Universidad del Valle de Guatemala
Base de Datos 1
Sergio Alonzo

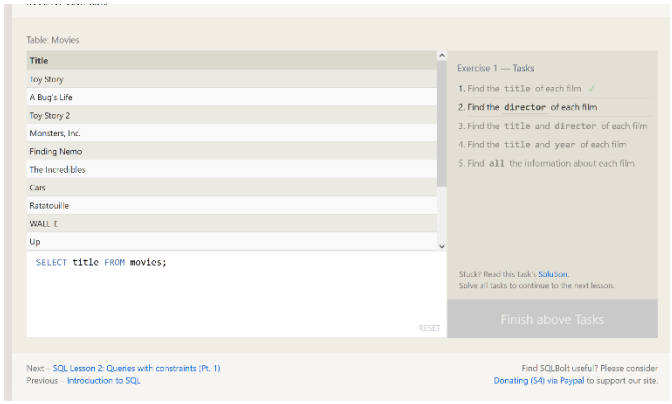
# Tarea 1

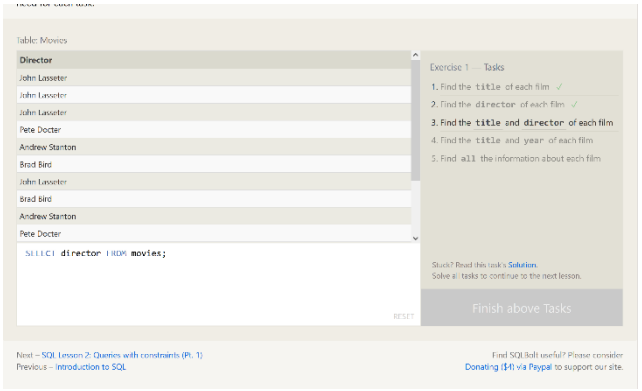
Alumno: Josúe Samuel Argueta Hernández 211024

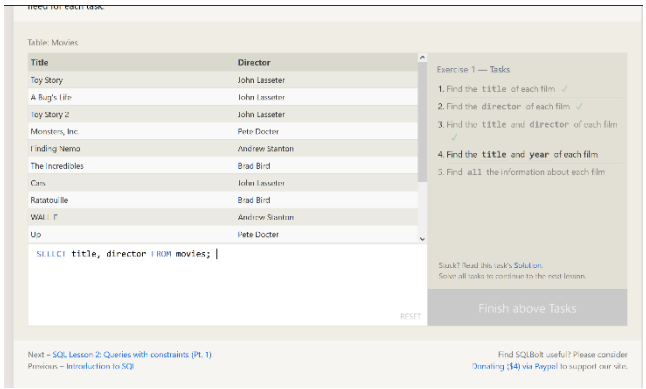
## Ejercicio 1: Tutorial de SQL (20 puntos)

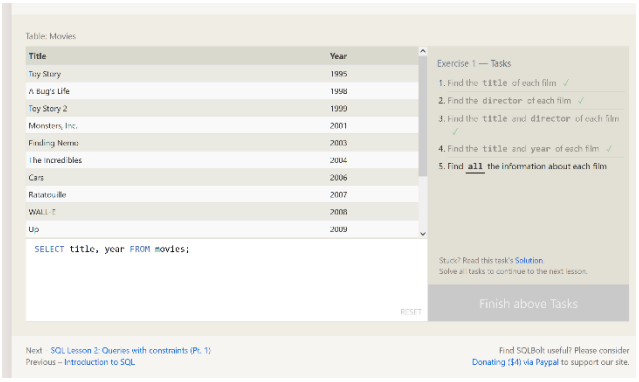
SQL es un lenguaje bastante común en bases de datos, por lo que existe mucho material de apoyo para su aprendizaje en línea[1].
Esta parte de la tarea consistirá en que lleve a cabo el siguiente tutorial para repasar los conceptos básicos de SQL vistos en la primera parte del curso:
https://sqlbolt.com/
Debe completar todas las lecciones del tutorial interactivo (Interactive tutorial) y leer los tópicos de Subqueries y Unions, Intersections & Exceptions.
En su entrega de la tarea debe registrar pantallazos de cada uno de los ejercicios realizados en el sitio web.

