MySQL Assignment

Second assignment

Tyler Slack

2024

Contents

[MySQL Task 1 2](#_Toc182388881)

[Task 1 2](#_Toc182388882)

[Task 2 2](#_Toc182388883)

[Task 3 2](#_Toc182388884)

[Task 4 3](#_Toc182388885)

[Task 5 4](#_Toc182388886)

[Task 6 4](#_Toc182388887)

[EER Diagram – Task 1 5](#_Toc182388888)

[MySQL Task 2 6](#_Toc182388889)

[Task 1 6](#_Toc182388890)

[Task 2 6](#_Toc182388891)

[Task 3 7](#_Toc182388892)

[Task 4 7](#_Toc182388893)

[Task 5 8](#_Toc182388894)

[Task 6 8](#_Toc182388895)

[Task 7 8](#_Toc182388896)

[Task 8 9](#_Toc182388897)

[Task 9 9](#_Toc182388898)

[Optional 9](#_Toc182388899)

[EER Diagram – Task 2 10](#_Toc182388900)

[Interview Questions 11](#_Toc182388901)

# MySQL Task 1

## Task 1

For Task 1, I modified the query code to achieve two objectives:

**Updated Points Calculation:** Changed the points calculation to *(points \* 10) + 100*, displaying it alongside customer names.

**Added Discount Factor:** Created a *discoun*t column with the formula *(points + 10) \* 100*, showing a new value in a header labelled "*discount*."

These adjustments allow us to see both the modified points and an additional discount factor for each customer.

## Task 2

For Task 2, I wrote a SQL query to display each product's original price and an updated price with a 10% increase:



**Original Price Column:** Retrieved each product's name and *unit\_price*.

**New Price Calculation:** Created a *new\_price* column that applies a 10% increase to *unit\_price* using the expression (*unit\_price \* 1.1*).

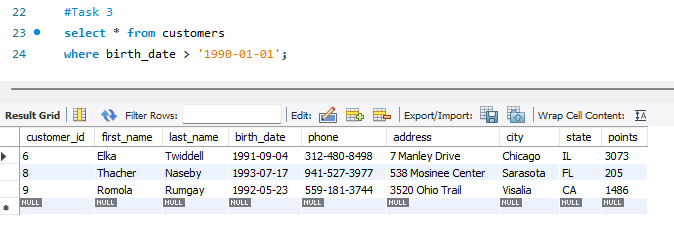
This query allows us to view each product’s original price alongside its updated price after the increase.

## Task 3

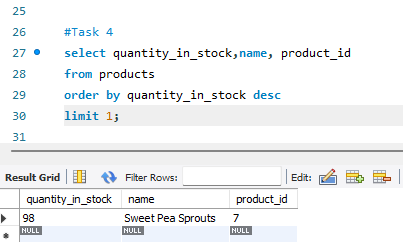
For or Task 3, I created a SQL query to identify all customers born after January 1, 1990:

**Retrieved Customer Data:** Selected all columns from the customer’s table.

**Filtered by Birth Date:** Used a *WHERE* clause to include only customers with a *birth\_date* greater than *'1990-01-01'*.

This query helps us list all customers who were born after this specific date.

## Task 4

For Task 4, I wrote a SQL query to find the product with the highest stock quantity:

