1`

SQL Assignment

An introduction to relationaldatabases

Giacomo Carta

20/10/2023

Contents

[Introduction 3](#_Toc149046351)

[Databases overview 3](#_Toc149046352)

[The relational model 3](#_Toc149046353)

[The importance of relationships 3](#_Toc149046354)

[Disadvantages 4](#_Toc149046355)

[Important concepts 4](#_Toc149046356)

[Task 1: List the different types of relationships in relational databases and provide examples. 5](#_Toc149046357)

[One-to-One (1:1) relationships 5](#_Toc149046358)

[Possible issues with 1:1 relationships 6](#_Toc149046359)

[One-to-Many (1:N) relationships 7](#_Toc149046360)

[Many-to-Many(N:N) relationships 8](#_Toc149046361)

[Possible issues with N:N relationships 8](#_Toc149046362)

[Reducing redundancy: Junction tables 9](#_Toc149046363)

[Task 2: What is Normalization and why is it important to database development? 12](#_Toc149046364)

[Normalization overview 12](#_Toc149046365)

[Importance of data integrity 12](#_Toc149046366)

[The problem with Data redundancy 12](#_Toc149046367)

[When to de-normalize 13](#_Toc149046368)

[Example of de-normalization 13](#_Toc149046369)

[Task 3: Using COUNT, get the number of cities in the USA 14](#_Toc149046370)

[Task 4: Find out what the population and life expectancy for people in Argentina (ARG) is 16](#_Toc149046371)

[Task 5: Using ORDER BY, LIMIT, what country has the highest life expectancy? 17](#_Toc149046372)

[Task 6: Select 25 cities around the world that start with the letter 'F' in a single SQL query. 19](#_Toc149046373)

[Task 7: Create a SQL statement to display columns “Id”, “Name”, “Population” from the “city” table and limit results to first 10 rows only. 20](#_Toc149046374)

[Task 8: Create a SQL statement to find only those cities from city table whose population is larger than 2,000,000. 21](#_Toc149046375)

[Task 9 (Optional): Create a SQL statement to find all city names from city table whose name begins with “Be” prefix. 22](#_Toc149046376)

[Task 10 (Optional): Create a SQL statement to find only those cities from city table whose population is between 500,000 - 1,000,000. 23](#_Toc149046377)

[Task 11 (Optional): Create a SQL statement to find a city with the lowest population in the city table. 24](#_Toc149046378)

[Task 12 (Optional): Create a SQL statement to show the population of Switzerland and all the languages spoken there. 26](#_Toc149046379)

[Extension: using Stored Procedures to run queries with a simple click 29](#_Toc149046380)

[Task 13: Creating an EER Diagram 31](#_Toc149046381)

[Task 14: Use the EER diagram to answer the following: 33](#_Toc149046382)

[A) Identify the primary key in country table. 33](#_Toc149046383)

[B) Identify the primary key in city table. 33](#_Toc149046384)

[C) Identify the primary key in countrylanguage table. 33](#_Toc149046385)

[D) Identify the foreign key in city table. 34](#_Toc149046386)

[34](#_Toc149046387)

[E) Identify the foreign key in countrylanguage table. 34](#_Toc149046388)

[Reflections 35](#_Toc149046389)

# Introduction

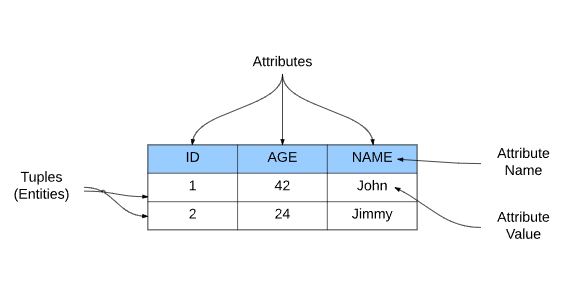
## Databases overview

Databases are tools used for storing, managing and retrieving large volumes of data. In today’s world, they play a crucial role in organising information, therefore they are used extensively in all industries. Throughout the years, they have evolved to become more efficient and their structure has adapted to manage and organise massive amounts of data, in order to grant easy access, retrieval and manipulation of information. Because of databases, companies are able to analyse data and take informed decision, rationalise operations and improve productivity.