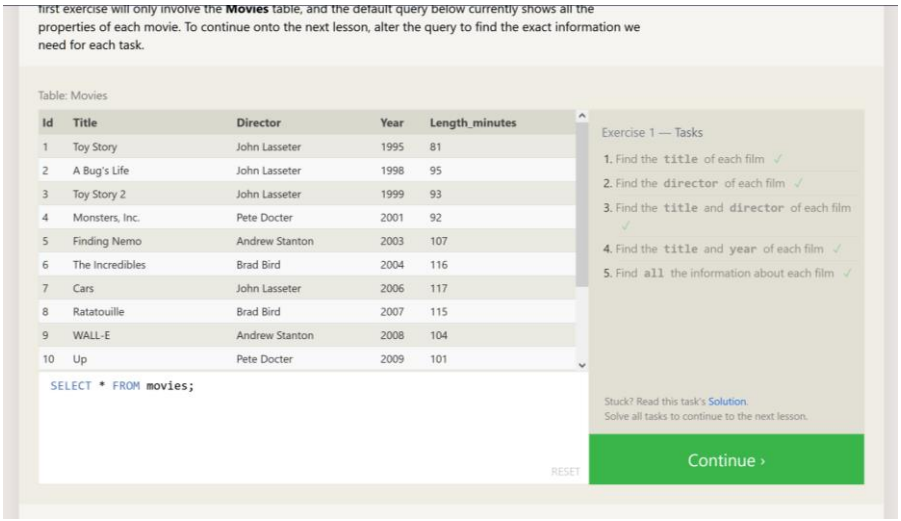
### SQL LESSON 1: SELECT QUERIES 101



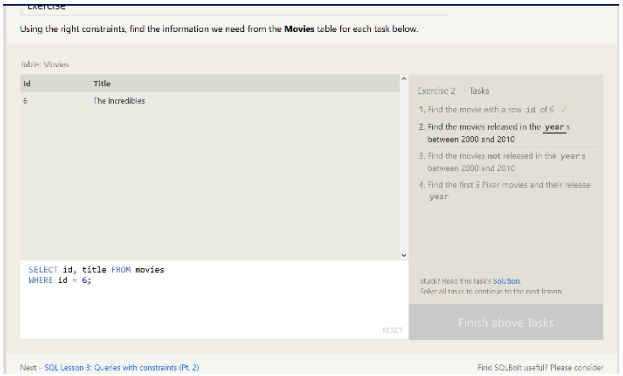



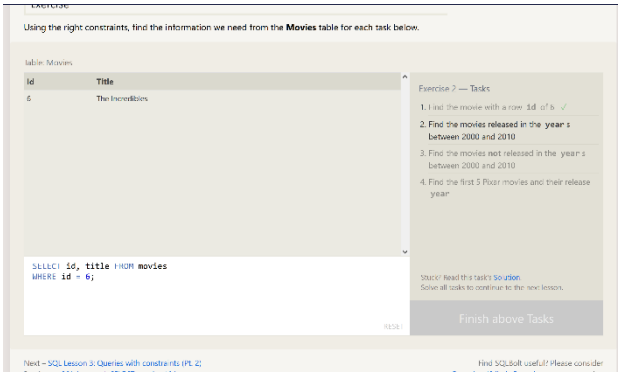



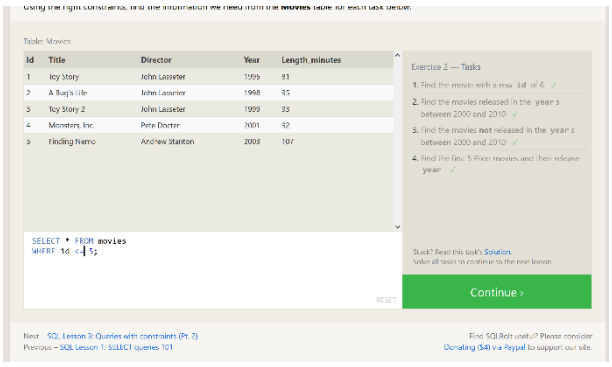



## SQL Lesson 2: Queries with constraints (Pt. 1)

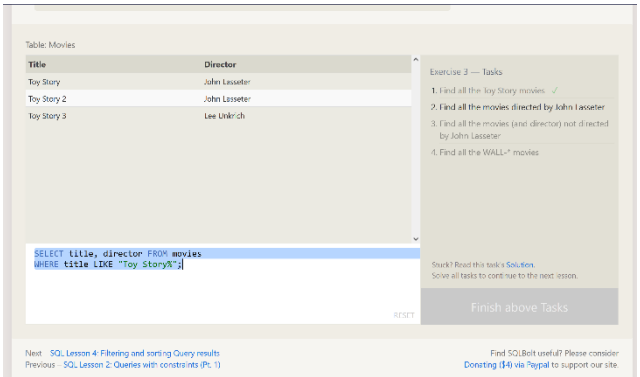


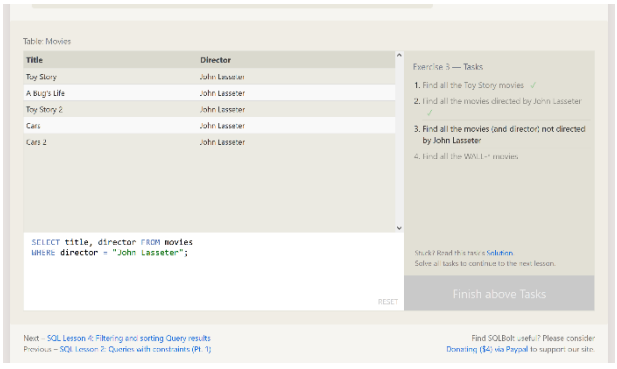



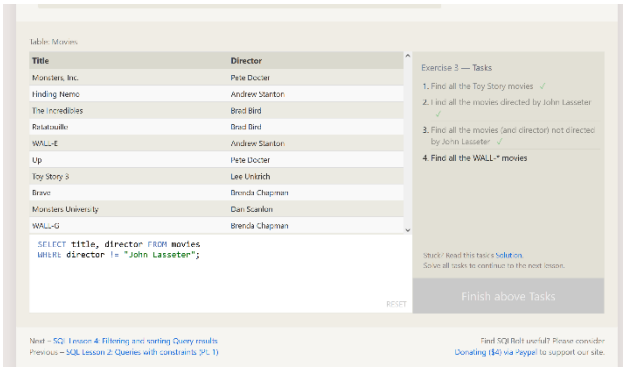


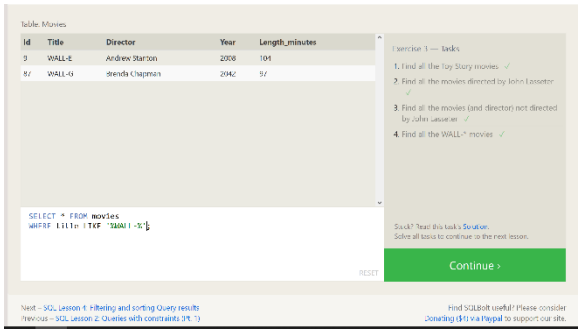



## SQL Lesson 3: Queries with constraints (Pt. 2)

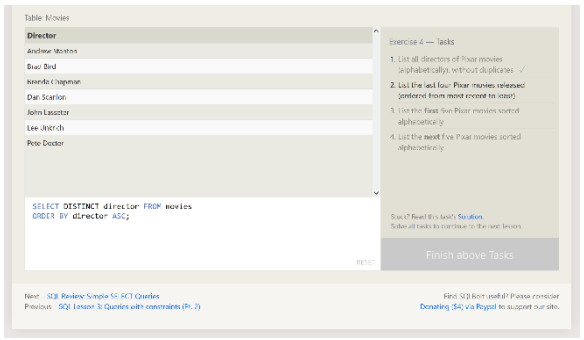


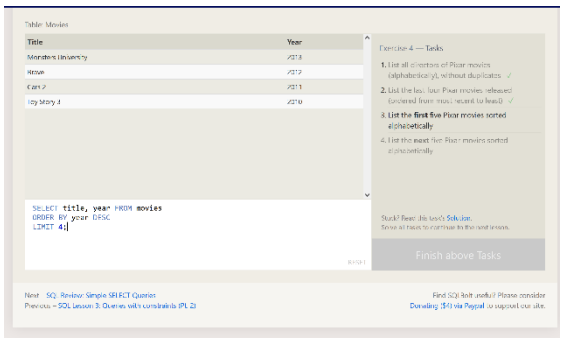



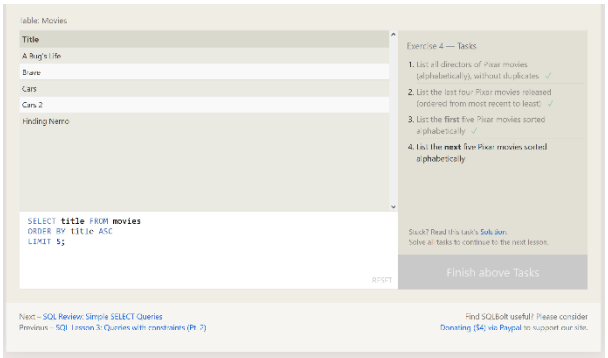


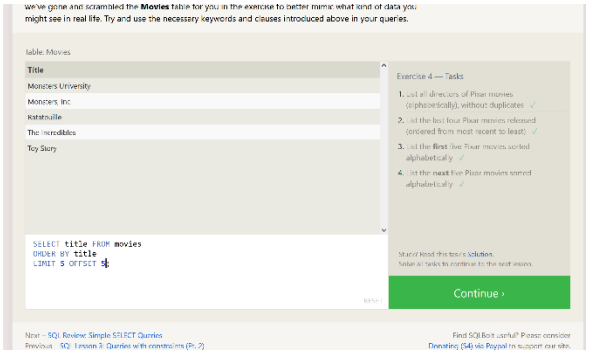


## SQL Lesson 4: Filtering and sorting Query results

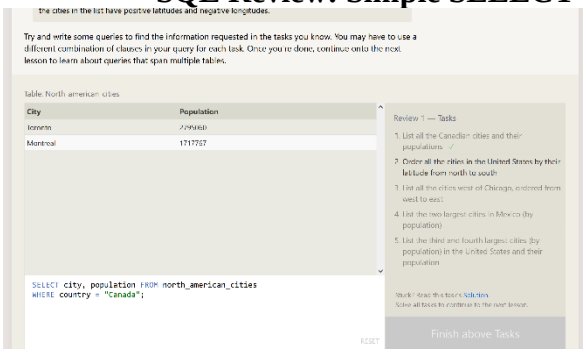


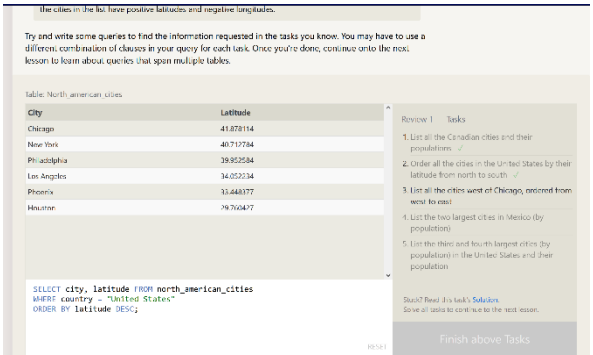


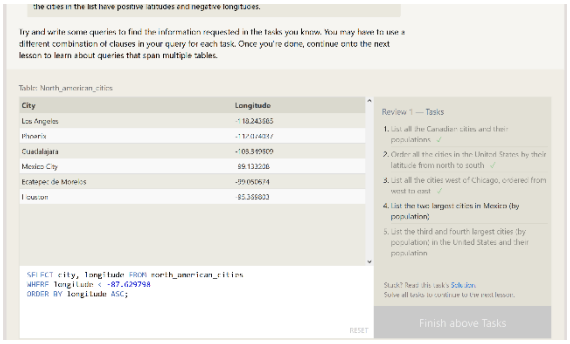


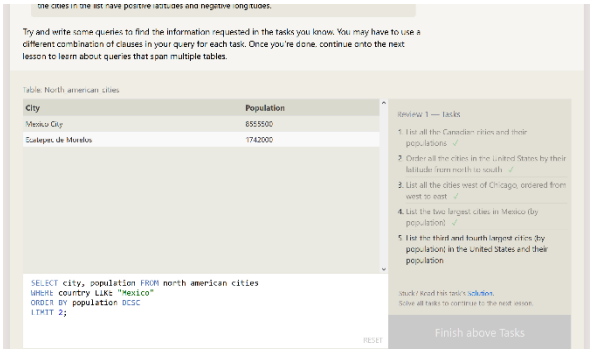


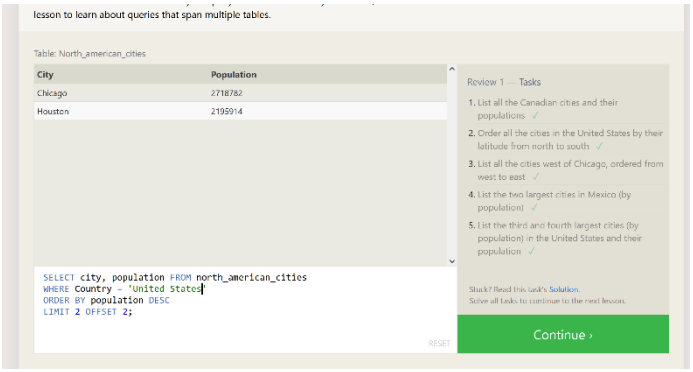
## SQL Review: Simple SELECT Queries



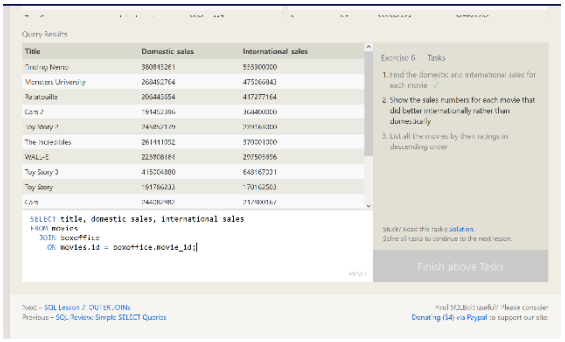


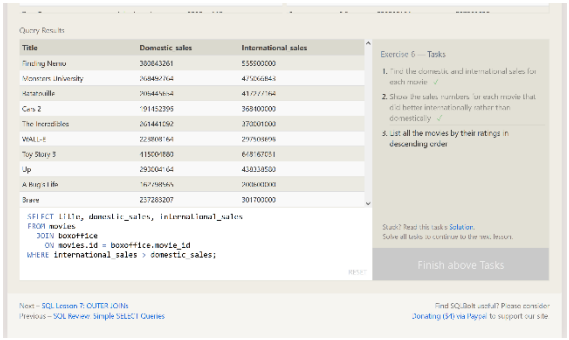


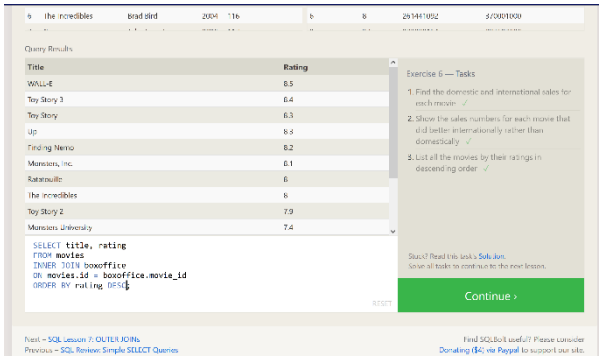




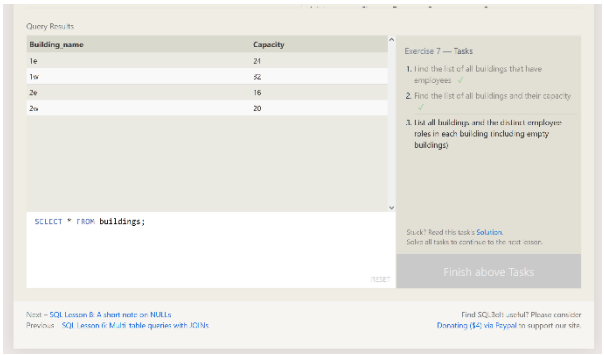
# SQL Lesson 6: Multi-table queries with JOINs

