Structured Query Language: Basics!

The world is a zoo, you've seen the news

Ted Tran ttran@student.42.us.org
42 Ghost Bum pedago@42.fr

Summary: This project is an introduction to **SQL** *using material from SQLZoo.*

Contents

Chapter I

Foreword

You know, at the end of the day, if you're stuck, you can easily get the vast majority of answers using google. SQLBolt and SQLZoo are probably the most popular resources to learn SQL these days.

But have no fear!

I wrote a `TON` of extra queries for you all to complete and have fun doing. And by fun, I mean as fun as falling into a gorilla enclosure at a zoo. Prepare for a world of `hurt` due to one man's good intentions.

Good luck!

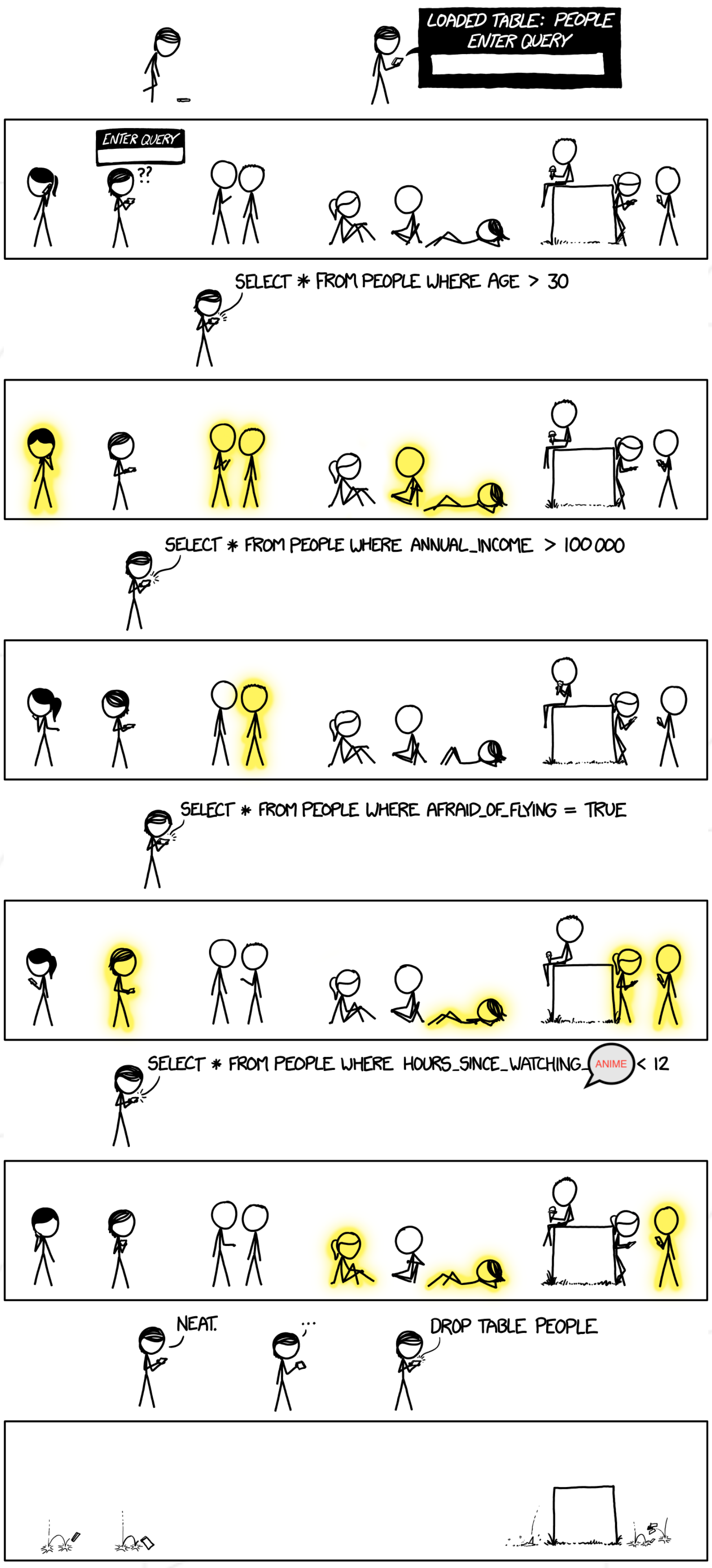
If you need help, you won't get it until you go onto the HackHighSchool slack and @(any mentor who's not in charge of the SQL curriculum) 10 times and show me proof.