## The relational model

In 1970, a computer scientist named Edgar Frank Codd came up with a theory to organise databases in a more efficient way, using tables to store information. The relational model (RM) was a new approach of managing data using four main principles:

* **Relations**: Data is organised into relations, which are tables composed of rows and columns.
* **Tuple**: a row, representing a single instance or record.
* **Attribute**: a column, which represent a property of an instance.
* **Value**: a specific valid value for the type of attribute.



## The importance of relationships

In relational databases, relationships have an important role as they help with:

* **Data integrity**: it is possible to create rules such as delete cascades (deleting one record, other related records in other tables are deleted) or restrict actions that would violate data integrity.
* **Efficiency**: with relational databases, it is possible to use functions such as joins in SQL to combine data from multiple tables based on the defined relationships. This helps with retrieval of complex data by returning only the information that is needed for a specific task.
* **Data redundancy**: Relationships mitigate issues that might arise due to storage of large volumes of data. By saving related data across multiple tables reduces duplication and inconsistencies.

## Disadvantages

Some of the potential disadvantages or limitations of relational databases include:

* **Complexity**: relational databases can be complex to design and manage, and initial training is required for programmers and users.
* **Cost**: relational databases can be expensive to maintain and to build as they require substantial hardware and software start-up costs.
* **Rigidity**: relational databases can be difficult to modify. To function, they need a defined structure to work (called “schema”), that must be updated before making any alterations that involve adding attributes or changing table design.

## Important concepts

Before analysing the different relationships that can exist within databases, it is important to introduce some basic but paramount concepts:

* **Primary Key**: a primary key is a unique identifier for a record in a table. This ensures that there is no duplication of instances in a database. It is a crucial element in building the relationships between tables and enables efficient joins. Every entity **must have** one.
* **Foreign Key**: a foreign key is used when creating links between tables by referencing a PK in another table. This enforces referential integrity which is, as seen before, a big part of relational databases.
* **Schema**: a database schema is the structure or blueprint of a database. It represents its tables, fields, relationships and constraints, visually displaying the logical organisation of data within the database. It is designed via an **ERD** (Entity-Relationship Diagram), which is a more abstract and conceptual model that is used to plan the logical or physical structure of a database.

# Task 1: List the different types of relationships in relational databases and provide examples.

## One-to-One (1:1) relationships

With one-to-one relationships, each record in one table can be related to only one record in another table (and vice versa).

A black line on a white background

Description automatically generated

An example of this type of relationship can be an individual and their passport. Every person has one passport and **one only**, every passport is associated with one person and **one only**.

Another example can be the one below:

A diagram of a diagram

Description automatically generated

In the **ERD** above, a library has a database with a table called “Books” and one called “BookDetails”. In the **Books** table are stored all information about titles and authors of books; in the **BookDetails** table are stored all information about the year in which the books were published.

A screenshot of a computer

Description automatically generatedThis is how the two tables could look:

Primary Key

Foreign Key

Primary Key

In this image, we can see that both tables have unique values to identify each record; these are the previously mentioned Primary Keys.In the **Books** and **BookDetails** tables, these are respectively the “BookID” and the “DetailID” column. This means that no two books, nor two set of details share the same identifier.

As for the Foreign Key, in the **BookDetails** table we notice that the column “BookID” refers directly to the **Books** table, creating a link between the two.