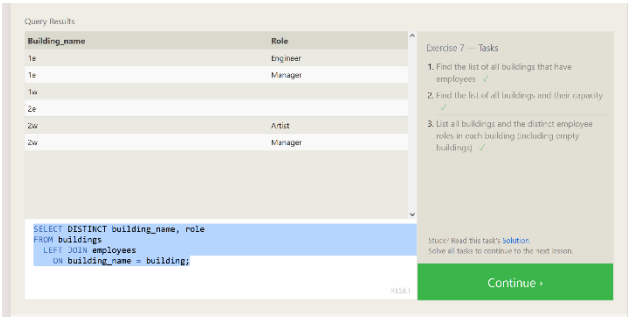




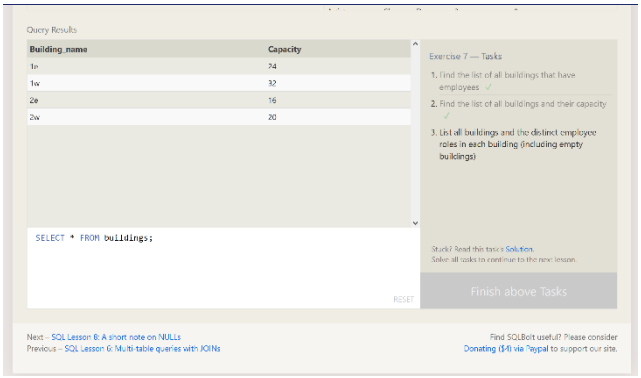


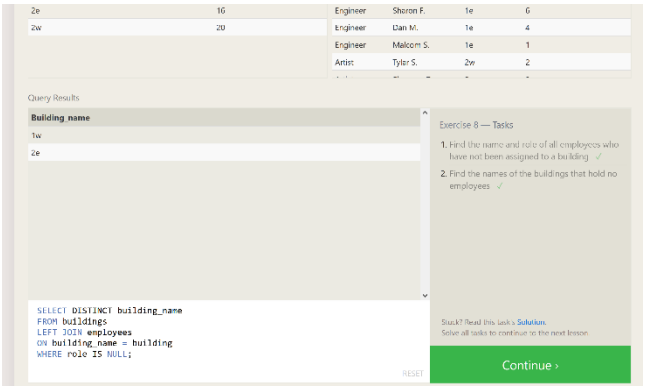
# SQL Lesson 7: OUTER JOINs

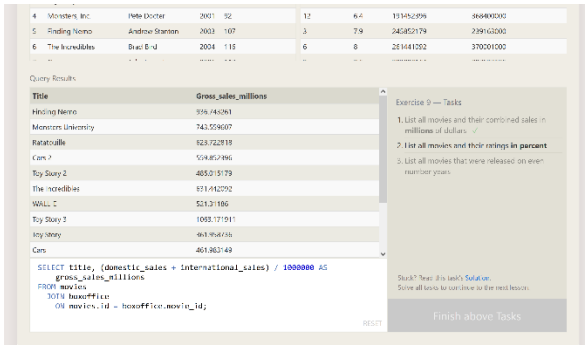



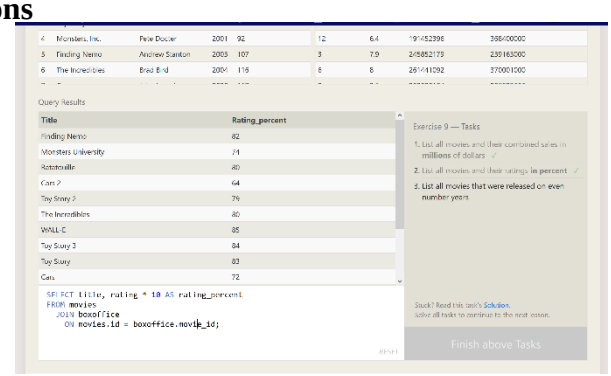

# SQL Lesson 8: A short note on NULLs

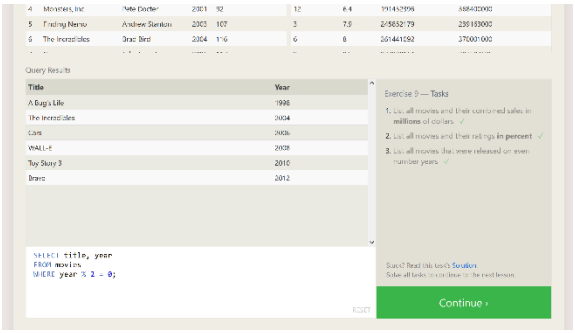


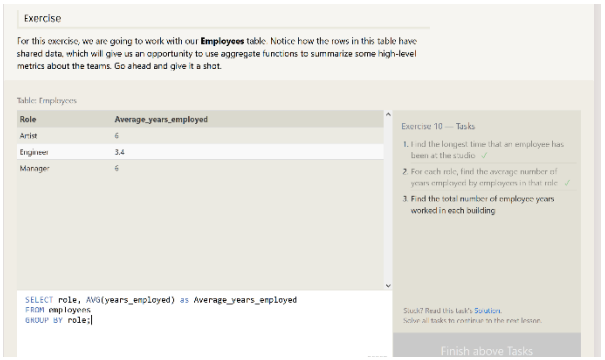

# SQL Lesson 9: Queries with expressions

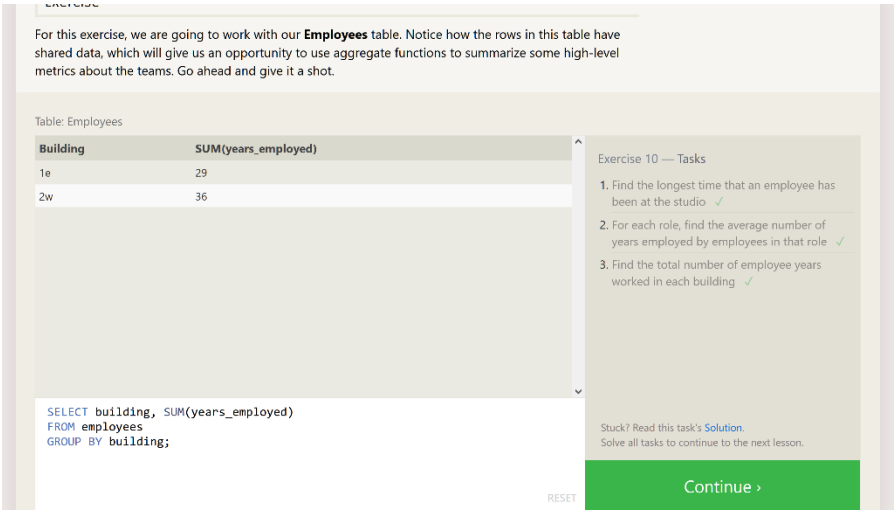


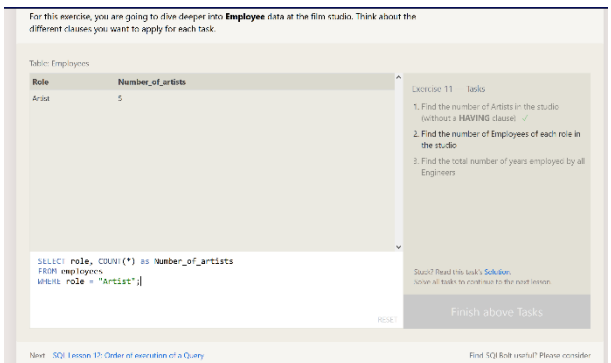


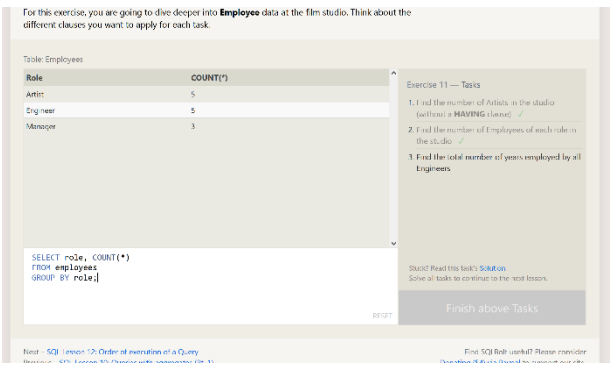

## SQL Lesson 10: Queries with aggregates (Pt. 1)

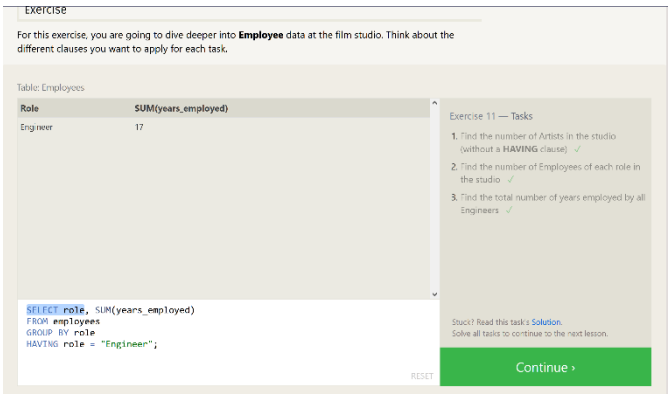



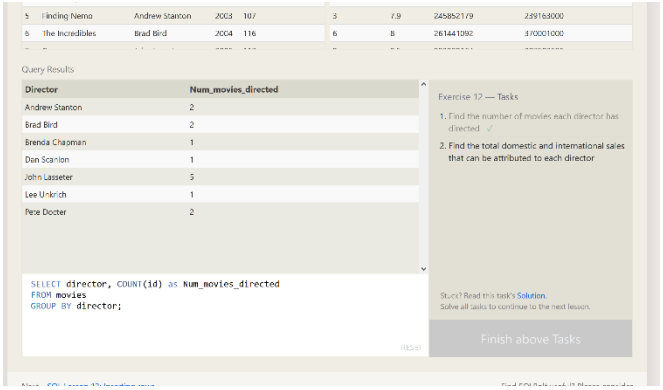

## SQL Lesson 11: Queries with aggregates (Pt. 2)

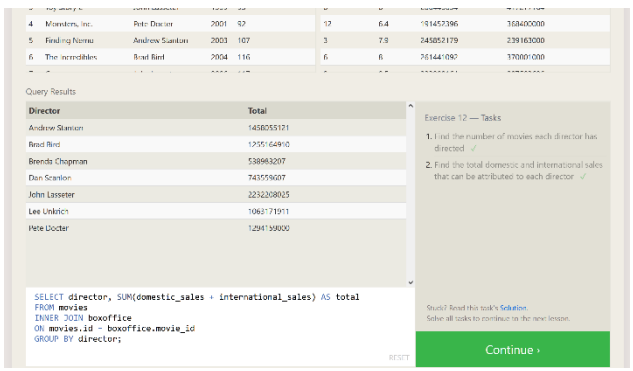


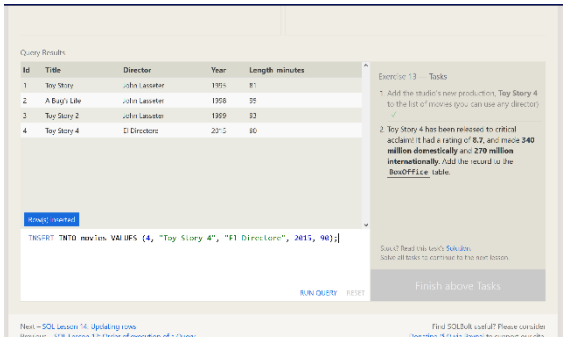


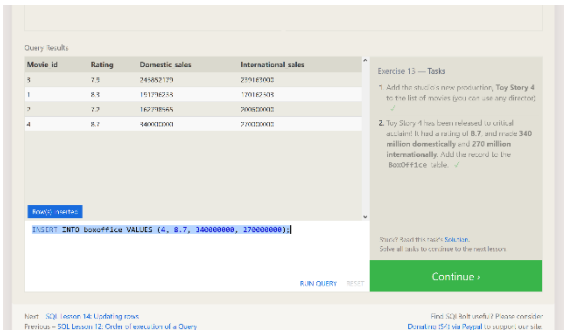

## SQL Lesson 12: Order of execution of a Query



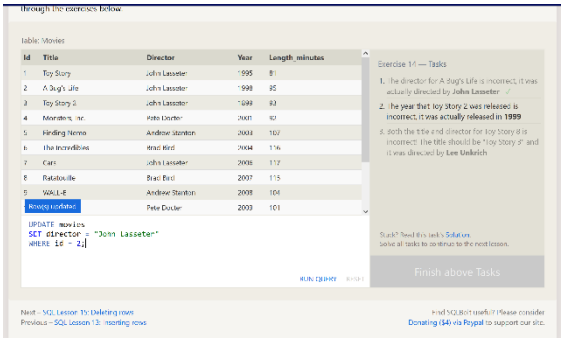



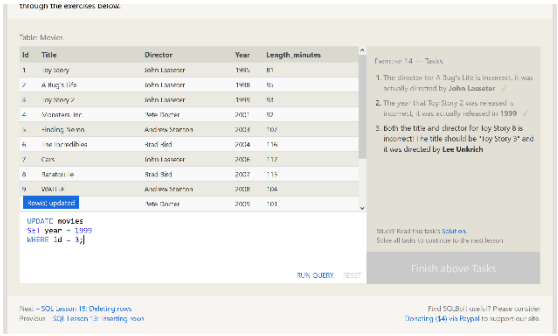