And remember– please try solving the problem on your own using your massive brain, Google, or asking your peers before coming to me.

I'm sure you're wondering, "Why is this foreword so long?". I'll tell you why, the real truth. It's because the xkcd comic on the page after this one is so long, so I need to put filler text in to fill this page so it doesn't look empty.

You know what? I think you're being ungrateful. Just because of that, I'm going to write the rest of the learning material in French then google translate it into Vietnamese, Spanish, Japanese, and German before English to get the real authenticity of a 42 PDF.

But have no fear, High School students! -> Do not worry, student!


LOADED TABLE: PEOPLE
ENTER QUERY
ENTER QUERY
??
SELECT * FROM PEOPLE WHERE AGE > 30
SELECT * FROM PEOPLE WHERE ANNUAL_INCOME > 100 000
SELECT * FROM PEOPLE WHERE AFRAID_OF_FLYING = TRUE
SELECT * FROM PEOPLE WHERE HOURS_SINCE_WATCHING_ANIME < 12
NEAT.
...
DROP TABLE PEOPLE

Chapter II

Introduction

What the hell am I going to be doing?

- First, you're going to be running a ton of docker commands so that you can run various things on these 42 lab computers
- Next, you're going to be learning all the SQL fundamentals!
- It'll be learning through experience, so be prepared to write a ridiculous amount of queries.
- Once you have the fundamentals down, it'll be time for you to write a wrapper so users can interact with the database without writing any SQL queries(since you wrote em' already!)

Testing is provided for you, so you only need to run a single command to check your answers! Go for the green, buddy.

Chapter III

Goals

- Learn best practices for writing SQL queries
- Learn SQL fundamentals to query the database for required information
- Create a wrapper using a programming language of your choice, so that a person can obtain information from the database without having to write SQL queries themselves.

Your goal is to pass every single test for a beautiful 100 percent GREEN. Then after that, use what you've learned at H2S to create an interface for users to get what they want from the DB without writing SQL themselves.

Chapter IV

General instructions

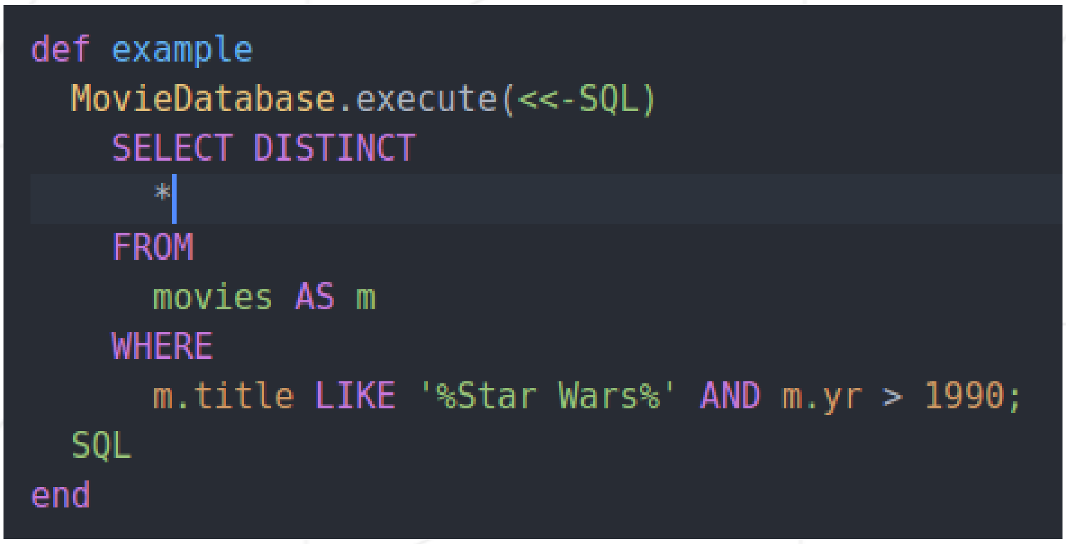
- This project will be corrected by humans only. You're allowed to organize and name your files as you see fit, but you must follow the following rules.
- The first rule of SQL Club is: You do not talk about SQL Club.
- The second rule of SQL Club is: You do not talk about SQL Club.
- I was going to copy the third rule from the 42 PDF, "You are never allowed to submit code you did not write yourself" but that'd make me a hypocrite. Go wild!
- Within the mandatory part, you are only allowed to use SQL.
- You can ask your questions on slack and random people in your nearby vicinity that appear to be older than you. Of course, with the exception being me– don't ask me any questions. Everybody at 42 is a SQL master, and H2S mentors are required to have a PhD in Structured Query Language studies! So don't be afraid to come to them for guidance.
- You will be provided with instructions on how to setup your programming environment, create the DB and import data, a skeleton to code in, and a full testing suite to check your answers.