Each record/row in the **Books** table is linked to **only one** detail record/row in the **BookDetails** table. What tells us that this is a 1:1 relationship is that:

* Each book has only one set of details in the **BookDetails** table
* Each set of book details is linked to one specific book in the **Books** table

## Possible issues with 1:1 relationships

The example above raises immediately a doubt: is there actually a need to have the year of publication in a separate table? The simple answer is, absolutely **not**. In fact, one-to-one relationships can add complexity to a database design, creating repetitive links which can lead to data redundancy and can have a negative impact on the overall performance of a database.

In this case, a better way to store the book information would be a single table called **Books** with BookID (PK), Title, Author and Year as columns/attributes.

## One-to-Many (1:N) relationships

One-to-many relationships are the most commonly used in relational databases. In this case, one record from one table is linked to many records in another.

A black line on a white background

Description automatically generated

An example could be an e-commerce website, where one customer can make many orders, but a single, specific order can belong to one and **only one** customer.

A diagram of a diagram

Description automatically generated

The unique record identifiers, or Primary Keys, are “CustomerID” and “OrderID”; the Foreign Key, which links the **Orders** table to the **Customers** table, is “CustomerID” in the **Orders** table. This is a one-to-many relationship because:

* One customer, identified by the “CustomerID” value, can place multiple orders.
* One order, identified by the “OrderID” value, can belong only to one customer.

A screenshot of a computer

Description automatically generated

As we can see, Mike Byson and Tina Burner can place **multiple** orders, but each order belongs to one and **only one** of them (e.g. order **445** belongs to Mike Byson and cannot belong to anyone else).

Note: when considering the relationship between the **Customers** and **Orders** table, we will have a **one-to-many** relationship; when considering the relationship between the **Orders** table and the **Customers** table, we will have a **many-to-one** relationship. The relationship is denoted as 1:N or N:1 where “1” indicates a single entity and “N” represents any number of related entities (0,1 or many ).

## Many-to-Many(N:N) relationships

Many-to-many relationships occur when multiple records in one table are related to multiple records in another table.

A black line on a white background

Description automatically generated

In our world this relationship is very common, for example a student can have multiple teachers and a teacher can teach many students:

A diagram of a graph

Description automatically generated with medium confidence

A visual representation of the tables could be:

A table with text and numbers

Description automatically generatedIn these tables, we see that James is enrolled with Mr. Smith and Mr. Jones; also, we see that Mr. Brown teaches Charlie and Lola.

However, there are a few problems with this approach, that we will discuss in the next paragraph.

## Possible issues with N:N relationships

As stated above, the previous representation raises some problems. Many-to-many relationships are inefficient, as the same data needs to be stored in two different tables, creating redundant values.

Another issue would be to update the database: for example, if James decides to take English with Mr. Brown, it would be necessary to update both tables, adding the “TeacherID” value from the second table to the “Teacher” column in the first, then adding the “StudentID” value from the first table to the “Student” column in the second.

All of this would result in unnecessary complexity, more confusion, increased risk of errors and more laborious data retrieving processes. Fortunately, there is a way to solve this problem.

## Reducing redundancy: Junction tables

Creating an ERD with many-to-many relationships can lead to several problems. A solution to this is to create a **junction table** that will have two attributes/columns, one for “StudentID” and one for “TeacherID”, like this:

A diagram of a student

Description automatically generated

By creating the “**student\_teacher**”table, we can represent the relationship between two entities without having to create a separate table to show each student-teacher pair. The benefits will be:

* More efficiency, as we need to store the relationship data in one table only
* Much easier to update, as we only need to update the junction table to add more students to teachers and vice versa
* Faster to create queries using the **JOIN** command (later on this topic)

To give a visual representation of this in Microsoft Access:

A screenshot of a computer

Description automatically generated

We can see that the student\_id (PK in the “students” table) and student\_id (FK in the “student\_teacher” table) create a one-to-many relationship, as one student (in the “students” table) can be associated with multiple teachers (in the “student\_teacher” table). The same happens with teacher\_id (PK in the “teachers” table) and teacher\_id (FK in the “student\_teacher” table).