## SQL Lesson 13: Inserting rows

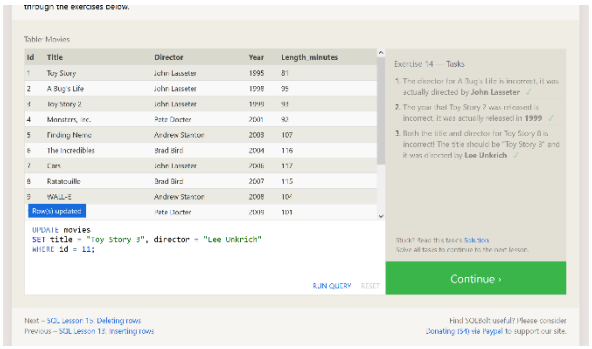


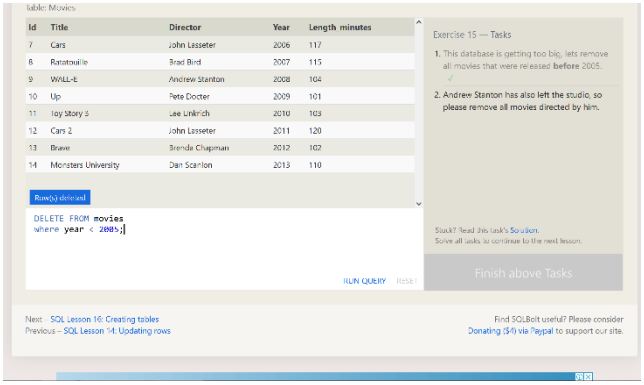

## SQL Lesson 14: Updating rows

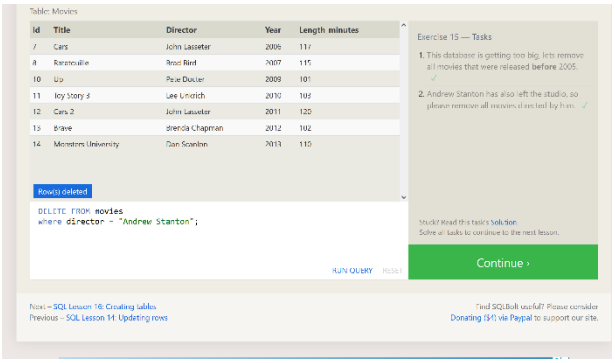


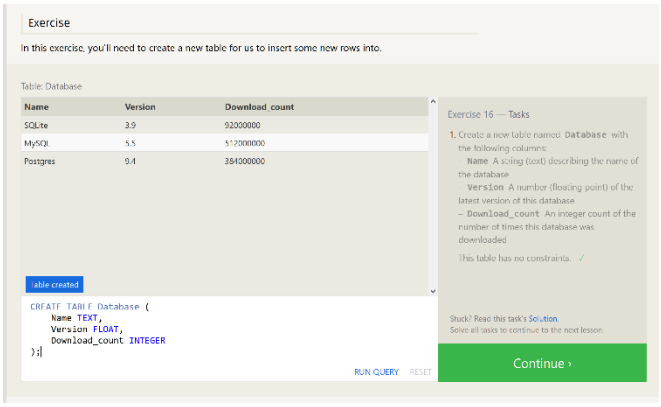


## SQL Lesson 15: Deleting rows

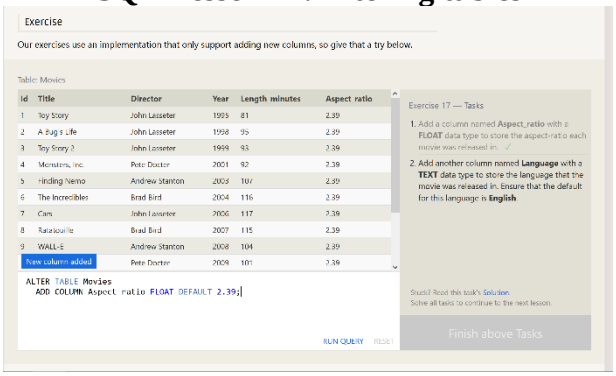


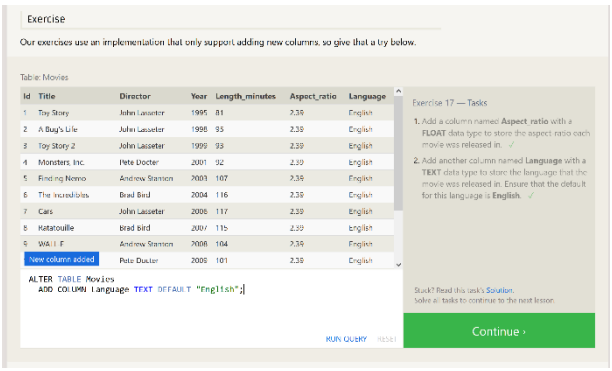


## SQL Lesson 16: Creating tables

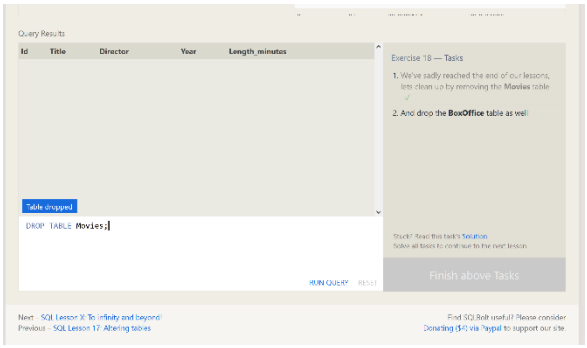


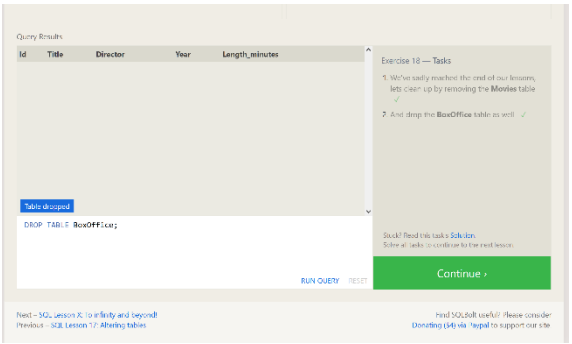
## SQL Lesson 17: Altering tables

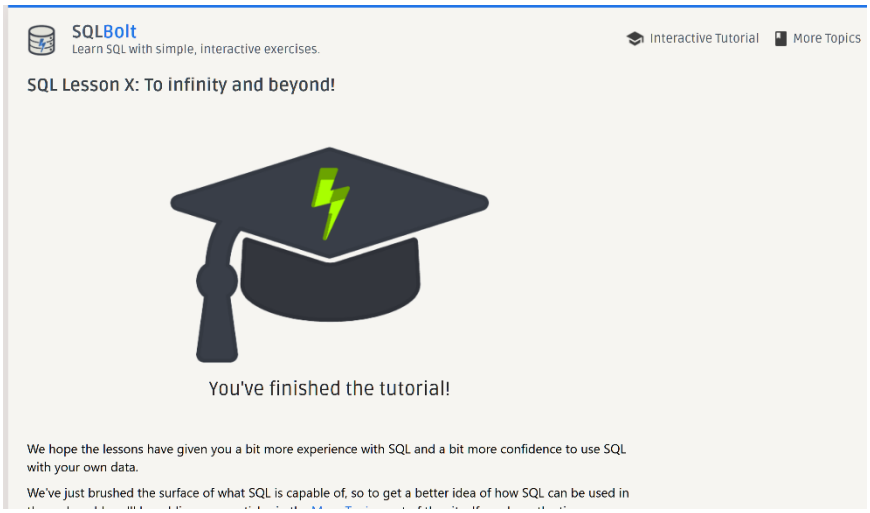




**SQL Lesson 18: Dropping tables**




# EJERCICIO # 2

En esta segunda parte de la tarea se le solicitará completar un segundo tutorial que gira alrededor del análisis de un caso de uso más real, con el fin de ver aplicados los conceptos que se van aprendiendo:
https://selectstarsql.com/
Debe completar los cinco capítulos del tutorial, registrando pantallazos de cada uno de los ejercicios realizados dentro de las herramientas del sitio web.