Chapter V

Day 00 : Beginning your SQL studies!

	Exercise
SQL SELECT/FROM/WHERE, operators, DISTINCT, expression aliases, *	
Turn-in directory :	
Files to turn in : `01_select.rb`, `02_select.rb`, `03_select.rb`, `03a_select.rb`	
Forbidden functions : `None`	
Notes : `Make sure to use docker cp to store your files on the host computer, then add it to a git repo after!`	

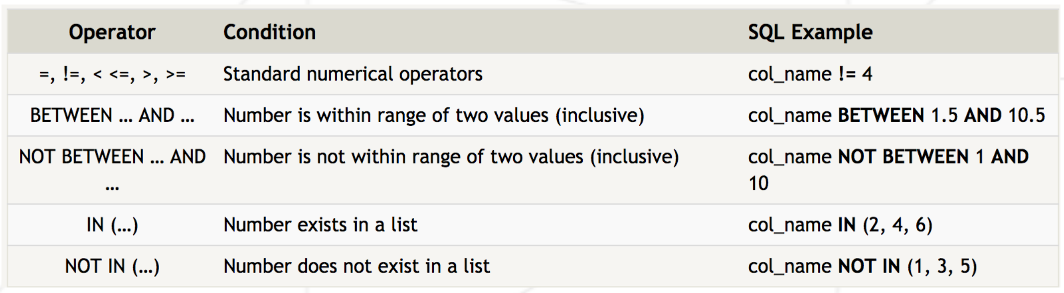
Look at this example!



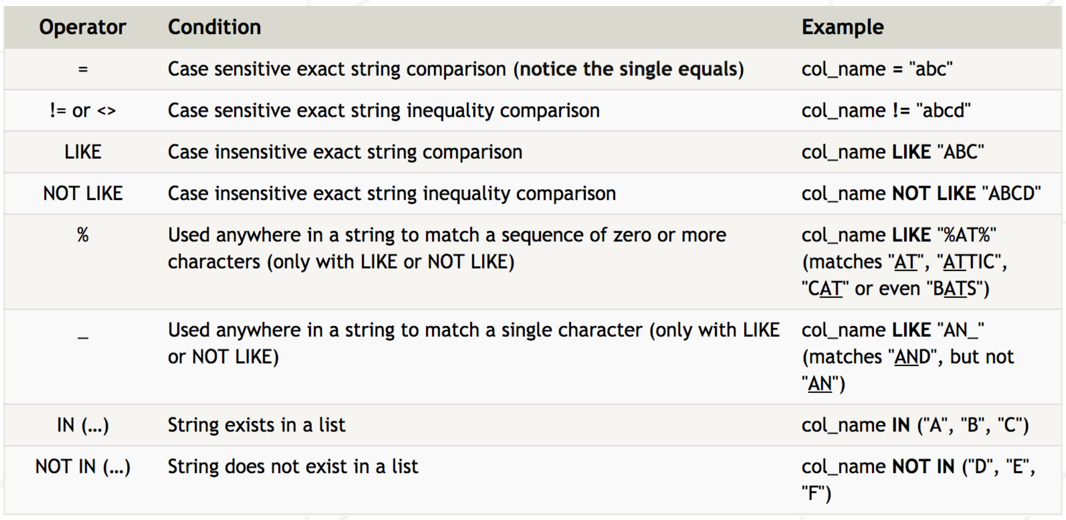
```
def example
  MovieDatabase.execute(<<-SQL)
    SELECT DISTINCT
      *
    FROM
      movies AS m
    WHERE
      m.title LIKE '%Star Wars%' AND m.yr > 1990;
  SQL
end
```