We will now have three tables, one storing students’ data, one for teachers’ data and one to show the multiple combinations that can occur between students and teachers.

A screenshot of a computer

Description automatically generatedThe “students” and “teachers” tables will look like this:

A screenshot of a computer

Description automatically generated

This will be the junction table:

A screenshot of a calendar

Description automatically generated

As we can see, there are many possible combinations, as one teacher can teach many students, and a student can have multiple teachers.

Now, if we want to query the database to see the students that one teachers have in their course, or to see how many courses a student has enrolled, it will be very simple.

Let’s look at James and let’s create a query to see which courses he is taking an who are his teachers:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generatedThe result will be a table with all records for James:

Attribute from the “students” table.

Attributes from the “teachers” table.

In conclusion, as in the real world the many-to-many relationships are numerous, using junction tables can save a lot of time to data analysts and database users, making maintenance and querying a more efficient process.

# Task 2: What is Normalization and why is it important to database development?

## Normalization overview

Normalization plays a big role in the context of database development. It aims to organise and structure data in a relational database to eliminate data redundancy and improve data integrity. To do so, complex data structures are broken down into simpler, more manageable tables. A database that is normalized is more efficient, easier to maintain and to manipulate.

## Importance of data integrity

Data integrity is a crucial element in every database. It ensures that all information stored within the database is accurate, consistent and reliable. Without normalization, data can be inconsistent and can be recorded multiple times in different ways. For example, the same product could appear with various names, making it challenging to make sense of the data.

A screen shot of a computer

Description automatically generatedWith all these data inconsistencies to describe the name of the same entity, it would be hard to understand trends in sales for the crisps, as there would be data left out of queries or numerous queries needed.

## The problem with Data redundancy

Redundancy occurs when the same data is stored in multiple tables in a relational database. This redundancy wastes storage space and makes it challenging to maintain consistency and accuracy across the database.

A group of names on a blue background

Description automatically generated

The duplication of the customer’s information in the table is an example of data redundancy. If John Wayne decides to move to another house, updating the values would be a complex process that could lead to errors. To avoid issues with this, we would have to normalize the database and have two separate tables, one for customers and one for orders, like this:

A screenshot of a computer

Description automatically generated

Having normalized the database, we can now update the address in the customers table without having to do changes elsewhere and minimising the risk of errors and inconsistencies.

## When to de-normalize

Although normalization is the standard practice in database design to improve data integrity and reduce redundancy, in some situations de-normalization can be a sensible option. It can be used to simplify complex joins, optimise query performance and generally improve the efficiency in maintaining large and intricate databases.

An example where this procedure can be used is in “data warehousing”, a process that involves collecting, storing and managing data from multiple sources in a centralised repository. Here, data is organised in such a way that is easier for companies to extract information among several sources.

## Example of de-normalization

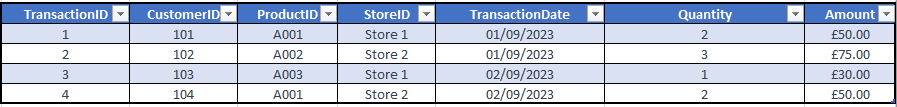
For instance, a company that has several sites in different locations uses data warehouses to store the separate data from each site into a central location. Doing this, they can easily retrieve data to understand trends, check overall sales and track inventory levels.

In this scenario, de-normalization could be beneficial for a few reasons:

* De-normalized data will be stored in fewer, although larger, tables. This will result in less use of complex joins required for querying and an overall faster query performance
* The company will be able to produce reports that provide a comprehensive view of sales across all sites in a simpler way

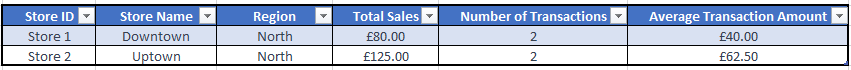
Let’s say that the company wants to generate a report to compare sales in the North region. A normalized table would require multiple joins to gather the information from different sources. In this case, a de-normalized table would be the better option.

