gross) AS avg\_gross

FROM films

WHERE release\_year > 1990

GROUP BY release\_year

HAVING AVG(budget) > 60000000

-- Order the results from highest to lowest average gross and limit to one

ORDER BY avg\_gross DESC

LIMIT 1;

| **release\_year** | **avg\_budget** | **avg\_gross** |
| --- | --- | --- |
| 2005 | 70323938.231527093596 | 41159143.290640394089 |

Superb work! SQL queries can get rather long, but breaking them down into individual clauses makes them easier to write.

3

### Case Study - Country Club

Save

3 - 6 Hours

82 Points

In this case study, you'll use MySQL, PHPMyAdmin, Juptyer Notebook, and SQLite to tackle a series of challenges on a database containing information about a country club. For this case study, you'll see what a database looks like on a nice interface like PHPMyAdmin, and connect to a local instance of the database using Python and SQLite. Data scientists use these methodologies to do work on a daily basis. You'll retrieve particular pieces of information and translate requests into SQL queries. Make use of the DataCamp courses from the previous subunits to succeed in this case study!

There are two difficulty tiers for this case study. The only difference between them is the amount of guidance we give in setting up the SQLite connection in Python. We always encourage students to challenge themselves, but it's also totally fine if you want more guidance while working on complex assignments like this one.

* [**Tier Two**](https://www.springboard.com/archeio/download/18a1b2578e084d51999efa6a9fd2144a/)**:** If you feel confident tackling the challenge of exploring data from PHPMyAdmin and establishing a connection to a local SQLite database, pick Tier Two.
* [**Tier One**](https://www.springboard.com/archeio/download/1bf0319530ab4b599eb7973508728921/): If you want some helpful pointers during that process, go for Tier One. The SQL queries will be the same for both tiers. In this project folder, you will also find two resources that will guide you through the setup of SQLite.

Once you’ve decided on your difficulty level, paste the relevant SQLFiles folder into your working directory on PHPMyAdmin. Open the SQLTasks file containing the instructions for your case study. Happy querying!

Top of Form

Paste URL(s) to your document(s):

Please ensure that the document url is shared with public accessibility. Not making your submission publicly accessible could increase the time to receive feedback

Top of Form



Bottom of Form

+ Add another link

Bottom of Form

You can submit a project multiple times.