# THE BEAZLEY CASE

## A First SQL Query

that an eye for an eye does not constitute justice. Our task is to retrieve his statement from the database.

### A First SQL Query

**Run this query to find the first 3 rows of the 'executions' table.**
Viewing a few rows is a good way to find out the columns of a table. Try to remember the column names for later use.

```
1 SELECT * FROM executions LIMIT 3
```

Run ⇩ Reset

| | | | | | | |
|---|---|---|---|---|---|---|
| Christopher Anthony | Young | 553 | 34 | 2018-07-17 | Bexar | I want to make sure the Patel family knows I love them like they love me. Make sure the kids in the world know I'm being executed and those kids I've been mentoring keep this fight going. I'm good Warden. |
| Danny Paul | Bible | 552 | 66 | 2018-06-27 | Harris | null |
| Juan Edward | Castillo | 551 | 37 | 2018-05-16 | Bexar | To everyone that has been there for me you know who you are. Love y'all. See y'all on the other side.That's it. |

The SQL query may look like an ordinary sentence, but you should view it as three Lego blocks: `SELECT *` `FROM executions` `LIMIT 3`. As with Lego, each block has a fixed format and the different blocks have to fit together in particular ways.

## The SELECT Block

### The SELECT Block

The `SELECT` block specifies which columns you want to output. Its format is `SELECT <column>, <column>, ...`. Each column must be separated by a comma, but the space following the comma is optional. The star (ie. `*`) is a special character that signifies we want all the columns in the table.

**In the code editor below, revise the query to select the last_statement column in addition to the existing columns.**
Once you're done, you can hit Shift+Enter to run the query.

```
1 SELECT first_name, last_name, last_statement
2 FROM executions
3 LIMIT 3
```

Run ⇩ Show Solution Reset

Correct

| first_name | last_name | last_statement |
|---|---|---|
| Christopher Anthony | Young | I want to make sure the Patel family knows I love them like they love me. Make sure the kids in the world know I'm being executed and those kids I've been mentoring keep this fight going. I'm good Warden. |
| Danny Paul | Bible | null |
| Juan Edward | Castillo | To everyone that has been there for me you know who you are. Love y'all. See y'all on the other side.That's it. |

> **SQL Comments**
> Notice that clicking "Show Solution" displays the solution in the editor preceded by `/*`. The contents between `/*` and `*/` are taken as

## The FROM Block

### The FROM Block

The `FROM` block specifies which table we're querying from. Its format is `FROM <table>`. It always comes after the `SELECT` block.

**Run the given query and observe the error it produces. Fix the query.**
Make it a habit to examine error messages when something goes wrong. Avoid debugging by gut feel or trial and error.

```
1 SELECT first_name FROM executions LIMIT 3
```

Run ⇩ Show Solution Reset

Correct

| first_name |
|---|
| Christopher Anthony |
| Danny Paul |
| Juan Edward |

In the next example, observe that we don't need the `FROM` block if we're not using anything from a table.

**Modify the query to divide 50 and 51 by 2.**
SQL supports all the usual arithmetic operations.

```
1 SELECT 50 + 2, 51 * 2
```

In the next example, observe that we don't need the `FROM` block if we're not using anything from a table.

**Modify the query to divide 50 and 51 by 2.**
SQL supports all the usual arithmetic operations.

```
1 SELECT 50 / 2, 51 / 2
```

Run ⇩ Show Solution Reset

Correct

| 50 / 2 | 51 / 2 |
|---|---|
| 25 | 25 |

Isn't it strange that `51 / 2` gives `25` rather than `25.5`? This is because SQL is doing integer division. To do decimal division, at least one of the operands must be a decimal, for instance `51.0 / 2`. A common trick is to multiply one number by `1.0` to convert it into a decimal. This will come in useful in the later chapters.

> **Capitalization**
> Even though we've capitalized `SELECT`, `FROM` and `LIMIT`, SQL commands are not case-sensitive. You can see that the code editor recognizes them and formats them as a command no matter the capitalization. Nevertheless, I recommend capitalizing them to differentiate them from column names, table names and variables.
>
> Column names, table names and variables are also not case-sensitive in this version of SQL, though they are case-sensitive in many other

# The WHERE Block

goes after the `FROM` block. Here, a clause refers to a Boolean statement that the computer can evaluate to be true or false like `ex_number = 145`. You can imagine that the computer will go through each row in the table checking if the clause is true, and if so, return the row.

**Find the first and last names and ages (ex_age) of inmates 25 or younger at time of execution.**
Because the average time inmates spend on death row prior to execution is 10.26 years, only 6 inmates this young have been executed in Texas since 1976.

```
SELECT first_name, last_name, ex_age FROM executions
WHERE ex_age <= 25;
```

Run ⇩ Show Solution Reset

Correct

| first_name | last_name | ex_age |
|---|---|---|
| Toronto | Patterson | 24 |
| T.J. | Jones | 25 |
| Napoleon | Beazley | 25 |
| Richard | Andrade | 25 |
| Jay | Pinkerton | 24 |
| Jesse | De La Rosa | 24 |

It's clear how we can use arithmetic operators like `<` and `<=` to build clauses. There are also a collection of string operators to

> named 'where' and a column named 'from'. (Who would be so cruel as to do this?!) You would have to do `SELECT `from` FROM `where` WHERE ...`. This is another example why capitalization of SQL commands helps.

As you've seen in the previous exercise, complex clauses can be made out of simple ones using Boolean operators like `NOT`, `AND` and `OR`. SQL gives most precedence to `NOT` and then `AND` and finally `OR`. But if, like me, you're too lazy to remember the order of precedence, you can use parenthesis to clarify the order you want.

**Insert a pair of parenthesis so that this statement returns 0.**
Here we're relying on the fact that 1 means true and 0 means false.

```
SELECT 0 AND (0 OR 1)
```

Run ⇩ Show Solution Reset

Correct

| 0 AND (0 OR 1) |
|---|
| 0 |

Let's take a quick quiz to cement your understanding.

**Select the `WHERE` blocks with valid clauses.**
These are tricky. Even if you've guessed correctly, read the explanations to understand the reasoning.

WHERE 0

Check Answers Show Explanations

All correct!

Now you have the tools you need to complete our project.

**Find Napoleon Beazley's last statement.**

```
SELECT last_statement FROM executions
WHERE first_name LIKE '%Napoleon%'
AND last_name LIKE '%Beazley%';

```

Run ⇩ Show Solution Reset

Correct

| last_statement |
|---|
| The act I committed to put me here was not just heinous, it was senseless. But the person that committed that act is no longer here - I am. I'm not going to struggle physically against any restraints. I'm not going to shout, use profanity or make idle threats. Understand though that I'm not only upset, but I'm saddened by what is happening here tonight. I'm not only saddened, but disappointed that a system that is supposed to protect and uphold what is just and right can be so much like me when I made the same shameful mistake. If someone tried to dispose of everyone here for participating in this killing, I'd scream a resounding, 'No.' I'd tell them to give them all the gift that they would not give me...and that's to give them all a second chance. I'm sorry that I am here. I'm sorry that you're all here. I'm sorry that John Luttig died. And I'm sorry that it was something in me that caused all of this to happen to begin with. Tonight we tell the world that there are no second chances in the eyes of |

Isn't it amazing how profound and eloquent Beazley is? Recall that he was only 25 when he made the statement and had been in prison since he was 18.

## Recap

The most powerful of these is probably `LIKE`. It allows us to use wildcards such as `%` and _ to match various characters. For instance, `first_name LIKE '%roy'` will return true for rows with first names 'roy', 'Troy', and 'Deroy' but not 'royman'. The wildcard _ will match a single character so `first_name LIKE '_roy'` will only match 'Troy'.

**Modify the query to find the result for Raymond Landry.**
You might think this would be easy since we already know his first and last name. But datasets are rarely so clean. Use the LIKE operator so you don't have to know his name perfectly to find the row.

```
SELECT first_name, last_name, ex_number
FROM executions
WHERE first_name LIKE '%Raymond%'
  AND last_name LIKE '%Landry%'
```

Run ⇩ Show Solution Reset

Correct

| first_name | last_name | ex_number |
|---|---|---|
| Raymond | Landry, Sr. | 29 |

> **Quotes**
> In SQL, strings are denoted by single quotes. Backticks (ie `) can be used to denote column and table names. This is useful when the column or table name is the same as a SQL keyword and when they have a space in them. It's possible to have a database with a table

# CHAPTER # 2, POSSIBLE INNOCENCE

## The COUNT function

### The COUNT Function

`COUNT` is probably the most widely-used aggregate function. As the name suggests, it counts things! For instance, `COUNT(<column>)` returns the number of non-null rows in the column.

**Edit the query to find how many inmates provided last statements.**
We can use `COUNT` here because `NULL`s are used when there are no statements.

```
SELECT COUNT(last_statement) FROM executions
```

Run | Show Solution | Reset

Correct

| COUNT(last_statement) |
|---|
| 443 |

As you can tell, the `COUNT` function is intrinsically tied to the concept of `NULL`s. Let's make a small digression to learn about `NULL`s.

> **Nulls**
> In SQL, `NULL` is the value of an empty entry. This is different from the empty string `''` and the integer `0`, both of which are *not* considered `NULL`. To check if an entry is `NULL`, use `IS` and `IS NOT` instead of `=` and `!=`.

**Verify that 0 and the empty string are not considered NULL.**
Recall that this is a compound clause. Both of the two `IS NOT NULL` clauses have to be true for the query to return `true`.

```
SELECT (0 IS NOT NULL) AND ('' IS NOT NULL)
```

Run | Reset

| (0 IS NOT NULL) AND ('' IS NOT NULL) |
|---|
| 1 |

With this, we can find the denominator for our proportion:

**Find the total number of executions in the dataset.**
The idea here is to pick one of the columns that you're confident has no `NULL`s and count it.

```
SELECT COUNT(ex_number) AS total_executions FROM execut
```

Run | Show Solution | Reset

Correct

| total_executions |
|---|
| 553 |

### Variations on COUNT

So far so good. But what if we don't know which columns are `NULL`-free? Worse still, what if none of the columns are `NULL`-free?