Let’s look at how a normalized table would appear:



We can see that data is stored at its most granular level. Although this table is a great example of following normalization rules, it heavily relies on relationships to connect to other tables in order to gain extra knowledge about the data. For example, we would need to look at the **Stores** table to see the names and regions of the stores with the most sales.

Here is how a de-normalized table could look:

We see that a de-normalized table structure simplifies the process of generating a sales report for the North region. All the needed data is combined into a single table (instead of being stored in several tables), reducing the need for complex joins and increasing query performance.

# Task 3: Using COUNT, get the number of cities in the USA

The first task with MySQL involves using the COUNT function to count the number of records in the database, then filtering the result, using the WHERE clause, to only return instances that include the attribute USA. We first start selecting the database that we want to work with using this line of code:



Now, to get familiar with the database, there are different ways to proceed. I decided to take a look at the city table:



This is the result, a long list with 4079 records, each representing a city in the database:

A screenshot of a computer

Description automatically generated



We can see that the column “CountryCode” gives us information about the country. Therefore, we can do two things:

* Scrolling down until we find USA among the values.
* Looking for more information in another table.

As the first option is self-explanatory, we are going to attempt the second method. To do so, we will select another table between all the ones that are in the database.

A screenshot of a computer

Description automatically generated

The “country” table is the most sensible option. Again, using the SELECT \* option, we will take a look at the table.

This time the result will be only 239 rows, which will help to identify the USA without having to scroll down through too many records.

A screenshot of a computer

Description automatically generated



Immediately, it appears that the code used for United States is USA. As the United States of America are well known by their code, these steps might seem a waste of time; but in case we had to look for cities in Uruguay or somewhere else, the code would have been a bit more difficult to guess, so following these steps would be very useful to be sure we are selecting the right data.

Now that we know that the code we have to filter the records by is “USA”, we can go back to the “city” table, where we need to count all the records that include “USA” in the “CountryCode” column:

A screenshot of a computer

Description automatically generated

By using the formula above, MySQL will count the number of rows in the database where the country column is equal to USA. As we can see, the result is a single integer value, which is the number of cities in the USA in the database: **274**.

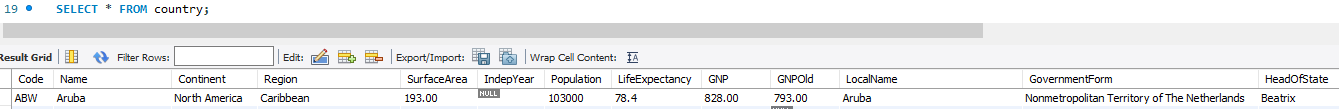
If we wanted to extend our search further, we could narrow the values down to the number of cities in the USA that are in California. To do so, we can simply add AND to our query:

A screenshot of a computer

Description automatically generated

# Task 4: Find out what the population and life expectancy for people in Argentina (ARG) is

In this case we now the name and country code of the country that we have to investigate. To find out all of this information, we are going to start by simply having a look at the “country” table:





This has the goal to help us visualise the attributes we will be analysing. Another quick way to see this is by looking at the “Schemas” window on the left of the screen:

A screenshot of a computer

Description automatically generated



Looking at the column names is very useful, so that, when we try to retrieve data from databases, we don’t make any errors selecting the right attributes with the wrong names.



In this case, we need to show the columns “Name”, “Population” and “LifeExpectancy”, using the condition WHERE to select only Argentina:

A screenshot of a computer

Description automatically generated

This shows that the population of Argentina, at the time of the creation of this database, was 37,032,000 people and the life expectancy at birth was of 75.1 years.