- SELECT chooses which columns to grab
- FROM chooses which table to start from
- DISTINCT means you want this to be unique! No duplicates

- * basically means that you'll grab ALL.
- You can aliases expressions or tables using "AS", this keeps your SQL code DRY.
- WHERE is used to filter records by condition!
- Now, what are some useful operators you can use in the WHERE clause?



Operator	Condition	SQL Example
=, !=, < <=, >, >=	Standard numerical operators	col_name **!=** 4
BETWEEN … AND …	Number is within range of two values (inclusive)	col_name **BETWEEN** 1.5 **AND** 10.5
NOT BETWEEN … AND …	Number is not within range of two values (inclusive)	col_name **NOT BETWEEN** 1 **AND** 10
IN (…)	Number exists in a list	col_name **IN** (2, 4, 6)
NOT IN (…)	Number does not exist in a list	col_name **NOT IN** (1, 3, 5)



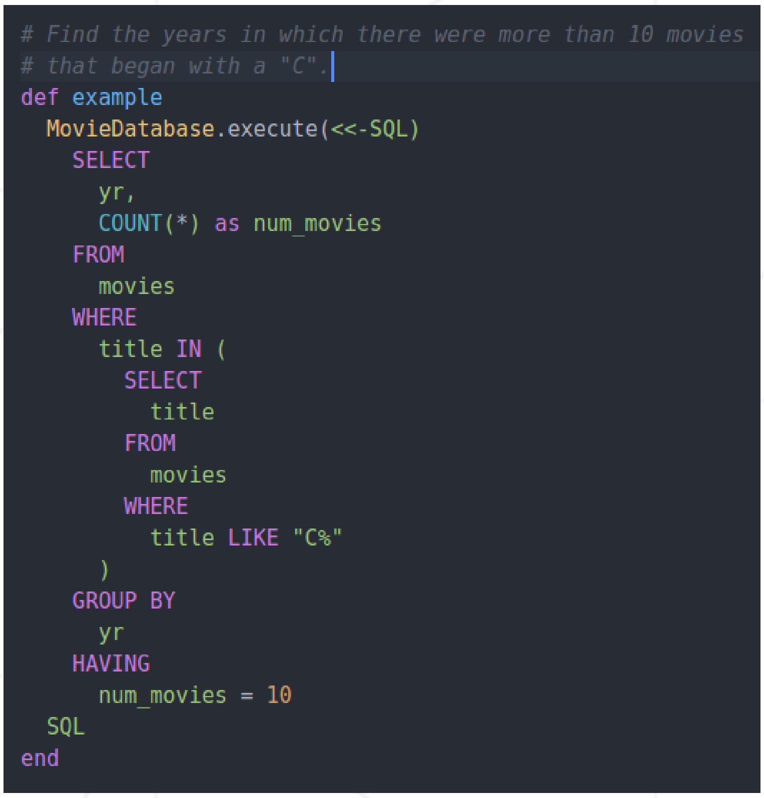
Operator	Condition	Example
=	Case sensitive exact string comparison (**notice the single equals**)	col_name = "abc"
!= or <>	Case sensitive exact string inequality comparison	col_name != "abcd"
LIKE	Case insensitive exact string comparison	col_name **LIKE** "ABC"
NOT LIKE	Case insensitive exact string inequality comparison	col_name **NOT LIKE** "ABCD"
%	Used anywhere in a string to match a sequence of zero or more characters (only with LIKE or NOT LIKE)	col_name **LIKE** "%AT%" (matches "AT", "ATTIC", "CAT" or even "BATS")
_	Used anywhere in a string to match a single character (only with LIKE or NOT LIKE)	col_name **LIKE** "AN_" (matches "AND", but not "AN")
IN (…)	String exists in a list	col_name **IN** ("A", "B", "C")
NOT IN (…)	String does not exist in a list	col_name **NOT IN** ("D", "E", "F")

Screenshots taken from SQLBolt!