## Variations on COUNT


### Variations on COUNT

So far so good. But what if we don't know which columns are `NULL`-free? Worse still, what if none of the columns are `NULL`-free? Surely there must still be a way to find the length of the table!


The solution is `COUNT(*)`. This is reminiscent of `SELECT *` where the `*` represents all columns. In practice `COUNT(*)` counts rows as long as *any one* of their columns is non-null. This helps us find table lengths because a table shouldn't have rows that are completely null.

**Verify that `COUNT(*)` gives the same result as before.**

```
SELECT COUNT(*) FROM executions
```

Run | Reset

| COUNT(*) |
|---|
| 553 |

Another common variation is to count a subset of the table. For instance, counting Harris county executions. We could run `SELECT COUNT(*) FROM executions WHERE county='Harris'` which filters down to a smaller dataset consisting of Harris executions

This is admittedly one of the clunkier parts of SQL. A common mistake is to miss out the `END` command and the `ELSE` condition which is a catchall in case all the prior clauses are false. Also recall from the previous chapter that clauses are expressions that can be evaluated to be true or false. This makes it important to think about the boolean value of whatever you stuff in there.

**This query counts the number of Harris and Bexar county executions. Replace `SUM`s with `COUNT`s and edit the `CASE WHEN` blocks so the query still works.**
Switching `SUM` for `COUNT` alone isn't enough because `COUNT` still counts the 0 since 0 is non-null.

```
SELECT
    SUM(CASE WHEN county='Harris' THEN 1
        ELSE 0 END) AS Harris,
    SUM(CASE WHEN county='Bexar' THEN 1
        ELSE 0 END) AS Bexar
FROM executions
```

Run | Show Solution | Reset

Correct

| Harris | Bexar |
|---|---|
| 128 | 46 |

```
    SUM(CASE WHEN county='Bexar' THEN 1
        ELSE 0 END) AS Bexar
FROM executions
```

Run Show Solution Reset

Correct

| Harris | Bexar |
|---|---|
| 128 | 46 |

## Practice

**Find how many inmates were over the age of 50 at execution time.**
This illustrates that the `WHERE` block filters before aggregation occurs.

```
T SUM(CASE WHEN ex_age > 50 THEN 1 ELSE 0 END) AS inmat
executions;
```

Run Show Solution Reset

Correct

| inmates |
|---|
| 68 |

**Find the number of inmates who have declined to give a last statement.**
For bonus points, try to do it in 3 ways:
1) With a `WHERE` block,
2) With a `COUNT` and `CASE WHEN` block,
3) With two `COUNT` functions.

Run Show Solution Reset

Correct

| inmates |
|---|
| 68 |

**Find the number of inmates who have declined to give a last statement.**
For bonus points, try to do it in 3 ways:
1) With a `WHERE` block,
2) With a `COUNT` and `CASE WHEN` block,
3) With two `COUNT` functions.

```
SELECT COUNT(CASE WHEN last_statement IS NULL THEN 1 ELS
FROM executions;
```

Run Show Solution Reset

| total |
|---|
| 553 |

It is worthwhile to step back and think about the different ways the computer handled these three queries. The `WHERE` version had it filter down to a small table first before aggregating while in the other two, it had to look through the full table. In the `COUNT` + `CASE WHEN` version, it only had to go through once, while the double `COUNT` version made it go through twice. So even though the output was identical, the performance was probably best in the first and worst in the third version.

had it filter down to a small table first before aggregating while in the other two, it had to look through the full table. In the `COUNT` + `CASE WHEN` version, it only had to go through once, while the double `COUNT` version made it go through twice. So even though the output was identical, the performance was probably best in the first and worst in the third version.

**Find the minimum, maximum and average age of inmates at the time of execution.**
Use the `MIN`, `MAX`, and `AVG` aggregate functions.

```
SELECT MIN(ex_age) AS minimun,
    MAX(ex_age) AS maximun,
    AVG(ex_age) AS average
FROM executions
```

Run Show Solution Reset

Correct

| minimun | maximun | average |
|---|---|---|
| 24 | 67 | 39.47016274864376 |

### Looking Up Documentation

This book was never intended to be a comprehensive reference for the SQL language. For that, you will have to look up other online resources. This is a skill in itself, and one that is worth mastering because you will be looking up documentation years after you've achieved familiarity with the language.

The good news is that with the mental models you will learn in this book, lookups should be quick and painless because you will just be checking details like whether the function is called `AVERAGE` or `AVG` instead of figuring out what approach to take.

For lookups, I often use W3 Schools, Stack Overflow and the official SQLite documentation.

**Find the average length (based on character count) of last statements in the dataset.**
This exercise illustrates that you can compose functions. Look up the documentation to figure out which function which returns the number of characters in a string.

```
SELECT AVG(LENGTH(last_statement)) AS average
FROM executions;
```

Run Show Solution Reset

Correct

| average |
|---|
| 537.492099322799 |

**List all the counties in the dataset without duplication.**
We can get unique entries by using `SELECT DISTINCT`. See documentation.

Run Show Solution Reset

Correct

| average |
|---|
| 537.492099322799 |

**List all the counties in the dataset without duplication.**
We can get unique entries by using `SELECT DISTINCT`. See documentation.

```
SELECT DISTINCT(county) FROM executions;
```

Run Show Solution Reset

Correct

| county |
|---|
| Bexar |
| Harris |
| Tarrant |
| Lubbock |
| Dallas |
| Hidalgo |

`SELECT DISTINCT` isn't really an aggregate function because it doesn't return a single number and because it operates on the output of the query rather than the underlying table. Nevertheless, I've included it here because it shares a common characteristic of operating on multiple rows.

## A Strange Query

Before we wrap up, let's take a look at this query:
`SELECT first_name, COUNT(*) FROM executions`.

Doesn't it look strange? If you have a good mental model of aggregations, it should! `COUNT(*)` is trying to return a single entry consisting the length of the execution table. `first_name` is trying to return one entry for each row. Should the computer return one or multiple rows? If it returns one, which `first_name` should it pick? If it returns multiple, is it supposed to replicate the `COUNT(*)` result across all the rows? The shapes of the output just don't match!

**Let's try it anyway and see what happens.**

```
SELECT first_name, COUNT(*) FROM executions
```

Run Reset

| first_name | COUNT(*) |
|---|---|
| Charlie | 553 |

In practice, databases try to return something sensible even though you pass in nonsense. In this case, our database picks the first name from the last entry in our table. Since our table is in reverse chronological order, the last entry is Charlie Brooks Jr., the first person executed since the Supreme Court lifted the ban on the death penalty. Different databases will handle this case

### Conclusion and Recap

Let's use what we've learned so far to complete our task:

**Find the proportion of inmates with claims of innocence in their last statements.**
To do decimal division, ensure that one of the numbers is a decimal by multiplying it by 1.0. Use `LIKE '%innocent%'` to find claims of innocence.

```
SELECT 1.0 * COUNT(CASE WHEN last_statement LIKE '%inno
            THEN 1 ELSE NULL END) / COUNT(*)
            AS inmates_proportion
            FROM executions;
```

Run Show Solution Reset

Correct

| inmates_proportion |
|---|
| 0.05605786618444846 |

This method of finding claims of innocence is admittedly rather inaccurate because innocence can be expressed in other terms like "not guilty". Nevertheless, I suspect it underestimates the real number, and is probably of the right order of magnitude. The question we are left with then, is whether we are willing to accept the possibility that up to 5% percent of people we execute

# CHAPTER # 3. LONG TAILS

## The GROUP BY block

### The GROUP BY Block

This is where the `GROUP BY` block comes in. It allows us to split up the dataset and apply aggregate functions within each group, resulting in one row per group. Its most basic form is `GROUP BY <column>, <column>, ...` and comes after the `WHERE` block.

**This query pulls the execution counts per county.**

```
SELECT
  county,
  COUNT(*) AS county_executions
FROM executions
GROUP BY county
```

Run Reset

| county | county_executions |
|---|---|
| Anderson | 4 |
| Aransas | 1 |
| Atascosa | 1 |
| Bailey | 1 |
| Bastrop | 1 |
| Bee | 2 |
| Bell | 3 |

If you recall A Strange Query, alarm bells would be going off in

saves us from rewriting long expressions, and allows us to clarify the purpose of the expression.

**This query counts the executions with and without last statements. Modify it to further break it down by county.**
The clause `last_statement IS NOT NULL` acts as an indicator variable where 1 means true and 0 means false.

```
SELECT
  last_statement IS NOT NULL AS has_last_statement,
  county,
  COUNT(*)
FROM executions
GROUP BY has_last_statement, county
```

Run | Show Solution | Reset

Correct

| has_last_statement | county | COUNT(*) |
|---|---|---|
| 0 | Bell | 1 |
| 0 | Bexar | 9 |
| 0 | Brazoria | 1 |
| 0 | Brazos | 2 |
| 0 | Chambers | 1 |
| 0 | Collin | 1 |

## The HAVING Block

# The HAVING Block

## The HAVING Block

This next exercise illustrates that filtering via the `WHERE` block happens before grouping and aggregation. This is reflected in the order of syntax since the `WHERE` block always precedes the `GROUP BY` block.

**Count the number of inmates aged 50 or older that were executed in each county.**
You should be able to do this using `CASE WHEN`, but try using the `WHERE` block here.

```
SELECT county, COUNT(*)
FROM executions
WHERE ex_age >= 50
GROUP BY county;

```

Run | Show Solution | Reset

Correct

| county | COUNT(*) |
|---|---|
| Anderson | 1 |
| Bexar | 2 |
| Caldwell | 1 |
| Cameron | 1 |
| Collin | 2 |
| Comal | 1 |

forward into the future and grab information from there. To solve this problem, we use `HAVING`.

**List the counties in which more than 2 inmates aged 50 or older have been executed.**
This builds on the previous exercise. We need an additional filter—one that uses the result of the aggregation. This means it cannot exist in the `WHERE` block because those filters are run before aggregation. Look up the `HAVING` block. You can think of it as a post-aggregation `WHERE` block.

```
SELECT county
FROM executions
WHERE ex_age >= 50
GROUP BY county
HAVING COUNT(*) > 2;

```

Run | Show Solution | Reset

Correct

| county |
|---|
| Dallas |
| Harris |
| Lubbock |
| Montgomery |
| Tarrant |

## Practice

## Practice

This quiz is designed to challenge your understanding. Read the explanations even if you get everything correct.