# Task 5: Using ORDER BY, LIMIT, what country has the highest life expectancy?

We are now going to go further in our investigation, by selecting only the top value in the “LifeExpectancy” column. To do so, we need to use the function ORDER BY, to show all the values in descending order so to see the highest value first; then, we will use the function LIMIT (called TOP in Microsoft SQL Server) to only show the first, top value:

A screenshot of a computer

Description automatically generated

Andorra is the country with the highest life expectancy in the database, with 83.5 years.

Just out of curiosity, we can check which country has the lowest life expectancy. It is a very easy update to our query, where instead of descending order will show the values in ascending order, from lowest life expectancy to highest:

A screenshot of a computer

Description automatically generated

By doing this, we see that not all the records have a value for the attribute “LifeExpectancy” and instead display a NULL value. In this case, it might be that there is genuinely no information for this attribute. Therefore, we can filter the NULL values out so to only show us the lowest life expectancy where the values are actually displayed by applying a simple condition:

A screenshot of a computer

Description automatically generated

As a result we see that Zambia is now the country with the lowest life expectancy at 37.2 years.

The condition “WHERE LifeExpectancy IS NOT NULL” is asking to only show us attributes that have values. It is crucial to understand that NULL is not equivalent to zero, but means that for a record, that value is unknown, not applicable or not provided. It is the absence of a value.

If we want to dig deeper, we can look which countries have a NULL value for LifeExpectancy. In this case, we can also show what’s the population to see if there is any relationship between the data:

A screenshot of a computer

Description automatically generated

As we can see, the lack of values for life expectancy might have to do with the very low population and most likely absence of records.

# Task 6: Select 25 cities around the world that start with the letter 'F' in a single SQL query.

For this task, first, we are going to select the column “Name” from the “city” table.

Second, we are going to only show cities that start with the letter “F”. To do so:

* We will have to use the LIKE operator instead of the “=“ sign.
* This is because the LIKE operator is used when we are looking for a specific pattern in a column with a WHERE clause.
* We will use the “%” sign as a wildcard that can represent zero, one or multiple characters.

Finally, we are going to use LIMIT to only show 25 records:

A screenshot of a computer

Description automatically generated

In the condition “WHERE Name LIKE “F%”, we are asking to show all values that contain the character “F”, after which, zero, one ore multiple characters can follow.

As we can see below, only 25 rows are shown.



# Task 7: Create a SQL statement to display columns “Id”, “Name”, “Population” from the “city” table and limit results to first 10 rows only.

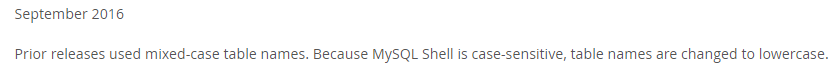
To complete this task, we need to use the SELECT statement, then list the name of the columns we want to retrieve from the database. In this case, they are the “Id”, “Name” and “Population” attributes. We will use the FROM clause to specify the “city” table, then we will use the LIMIT clause to specify the number of instances to return, in our case only 10. To be sure we are using the right column names, we will take a look at the Columns folder in the SCHEMAS window:

A screenshot of a computer

Description automatically generated

As we can see, the “ID” column does not appear as “Id” like the task displayed. In this occasion, it wouldn’t have been an issue because MySQL is not case sensitive by default on Windows (which is our OS). However, as MySQL is case sensitive by default on Unix and macOS systems, we would have had to use the correct case.

Other tools that can be used to interact with MySQL databases may also have different requirements. In the Legal Information document of this database it is stated that:



A screenshot of a computer

Description automatically generated



A screenshot of a computer

Description automatically generatedAgain, as we are using MySQL Workbench, we don’t have to worry about this case sensitivity, and as we can see below, the result is the exact same.



# Task 8: Create a SQL statement to find only those cities from city table whose population is larger than 2,000,000.

To only show cities that have a population above 2 million, we will:

* SELECT the columns we want to see, in this case “Name” and “Population”.
* Use FROM to select the “city” from which to retrieve the records.
* Use WHERE to select only cities that meet the population requirement.
* Create the requirement, which will be above (“>”) 2 million.