Chapter VI

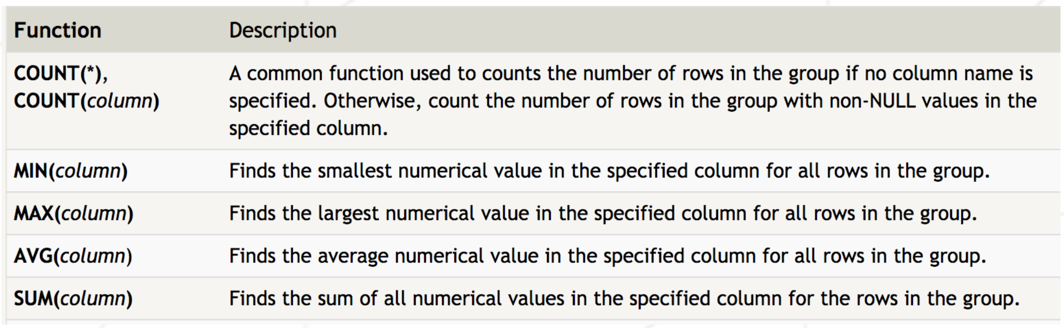
Day 01 : SLOWLY BECOMIN' POWERFUL IN SQL

	Exercise
SQL subqueries, GROUP BY, aggregate functions, HAVING	
Turn-in directory :	
Files to turn in : `04_subquery.rb, 05_aggregates.rb, 05a_aggregates.rb`	
Forbidden functions : `None`	
Notes : `Make sure to use docker cp to store your files on the host computer, then add it to a git repo after!`	



```ruby
# Find the years in which there were more than 10 movies
# that began with a "C".
def example
  MovieDatabase.execute(<<-SQL)
    SELECT
      yr,
      COUNT(*) as num_movies
    FROM
      movies
    WHERE
      title IN (
        SELECT
          title
        FROM
          movies
        WHERE
          title LIKE "C%"
      )
    GROUP BY
      yr
    HAVING
      num_movies = 10
  SQL
end
```

- Subqueries are easy, aren't they? It's just a query inside of another query! However, think about the type of subquery: is it correlated? A correlated subquery is where the subquery is dependent on a column from the outer query, meaning it runs repeatedly.
- GROUP BY groups the same column results together, allowing you to use aggregate functions. What happens when you use GROUP BY on more than one column?
- HAVING is just like WHERE, but it happens after the GROUP BY rather than before. We'll get to SQL's logical order later, for now– just know that SQL's syntactical order isn't its logical order.
- Aggregate functions are helpful functions you can use on groups! What are some of the most commonly used ones?
- If you look carefully, the text says "Find the years in which there were more than 10 movies", but I only put HAVING num_movies = 10. This error was merely a test to see your level of astuteness.