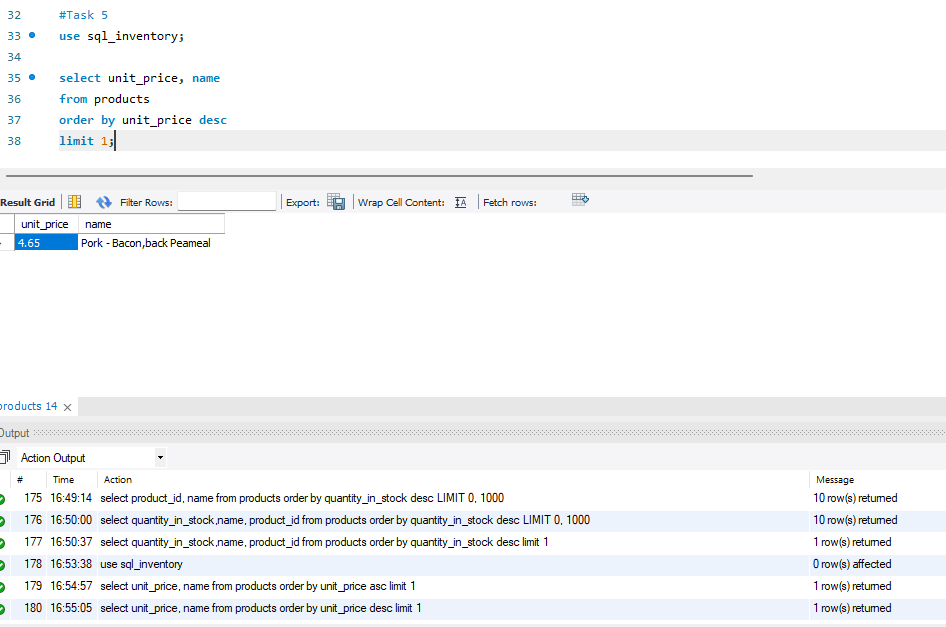
**Selected Relevant Columns:** Retrieved *quantity\_in\_stock*, *name*, and *product\_id* from the *products* table using *Select.*

**Sorted by Stock Quantity:** Used *ORDER BY* *quantity\_in\_stock* *DESC* to sort products in descending order by *quantity\_in\_stock*.

**Limited Result to Top Product:** Applied *LIMIT 1* to show only the product with the highest stock quantity.

This query provides the product name, ID, and stock quantity for the item with the most stock.

## Task 5

For Task 5, I wrote a SQL query to find the most expensive product:

**Selected Relevant Columns:** Retrieved *unit\_price* and name from the products table.

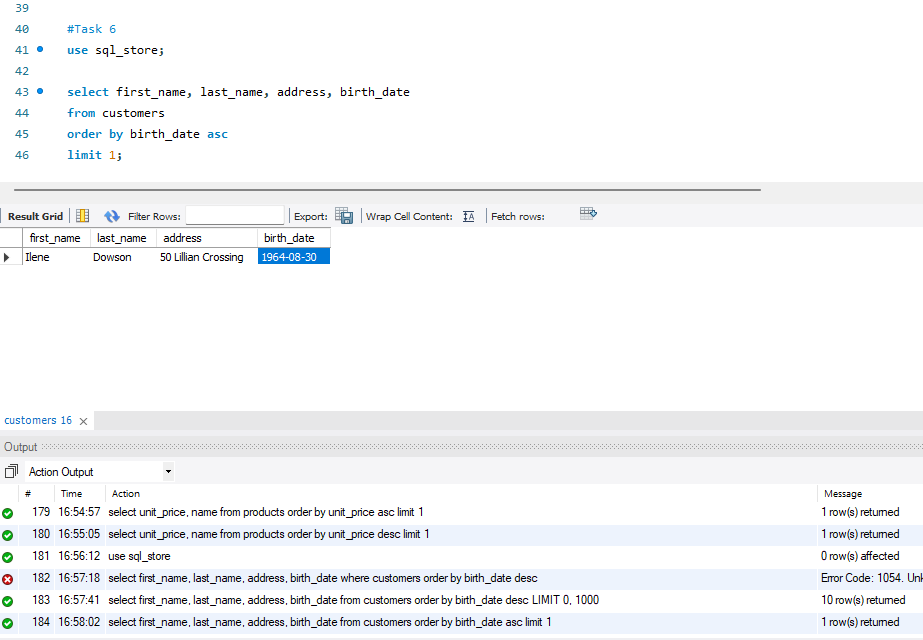
**Sorted by Price:** Used *ORDER* BY *unit\_price* *DESC* to sort the products in descending order by *unit\_price*, so the most expensive product appears at the top.