A screenshot of a computer

Description automatically generated 

As we can see, the list is quite long, telling us that there are 92 cities that have a population above 2 million. By using the ORDER BY keyword (or clicking on the “Population” cell above), we can sort and see only the top cities:

A screenshot of a computer

Description automatically generated

Another option could have been using LIMIT as shown before.

# Task 9 (Optional): Create a SQL statement to find all city names from city table whose name begins with “Be” prefix.

In this task, we are going to use the wildcard character “%” to only return values that start with “Be” in the “Names” column, in the “city” table.

A screenshot of a computer

Description automatically generated

Again, we are asking to show only values whise name begins with “Be”, and whose next character can be zero, one or multiple. The results is a list of 51 names as we can see below.



A screenshot of a computer

Description automatically generatedAnother wildcard that we could use is “\_”, which represents a single character and can be very useful in some situations. For example, we might want to find how many of these 51 cities have **exactly** 8 characters in their name:

By doing this, we narrowed our list to only 9 cities!

# Task 10 (Optional): Create a SQL statement to find only those cities from city table whose population is between 500,000 - 1,000,000.

To complete this task, we will:

* SELECT the columns “Name” and “Population” FROM the “city” table.
* Create a condition using WHERE.
* Use the BETWEEN operator to only select the range.
* Use the AND operator to separate the begin and end values.

A screenshot of a computer

Description automatically generated

The query returns 303 rows, telling us that there are 303 cities whose population is between 500,000 (included) and 1 million (included).

# Task 11 (Optional): Create a SQL statement to find a city with the lowest population in the city table.

To complete this task, we need to SELECT the “Name” and “Population” columns FROM the “city” table, then order them by population in an ascending order, finally limiting the results to only show the first one:

A screenshot of a computer

Description automatically generated

As we can see, Adamstown, with a population of 42, is the least populated country in the world (where values are not NULL obviously). But this doesn’t give us much information and we can do much better than this.

First, we want to see more than just the city name. We want to see the country name, the continent, the region and the surface area. All of this attributes are contained in another table; therefore we need to create a JOIN:

A screenshot of a computer

Description automatically generatedTo create a JOIN, we need to have two tables that have an attribute in common. On this common factor, we can create the JOIN.

To do so, we need to specify where this link is happening; in our case, the common element is the country code attribute, which both of the “country” and “city” table include.

We have also included aliases to avoid mix-ups with column names (in this case both city and country have a “Name” column).

The query above leaves us with a doubt: what would be the population density in a country such small?

To answer this question, we will need to perform a calculation; fortunately, in MySQL this task is quite simple, we are going to divide the population by the area and show this new attribute as “Density”:

A screenshot of a computer

Description automatically generated

With this query, we will have a more detailed description of the least populated city in the entire database.

# Task 12 (Optional): Create a SQL statement to show the population of Switzerland and all the languages spoken there.

To complete this task, we will need to use the JOIN clause as we did before. In this case, we will need the “country” table, which we need to retrieve the information relating to the country name and population; we will need to link this to the “countrylanguage” table that contains the data relating to the language and has a very interesting column named “Percentage”.

A screenshot of a computer

Description automatically generated

This time, we used aliases for the tables, so to make the query a bit shorter and less complicated. The column “IsOfficial” tell that it is True that all the four languages listed here are considered official in Switzerland.

The population column, however, seems to be redundant as it displays the value repeated multiple times. To make it more interesting and meaningful, we will create a calculation based on the “Percentage” column to show how many people in Switzerland speak each of those 4 languages.

As the percentages don’t add up to 100% (some minority languages have been left out), the total population will not be reflected on the speakers data.

A screenshot of a computer

Description automatically generated

As we can see, to find the number of speakers based on the percentages, we had to multiply the population by the percentage, which in turn needed to be set as a decimal, therefore divided by 100.