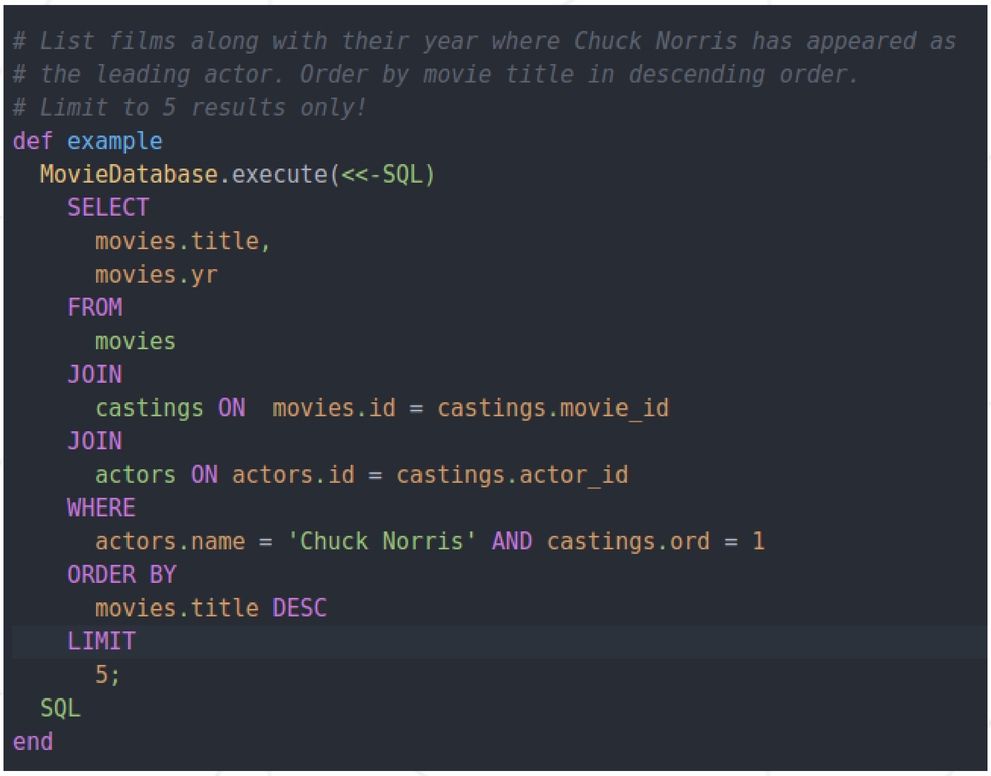
Function	Description
COUNT(*), COUNT(*column*)	A common function used to counts the number of rows in the group if no column name is specified. Otherwise, count the number of rows in the group with non-NULL values in the specified column.
MIN(*column*)	Finds the smallest numerical value in the specified column for all rows in the group.
MAX(*column*)	Finds the largest numerical value in the specified column for all rows in the group.
AVG(*column*)	Finds the average numerical value in the specified column for all rows in the group.
SUM(*column*)	Finds the sum of all numerical values in the specified column for the rows in the group.

Another chart from SQLBolt!

Chapter VII

Day 02 : YOUR POWER LEVEL IS RISING.

	Exercise
SQL JOINS, ORDER BY, LIMIT, join tables, primary key, foreign key, DB schema	
Turn-in directory :	
Files to turn in : `06_joins.rb, 07_joins.rb, 07a_joins.rb`	
Forbidden functions : `None`	
Notes : `Make sure to use docker cp to store your files on the host computer, then add it to a git repo after!`	



```ruby
# List films along with their year where Chuck Norris has appeared as
# the leading actor. Order by movie title in descending order.
# Limit to 5 results only!
def example
  MovieDatabase.execute(<<-SQL)
    SELECT
      movies.title,
      movies.yr
    FROM
      movies
    JOIN
      castings ON  movies.id = castings.movie_id
    JOIN
      actors ON actors.id = castings.actor_id
    WHERE
      actors.name = 'Chuck Norris' AND castings.ord = 1
    ORDER BY
      movies.title DESC
    LIMIT
      5;
  SQL
end
```

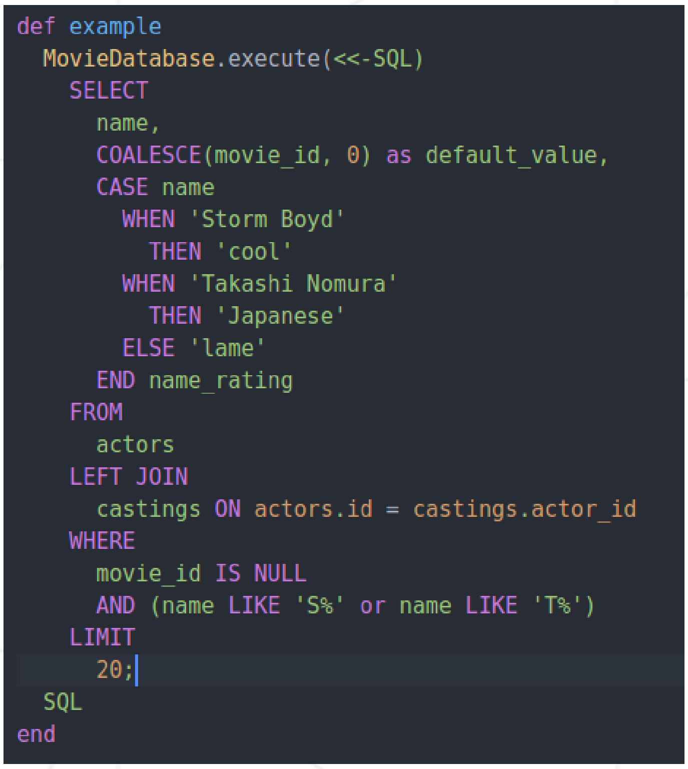
- What does a JOINS do? It combines rows from two or more tables together!
- ORDER BY orders your results based on some specific column, with options ASC or DESC
- LIMIT simply limits the amount of results you'll get back to a certain number
- A schema is basically just a view of the entire database
- Primary keys are used to uniquely identify each row in the table.
- Foreign keys are used to join two tables together!
- If an employees table has a boss_id foreign key, employees will belong to boss while boss will have many employees.
- A join table is needed to deal with a many-to-many relationship! Think about it for a lil' bit. Why is a join table needed?