**Limited to One Result:** Applied *LIMIT 1* to return only the most expensive product.

This query shows the name and price of the product with the highest unit price.

## Task 6

For Task 6, I wrote a SQL query to find the oldest customer based on their birthdate:

**Selected Relevant Columns:** Retrieved *first\_name*, *last\_name*, *address*, and *birth\_date* from the *customers* table.

**Sorted by Birth Date:** Used *ORDER BY* *birth\_date* *ASC* to sort customers in ascending order by *birth\_date*, so the oldest customer (with the earliest birth date) appears first.

**Limited to One Result:** Applied *LIMIT* 1 to return only the oldest customer.

This query provides the *first\_name*, *last\_name*, *address*, and *birth\_date* of the oldest customer in the database.

## EER Diagram – Task 1

In relational databases, tables store structured data, with *primary* and *foreign* keys linking them for organized relationships and efficient querying.

* *Primary Key*: Uniquely identifies each record in a table.
* *Foreign Key*: Links a field in one table to the *primary key* of another, establishing a relationship between the two.

In the given example:

1. **Customers**: Stores customer data, with *customer\_id* as the *primary key*.
2. **Products**: Contains product details, with *product\_id* as the *primary key*.
3. **Orders**: Records customer orders, with *order\_id* as the *primary key* and a *foreign key* linking to *customer\_id*.
4. **Order\_Items**: Details individual order items, with *foreign keys* to both *order\_id* and *product\_id*.
5. **Shippers**: Contains shipper details, with *shipper\_id* as the *primary key*, linking to orders.
6. **Order\_Statuses**: Tracks order statuses, with *order\_status\_id* as the *primary key*, linking to orders.

These relationships enable easy retrieval and management of order-related data across different tables.

# MySQL Task 2

## Task 1

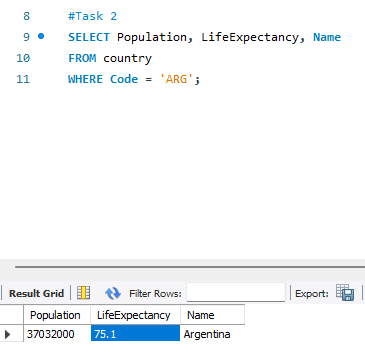
For Task 1, I wrote a SQL query to count the number of cities in the USA:

**Selected Count of All Rows:** Used *COUNT(*\*) to count all rows in the *city* table.

**Filtered by Country Code:** Applied a *WHERE* clause with *CountryCode = 'USA'* to limit the count to cities located in the USA.

This query returned the total number of cities in the USA in the *city* table.

## Task 2

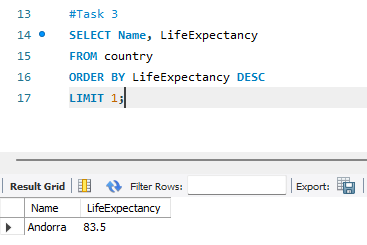
For Task 2, I wrote a SQL query to find the population and average life expectancy for Argentina:

**Selected Relevant Columns:** Retrieved *Population*, *LifeExpectancy*, and *Name* from the *country* table.

**Filtered by Country Code:** Used a *WHERE* clause with *Code = 'ARG'* to specifically get information for Argentina.

This query displays the population, life expectancy, and country name for Argentina.

## Task 3

For Task 3, I wrote a SQL query to find the country with the highest life expectancy:

**Selected Relevant Columns:** Retrieved *Name* and *LifeExpectancy* from the *country* table.

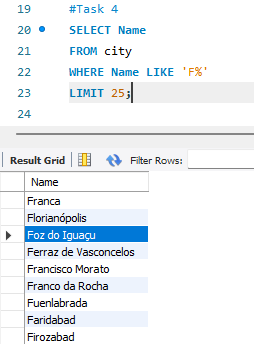
**Sorted by Life Expectancy:** Used *ORDER BY LifeExpectancy DESC* to arrange countries in descending order, with the highest life expectancy at the top.