**Mark the statements that are true.**
This query finds the number of inmates from each county and 10 year age range.

```
SELECT
  county,
  ex_age/10 AS decade_age,
  COUNT(*)
FROM executions
GROUP BY county, decade_age
```

- [x] The query is valid (ie. won't throw an error when run).
  Were you thrown off by `ex_age/10`? Grouping by transformed columns is fine too.
- [x] The query would return more rows if we were to use `ex_age` instead of `ex_age/10`.
  Remember that `ex_age/10` does integer division which rounds all the ages. This produces fewer unique groups.
- [x] The output will have as many rows as there are unique combinations of counties and decade_ages in the dataset.
  This is correct.
- [ ] The output will have a group ('Bexar', 6) even though no Bexar county inmates were between 60 and 69 at execution time.

Remember that `ex_age/10` does integer division which rounds all the ages. This produces fewer unique groups.

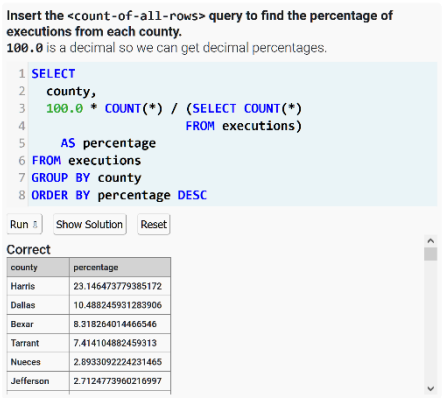
- [x] The output will have as many rows as there are unique combinations of counties and decade_ages in the dataset.
  This is correct.
- [ ] The output will have a group ('Bexar', 6) even though no Bexar county inmates were between 60 and 69 at execution time.
  The `GROUP BY` block finds all combinations *in the dataset* rather than all theoretically possible combinations.
- [ ] The output will have a different value of county for every row it returns.
  This would be true only if `county` were the only grouping column. Here, we can have many groups with the same county but different decade_ages.
- [ ] The output can have groups where the count is 0.
  This is similar to the ('Bexar', 6) question. If there are no rows with ('Bexar', 6), the group won't even show up.
- [x] The query would be valid even if we don't specify `county` in the `SELECT` block.
  The grouping columns don't necessarily have to be in the `SELECT` block. It would be valid, but not make much sense because we wouldn't know which counts are for which county.
- [ ] It is reasonable to add `last_name` to the `SELECT` block even without grouping by it.
  Even though it would be valid (in SQLite) for the reasons set forth in A Strange Query, it is poor form to have non-aggregate, non-grouping columns in the `SELECT` block.

Check Answers | Show Explanations


# Nested Queries

query and the outer one:


To reiterate, nesting is necessary here because in the `WHERE` clause, as the computer is inspecting a row to decide if its last statement is the right length, it can't look outside to figure out the maximum length across the entire dataset. We have to find the maximum length separately and feed it into the clause. Now let's apply the same concept to find the percentage of executions

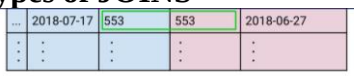



I've quietly slipped in an `ORDER BY` block. Its format is `ORDER BY <column>, <column>, ...` and it can be modified by appending `DESC` if you don't want the default ascending order.

# CHAPTER # 4, HIATUSES

# Thinking about JOINS

# Types of JOINS





Dates

# DATES

one another. But imagine you're the computer receiving a line like this. Do you return the number of days between the dates? Why not hours or seconds? To make things worse, SQLite doesn't actually have date or time types (unlike most other SQL dialects) so the `ex_date` and `last_ex_date` columns look like ordinary strings to you. You're effectively being asked to do `'hello' - 'world'`. What does that even mean?

Fortunately, SQLite contains a bunch of functions to tell the computer: "Hey, these strings that I'm passing you actually contain dates or times. Act on them as you would a date."

**Look up the documentation to fix the query so that it returns the number of days between the dates.**

```
SELECT JULIANDAY('1993-08-10') - JULIANDAY('1989-07-07')
AS day_difference
```

Run | Show Solution | Reset

Correct

| day_difference |
|---|
| 1495 |

## Self Joins

With what we learned about dates, we can correct our template

# Self JOINS

The next step is to build out the `previous` table.

**Write a query to produce the `previous` table.**
Remember to use aliases to form the column names `(ex_number, last_ex_date)`. Hint: Instead of shifting dates back, you could shift `ex_number` forward!

```
SELECT
  ex_number + 1 AS ex_number,
  ex_date AS last_ex_date
FROM executions
WHERE ex_number < 553
```

Run | Show Solution | Reset

Correct

| ex_number | last_ex_date |
|---|---|
| 553 | 2018-06-27 |
| 552 | 2018-05-16 |
| 551 | 2018-04-25 |
| 550 | 2018-03-27 |
| 549 | 2018-02-01 |
| 548 | 2018-01-30 |

Now we can nest this query into our template above:

**Nest the query which generates the `previous` table into the template.**
Notice that we are using a table alias here, naming the result of the nested query 'previous'.

Note that we still need to give one copy an alias to ensure that we can refer to it unambiguously.

```
SELECT
  previous.ex_date AS start,
  executions.ex_date AS end,
  JULIANDAY(executions.ex_date) - JULIANDAY(previous.ex
    AS day_difference
FROM executions
JOIN executions previous
  ON executions.ex_number = previous.ex_number + 1
ORDER BY day_difference DESC
LIMIT 10
```

Run | Show Solution | Reset

Correct

| start | end | day_difference |
|---|---|---|
| 1982-12-07 | 1984-03-14 | 463 |
| 1988-01-07 | 1988-11-03 | 301 |
| 2007-09-25 | 2008-06-11 | 260 |
| 1990-07-18 | 1991-02-26 | 223 |
| 1984-03-31 | 1984-10-30 | 213 |
| 1996-02-27 | 1996-09-18 | 204 |

We can now use the precise dates of the hiatuses to research what happened over each period. In the years immediately after the ban on capital punishment was lifted, there were long periods without executions due to the low number of death

**Nest the query which generates the previous table into the template.**
Notice that we are using a table alias here, naming the result of the nested query 'previous'.

```
SELECT
  last_ex_date AS start,
  ex_date AS end,
  JULIANDAY(ex_date) - JULIANDAY(last_ex_date)
    AS day_difference
FROM executions
JOIN (SELECT
        ex_number + 1 AS ex_number,
        ex_date AS last_ex_date
      FROM executions
      WHERE ex_number < 553) previous
  ON executions.ex_number = previous.ex_number
ORDER BY day_difference DESC
LIMIT 10
```

Run | Show Solution | Reset

Correct

| start | end | day_difference |
|---|---|---|
| 1982-12-07 | 1984-03-14 | 463 |
| 1988-01-07 | 1988-11-03 | 301 |
| 2007-09-25 | 2008-06-11 | 260 |
| 1990-07-18 | 1991-02-26 | 223 |
| 1984-03-31 | 1984-10-30 | 213 |
| 1996-02-27 | 1996-09-18 | 204 |

`previous` is derived from `executions`, so we're effectively joining

# CHAPTER # 5. CLOSING REMARKS

15894

**Find the most networked senator. That is, the one with the most mutual cosponsorships.**

A mutual cosponsorship refers to two senators who have each cosponsored a bill sponsored by the other. Even if a pair of senators have cooperated on many bills, the relationship still counts as one.

```
WITH mutuals AS (
  SELECT DISTINCT
    c1.sponsor_name AS senator,
    c2.sponsor_name AS senator2
  FROM cosponsors c1
  JOIN cosponsors c2
    ON c1.sponsor_name = c2.cosponsor_name
    AND c2.sponsor_name = c1.cosponsor_name
)

SELECT senator, COUNT(*) AS mutual_count
FROM mutuals
GROUP BY senator
ORDER BY mutual_count DESC
LIMIT 1
```

Run ⇩ Show Solution Reset

Correct

| senator | mutual_count |
|---|---|
| Brown, Sherrod | 65 |

**Now find the most networked senator from each state.**

```
WITH mutual_counts AS (
  SELECT
    senator, state, COUNT(*) AS mutual_count
  FROM (
    SELECT DISTINCT
      c1.sponsor_name AS senator,
      c1.sponsor_state AS state,
      c2.sponsor_name AS senator2
    FROM cosponsors c1
    JOIN cosponsors c2
      ON c1.sponsor_name = c2.cosponsor_name
      AND c2.sponsor_name = c1.cosponsor_name
    )
  GROUP BY senator, state
),

state_max AS (
  SELECT
    state,
    MAX(mutual_count) AS max_mutual_count
  FROM mutual_counts
  GROUP BY state
)

SELECT
  mutual_counts.state,
  mutual_counts.senator,
  mutual_counts.mutual_count
FROM mutual_counts
JOIN state_max
  ON mutual_counts.state = state_max.state
  AND mutual_counts.mutual_count = state_max.max_mutual_
```

Run ⇩ Show Solution Reset

Correct

| state | senator | mutual_count |
|---|---|---|
| TN | Alexander, Lamar | 43 |
| WI | Baldwin, Tammy | 44 |
| CO | Bennet, Michael F. | 31 |
| CT | Blumenthal, Richard | 55 |
| MO | Blunt, Roy | 57 |
| AR | Boozman, John | 54 |

**Find the senators who cosponsored but didn't sponsor bills.**


| | | |
|---|---|---|
| CO | Bennet, Michael F. | 31 |
| CT | Blumenthal, Richard | 55 |
| MO | Blunt, Roy | 57 |
| AR | Boozman, John | 54 |

**Find the senators who cosponsored but didn't sponsor bills.**


```
SELECT DISTINCT c1.cosponsor_name
FROM cosponsors c1
LEFT JOIN cosponsors c2
 ON c1.cosponsor_name = c2.sponsor_name
 -- This join identifies cosponsors
 -- who have sponsored bills
WHERE c2.sponsor_name IS NULL
-- LEFT JOIN + NULL is a standard trick for excluding
-- rows. It's more efficient than WHERE ... NOT IN.