Rather than posting an image of what happens on a regular join, here is a link to a visual representation of SQL joins.

Chapter VIII

Day 03 : AND THIS IS TO GO FURTHER BEYOND

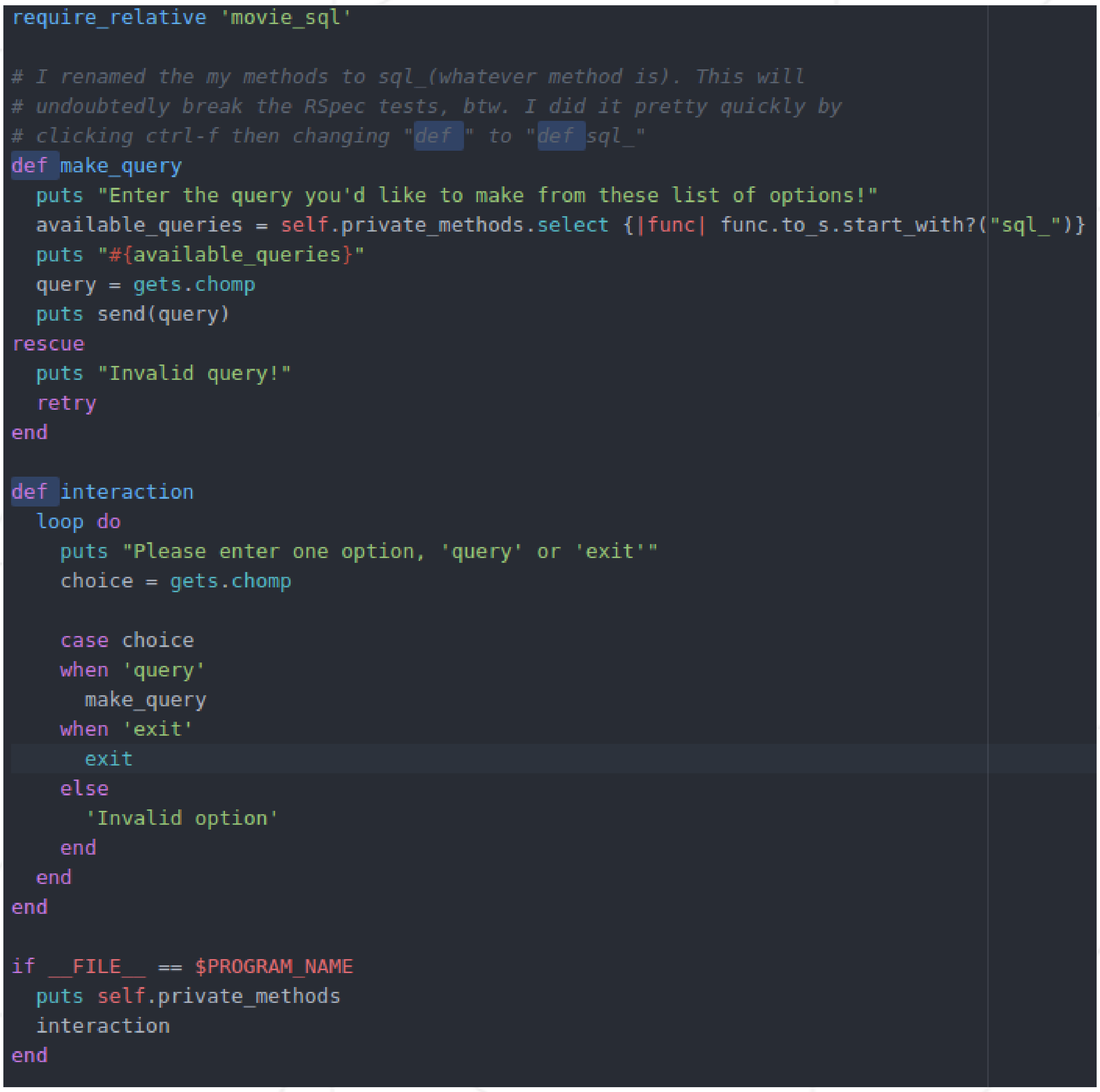
	Exercise
SQL IS NULL/NOT NULL, OUTER JOINS, COALESCE, CASE, self joins, ternary logic	
Turn-in directory :	
Files to turn in : `08_null.rb, 09_self_joins.rb, wrapper program`	
Forbidden functions : `None`	
Notes : `Make sure to use docker cp to store your files on the host computer, then add it to a git repo after!`	



```ruby
def example
  MovieDatabase.execute(<<-SQL)
    SELECT
      name,
      COALESCE(movie_id, 0) as default_value,
      CASE name
        WHEN 'Storm Boyd'
          THEN 'cool'
        WHEN 'Takashi Nomura'
          THEN 'Japanese'
        ELSE 'lame'
      END name_rating
    FROM
      actors
    LEFT JOIN
      castings ON actors.id = castings.actor_id
    WHERE
      movie_id IS NULL
      AND (name LIKE 'S%' or name LIKE 'T%')
    LIMIT
      20;
  SQL
end
```