**Limited to One Result:** Applied *LIMIT 1* to show only the country with the highest life expectancy.

This query returns the name and life expectancy of the country with the highest value in the database.

## Task 4

For Task 4, I wrote a SQL query to select 25 cities around the world that start with the letter "F":



**Selected City Name:** Retrieved the *Name* column from the *city* table.

**Filtered by Initial Letter:** Used *WHERE Name LIKE 'F%'* to find cities whose names begin with "F".

**Limited to 25 Results:** Applied *LIMIT 25* to return only the first 25 cities that match this condition.

This query displays up to 25 city names starting with "F" from the *city* table.

## Task 5

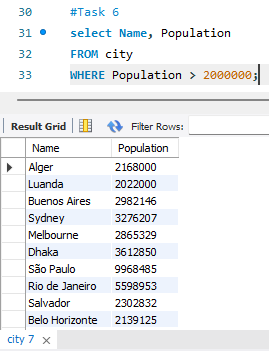
For Task 5, I wrote a SQL query to display the first 10 rows with columns for city ID, name, and population:

**Selected Relevant Columns:** Retrieved *ID*, *Name*, and *Population* from the *city* table.

**Limited to 10 Results:** Applied *LIMIT 10* to display only the first 10 rows.

This query provides the ID, name, and population for the first 10 cities in the *city* table.

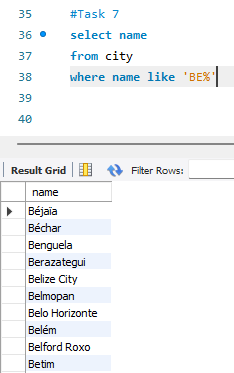
## Task 6

For Task 6, I wrote a SQL query to find cities with populations greater than 2,000,000:

**Selected Relevant Columns:** Retrieved *Name* and *Population* from the *city* table.

**Filtered by Population:** Used a *WHERE* clause with *Population > 2000000* to include only cities with a population over 2 million.

This query returns the names and populations of cities with more than 2 million residents.



## Task 7

For Task 7, I created a SQL query to list city names that start with "Be":

**Selected City Name:** Pulled the *name* column from the *city* table.

**Filtered by Prefix:** Used *WHERE name LIKE 'Be%'* to match cities beginning with "Be" (case-sensitive).

This gives us a list of cities whose names start with "Be" from the *city* table.

## Task 8

For Task 8, I wrote a SQL query to find cities with populations between 500,000 and 1,000,000:

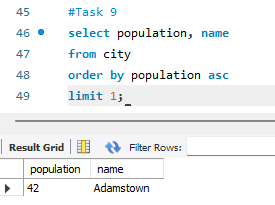
**Selected Relevant Columns:** Retrieved *population* and *name* from the *city* table.

**Filtered by Population Range:** Used *WHERE population BETWEEN 500000 AND 1000000* to include only cities within this range.

This query gives us a list of cities with populations between 500,000 and 1,000,000.

## Task 9

For Task 9, I wrote a SQL query to find the city with the lowest population:

**Pulled Relevant Data:** Grabbed the *population* and *name* columns from the *city* table.

**Sorted by Population:** Ordered the cities in *ascending* order by population using *ORDER BY population ASC*, so the city with the smallest population comes first.

**Limited to One Result:** Applied *LIMIT 1* to fetch only that one city.

This query returns the city with the lowest population from the *city* table.