```

Run ⇩ Show Solution Reset

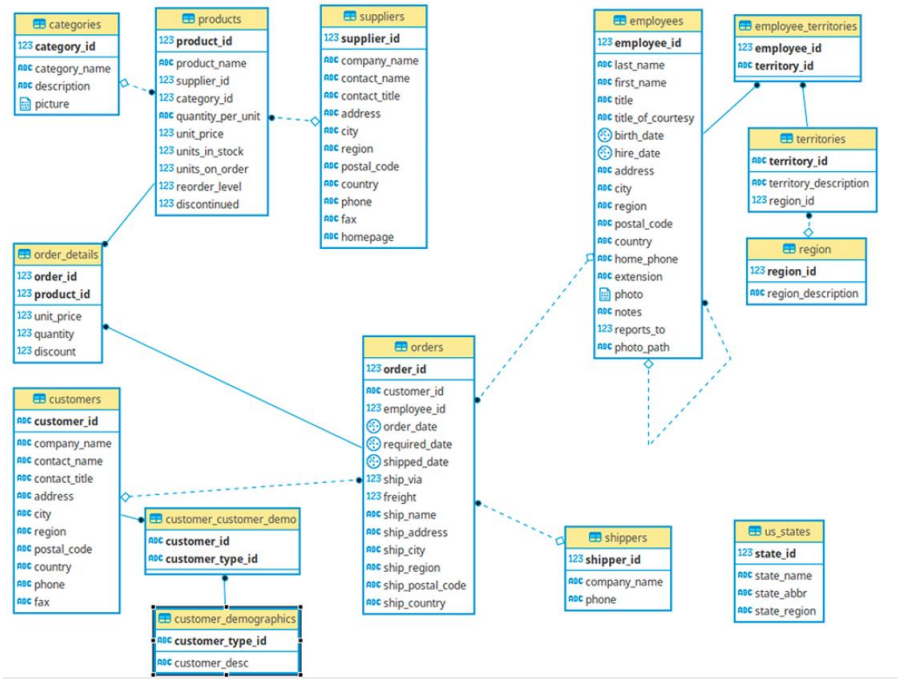
Correct

| cosponsor_name |
|---|
| Durbin, Richard |
| Murphy, Christopher S. |

Previous: Execution Hiatuses

# Ejercicio 3: Consultas en SQL (40 puntos)

Se le ha compartido el archivo **tarea1.sql** que contiene los *queries* necesarios para levantar la base de datos de una tienda distribuidora de productos alimenticios con sus ventas y proveedores de productos. La base de datos tiene una estructura descrita por el siguiente diagrama:




En esta base de datos se registran órdenes de venta (*orders*) en donde una orden de venta puede incluir múltiples productos (registrados por medio de la tabla *order_details*) y una orden está asociada a un solo cliente (*customers*) y vendida por un solo vendedor (*employees*).

1. Liste los productos (id, nombre y precio unitario) cuyo precio unitario está entre $15.00 y $20.00.

```
3912
3913  SELECT product_id, product_name, unit_price
3914  FROM products
3915  WHERE unit_price >= 15 AND unit_price <= 20;
```

Data Output | Messages | Notifications

| | product_id [PK] smallint | product_name character varying (40) | unit_price real |
|---|---|---|---|
| 1 | 1 | Chai | 18 |
| 2 | 2 | Chang | 19 |
| 3 | 16 | Pavlova | 17.45 |
| 4 | 35 | Steeleye Stout | 18 |
| 5 | 36 | Inlagd Sill | 19 |
| 6 | 39 | Chartreuse verte | 18 |
| 7 | 40 | Boston Crab Meat | 18.4 |
| 8 | 44 | Gula Malacca | 19.45 |
| 9 | 49 | Maxilaku | 20 |
| 10 | 50 | Valkoinen suklaa | 16.25 |

2. ¿Cuáles son los productos descontinuados? Para cada producto muestre su ID, nombre del producto y nombre de su categoría?

```
3917 SELECT product_id, product_name, category_name
3918 FROM products
3919 INNER JOIN categories
3920 ON products.category_id = categories.category_id
3921 WHERE discontinued = 1;
3922
```

Data Output | Messages | Notifications

| | product_id smallint | product_name character varying (40) | category_name character varying (15) |
|---|---|---|---|
| 1 | 1 | Chai | Beverages |
| 2 | 2 | Chang | Beverages |
| 3 | 5 | Chef Anton's Gumbo Mix | Condiments |
| 4 | 9 | Mishi Kobe Niku | Meat/Poultry |
| 5 | 17 | Alice Mutton | Meat/Poultry |
| 6 | 24 | Guaraná Fantástica | Beverages |
| 7 | 28 | Rössle Sauerkraut | Produce |
| 8 | 29 | Thüringer Rostbratwurst | Meat/Poultry |
| 9 | 42 | Singaporean Hokkien Fried Mee | Grains/Cereals |
| 10 | 53 | Perth Pasties | Meat/Poultry |

Total rows: 10 of 10 Query complete 00:00:00.075

3. Liste los productos (id, nombre y precio unitario) cuyo precio está por encima del promedio de precios

```
3924 SELECT product_id, product_name, unit_price
3925 FROM products
3926 WHERE unit_price > (
3927     SELECT AVG(unit_price) FROM products);
3928
```

Data Output | Messages | Notifications

| | product_id [PK] smallint | product_name character varying (40) | unit_price real |
|---|---|---|---|
| 1 | 7 | Uncle Bob's Organic Dried Pears | 30 |
| 2 | 8 | Northwoods Cranberry Sauce | 40 |
| 3 | 9 | Mishi Kobe Niku | 97 |
| 4 | 10 | Ikura | 31 |
| 5 | 12 | Queso Manchego La Pastora | 38 |
| 6 | 17 | Alice Mutton | 39 |
| 7 | 18 | Carnarvon Tigers | 62.5 |
| 8 | 20 | Sir Rodney's Marmalade | 81 |
| 9 | 26 | Gumbär Gummibärchen | 31.23 |
| 10 | 27 | Schoggi Schokolade | 43.9 |

Total rows: 25 of 25 Query complete 00:00:00.080

4. Genere un listado de las órdenes que aún no han sido entregadas (*shipped*), incluyendo el monto total de la orden (considerando descuentos)

```
3929 SELECT orders.order_id, (order_details.unit_price - order_details.unit_price * order_details.discount) AS total
3930 FROM orders
3931 INNER JOIN order_details ON orders.order_id = order_details.order_id
3932 WHERE orders.shipped_date IS NULL;
```

Data Output | Messages | Notifications

| | order_id [PK] smallint | total real |
|---|---|---|
| 1 | 11008 | 43.32 |
| 2 | 11008 | 13.3 |
| 3 | 11008 | 21.5 |
| 4 | 11019 | 12 |
| 5 | 11019 | 20 |
| 6 | 11039 | 45.6 |
| 7 | 11039 | 18 |
| 8 | 11039 | 20 |
| 9 | 11039 | 19.5 |
| 10 | 11040 | 10 |

Total rows: 73 of 73 Query complete 00:00:00.115 Ln 3931, C

5. ¿Cuál es el rango de precios de productos que maneja la tienda? Elabore un query que muestre el producto más caro y el mas barato manejado en el catálogo de productos de la tienda, considerando solo los productos que no estén descontinuados.

```
3936  SELECT product_name, unit_price
3937  FROM products
3938  WHERE unit_price = (SELECT MAX(unit_price) FROM products)
3939  OR unit_price = (SELECT MIN(unit_price) FROM products)
3940  AND discontinued != 1;
```

Data Output | Messages | Notifications

| | product_name character varying (40) | unit_price real |
|---|---|---|
| 1 | Geitost | 2.5 |
| 2 | Côte de Blaye | 263.5 |

6. Encuentre el cliente que hay registrado el mayor número de órdenes, mostrando su nombre y la cantidad de órdenes registradas

```
3947  SELECT  orders.order_id, orders.customer_id, company_name, quantity FROM orders
3948  INNER JOIN customers
3949  ON orders.customer_id = customers.customer_id
3950  INNER JOIN order_details
3951  ON orders.order_id = order_details.order_id
3952  WHERE quantity = (SELECT MAX(quantity) FROM order_details);
3953
```

Data Output | Messages | Notifications

| | order_id smallint | customer_id character | company_name character varying (40) | quantity smallint |
|---|---|---|---|---|
| 1 | 10764 | ERNSH | Ernst Handel | 130 |
| 2 | 11072 | ERNSH | Ernst Handel | 130 |

7. Liste los cinco productos que han generado el mayor volumen de dinero en ventas (registradas por medio órdenes). Para cada uno liste su el nombre del producto y el volumen monetario generado en órdenes.

```
3959  SELECT products.product_name, SUM(order_details.quantity * order_details.unit_price) AS total
3960  FROM order_details
3961  JOIN products
3962  ON products.product_id = order_details.product_id
3963  GROUP BY products.product_name
3964  ORDER BY total DESC
3965  LIMIT 5
```

Data Output | Messages | Notifications

| | product_name character varying (40) | total double precision |
|---|---|---|
| 1 | Côte de Blaye | 149984.20082092285 |
| 2 | Thüringer Rostbratwurst | 87736.40051269531 |
| 3 | Raclette Courdavault | 76296 |
| 4 | Camembert Pierrot | 50286.00037384033 |
| 5 | Tarte au sucre | 49827.89999771118 |

# Ejercicio 4: Cálculo de covarianza (20 puntos)

En este último ejercicio utilizaremos la base de datos **covid** que se encuentra disponible en CANVAS, con el objetivo de responder a la pregunta *¿cómo se relacionan el tabaco con respecto de la hipertensión en dicha base de datos?* Esto quiere decir que si un paciente tiene hipertensión, si esto está íntimamente relacionado con que fuma.

Una forma de estudiar la variación conjunta de dos variables es utilizar la *covariancia muestral* (si lo necesita, puede refrescar dicho concepto aquí). Mientras más alto sea el valor de la covarianza muestral entre dos variables, mayor será la tendencia de estas a comportante de la misma forma. Por el contrario, una covarianza negativa indica que las variables tienden a estar inversamente relacionadas.
Una forma de computar la covarianza entre dos variables $X$ y $Y$ es la siguiente:

4.1 Escriba un query para computar el término $1/(n-1)$ de la tabla de covid, en donde $n$ representa la cantidad de casos de covid.

```
3  SELECT 1.0 * COUNT(*) / (COUNT(*) - 1) AS result
4  FROM covid;
```

Data Output | Messages | Notifications

| | result<br>numeric |
|---|---|
| 1 | 1.00000095367613539912 |