- Why do you have to use IS NULL/IS NOT NULL instead of just = or != NULL? Recall that NULL is unknown, so if you compare NULL = NULL, you get false since they're both unknown.
- Outer joins! Look above to the visual representation of SQL joins. Think about why outer joins can result in NULLs.
- CASE statement is self explanatory
- COALESCE is used to fill out NULL values with default ones!
- In what cases would you use self joins and why?
- What is SQL's ternary logic? -> True, false, null

Look at this example of a basic wrapper program!
This is pretty poor, though. I had to use the main Object to get all the methods! I'd say when you create your own DB, you should wrap up all your methods into some sort of object.



```ruby
require_relative 'movie_sql'

# I renamed the my methods to sql_(whatever method is). This will
# undoubtedly break the RSpec tests, btw. I did it pretty quickly by
# clicking ctrl-f then changing "def " to "def sql_"
def make_query
  puts "Enter the query you'd like to make from these list of options!"
  available_queries = self.private_methods.select {|func| func.to_s.start_with?("sql_")}
  puts "#{available_queries}"
  query = gets.chomp
  puts send(query)
rescue
  puts "Invalid query!"
  retry
end

def interaction
  loop do
    puts "Please enter one option, 'query' or 'exit'"
    choice = gets.chomp

    case choice
    when 'query'
      make_query
    when 'exit'
      exit
    else
      'Invalid option'
    end
  end
end

if __FILE__ == $PROGRAM_NAME
  puts self.private_methods
  interaction
end
```

Chapter IX

Bonus part

Whoa, I can't believe you actually finished everything. You cheated, didn't you?! Well, whatever. Now, it's time for bonuses! Well, there's really only one plausible bonus for this project, so let's get right to it.

Your bonus will be creating 10 additional queries for information you want to get from the database! Write these 10 queries in a separate file, then make a skeleton and spec file for said queries. After that, challenge your peers to solve said queries! You will get the bonus points provided none of your peers can solve all 10 queries that you've created.

If your material is good, it'll be added to the curriculum to make the next gen of H2S SQL students suffer!

Chapter X

Turn-in and peer-evaluation

Turn your work in using your `GiT` repository, as usual. Only work present on your repository will be graded in defense.

Remember to store all your work on a git repository– make sure to add, commit, then push!

Good luck and don't forget to "cd (where you stored your work) && rm -rf *" once you're done.

.

.

.

.

.

.

.

.

.

.

.

.

.

If you've actually followed instructions and stored your work, you'll be fine.