## Optional

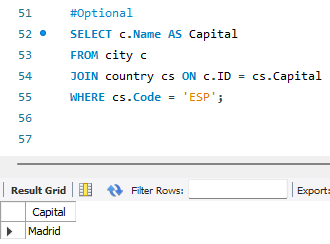
For the optional task, I wrote a SQL query to find the capital of Spain (ESP):

**Joined the Tables:** Used a *JOIN* between the *city* table (aliased as c) and the *country* table (aliased as cs) based on the condition that the *ID* from the *city* table matches the *Capital* field in the *country* table.

**Filtered by Country Code:** Applied *WHERE cs.Code = 'ESP'* to specifically get information for Spain.

**Renamed the Column:** Used *AS Capital* to label the result as "Capital."

This query gives us the name of the capital city of Spain by joining the relevant data from the two tables.



## EER Diagram – Task 2

* **Primary Key in the country table**: Code
* **Primary Key in the city table**: ID
* **Primary Key in the countrylanguage table**: CountryCode, Language (composite primary key)
* **Foreign Key in the city table**: CountryCode (references Code in the country table)
* **Foreign Key in the countrylanguage table**: CountryCode (references Code in the country table)

These relationships link cities and languages to their respective countries.

# Interview Questions

#### What is a Query?

A query in MySQL is basically a request you make to a database to get or manipulate data. It’s written in SQL, which stands for Structured Query Language. For example, when you want to retrieve data, you’d use a SELECT query. If you want to add, update, or delete data, you’d use INSERT, UPDATE, or DELETE queries respectively. Essentially, a query helps you communicate with the database to get the results you need or make changes to the data.

#### What is the SELECT statement?

The SELECT statement in SQL is used to pull data from a database. When you use SELECT, you're telling the database what information you want to see. For example, if you want to get everything from a table, you’d use SELECT \*. But if you just want specific columns, like e.g., names or job titles, you'd list those after SELECT. You can also add conditions with WHERE to filter the results or use ORDER BY to sort them. In essence, it's the way you query the database to get the data you’re looking for.

#### What is the WHERE clause?

The WHERE clause is used to almost filter records based on given conditions. For example, it allows you to narrow down the data you retrieve by specifying criteria. In the previous tasks, we had to find customers who were born after 01/01/1990. WHERE was used here to only retrieve those entries that matched the WHERE criteria given, allowing us to target specific rows.

#### What is the Primary key?

A primary key is a unique identifier for every record in a database table. It makes sure that no two roles can have the same value within a column. Primary keys are also integral in establishing relationships between various tables.

#### What is a Database?

A database is essentially a structured collection of data that’s stored in a way that can be accessed, updated and organised by various software. It allows information to be stored in tables, allowing for relationships to be built between them. Databases are also designed to handle large amounts of data efficiently in order to retrieve or manipulate data whenever necessary.

#### What is Normalisation?

Normalisation is the process of organising a database in a way that makes it more efficient. Usually, this means a reduction of duplicate entries by breaking up large pieces of data into smaller individual tables with relationships instead, so each set of data is only stored once. This helps to keep a database ‘clean’.

#### Modify query to show the population of Germany.

SELECT population FROM world

WHERE name = 'France'

SELECT population FROM world

WHERE name = 'Germany'

#### Select the query which gives the name of countries beginning with U.

**SELECT** name **FROM** world **WHERE** name **LIKE** 'U%'

#### Select the answer which shows the problem with this SQL code - the intended result should be the continent of France:

**SELECT** continent **FROM** world **WHERE** 'name' = 'France'

B) 'name' should be name

**SELECT** continent **FROM** world **WHERE** name = 'France'

Name in quotes causes MySQL to treat it as a string.

#### Select the code which shows the countries that end in A or L.

**SELECT** name **FROM** world **WHERE** name **LIKE** '%a' **OR** name **LIKE** '%l'

#### Given the table on the left, select the query which produces this table below

A table with numbers and letters

Description automatically generated

**SELECT** name, population **FROM** world **WHERE** population **BETWEEN** 1000000 **AND** 1250000