We used the ROUND function to show the results as whole numbers, as we are describing number of people, which can’t be fractions!

As we stated before, the values in the “NumberOfSpeakers” column don’t add up to 100, as the percentages add up only to 91.1.

Although there is a way around it, it will require adding a new row to the query. To do so:

* We will SELECT “Name” and “Population”, leaving the original values
* For the attribute “Language”, we will set it to “Others”
* For the “IsOfficial”, we will set it to false as the other languages are not official
* For the “Percentage”, we will create a calculation to show the result of all the other percentages taken away from the total (100)
* For “NumberOfSpeakers”, we will multiply the population by the percentage (as a decimal)
* We will then need to use the UNION operator to combine the results of the original query and the new row selected.

The result will be:

A screenshot of a computer

Description automatically generatedThis is the original query that we have left unchanged.

Here, using UNION, we combine the result sets for the two SELECT statements.



This part of the query adds a new row for “Others” .

We are keeping the JOIN as we need it to retrieve data from the “country” and “countrylanguage” tables.



The final result shows us all languages spoken in Switzerland, including the number of people for each language. Unfortunately, as we don’t have this information in the database, we cannot break down the “Others” value in a more detailed set of values. At, least, all percentages add to 100 and so do the number of speakers compared to the total population.

# Extension: using Stored Procedures to run queries with a simple click

As we ran several queries onto our editor, we notice that, if we want to go back to double check each task, we need to scroll up and down quite a bit. But there is a function that can help us with that, the Stored Procedure.

A stored procedure is a query that we can save and have on the side, under the SCHEMAS menu, so that we can select it and run it with a simple click. This can save a lot of time by adding a bit of automation into our work.

To create a stored procedure we can simply locate “Stored Procedures” under SCHEMAS on the left side of the screen, right click and select “Create Stored Procedure…”:

A screenshot of a computer screen

Description automatically generated

In the new window, we will need to simply copy paste the query that we want to store. As an example, we can use the last one that we made. We will paste it between BEGIN and END and change the name after PROCEDURE, then click Apply:

A screenshot of a computer

Description automatically generated



After reviewing the script, we will select Apply:

A screenshot of a computer

Description automatically generated



We have created our first Stored Procedure under the name Task12\_Ext2. To run it, simply press the lightning symbol on the right end side:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated A new tab will appear with our query and the results:



Alternatively, we can simply run our Stored Procedure manually by using the CALL command.

# Task 13: Creating an EER Diagram

An Enhanced Entity-Relationship diagram, also called EER or known as ERD, is a visual representation of the data model for a database. To create an EER Diagram we can go to the “Reverse Engineering” option under the Database menu; alternatively, we can simply press CTRL+R:

A screenshot of a computer

Description automatically generated



On this menu, select Next, then insert your password and select Next again. Another menu will appear where we will select the “world” database.

A screenshot of a computer

Description automatically generated



We will then select the tables to import, after which we will execute and complete the process:

A screenshot of a computer program

Description automatically generated



We will end up with our EER diagram:

A screenshot of a computer

Description automatically generated

This is a visual representation of the database that we worked on. We can see all the relationships that occur between them, all the data types and names of attributes, primary keys and foreign keys.

# Task 14: Use the EER diagram to answer the following:

## Identify the primary key in country table.

The primary key in the “country” table is the attribute “Code”. It is a unique identifier for each country in the database. Its uniqueness is crucial to ensure that each country is easily distinguishable and can be referenced without ambiguity. In this case, the primary key is made of letters combined in a unique way.

A close up of a blue box

Description automatically generated

In MySQL Workbench it is represented as a “golden key” to make its visual intuitive.

## Identify the primary key in city table.

The primary key in the “city” table is the column “ID”. Again, it serves as a unique identifier for each city in the database. In this case, the primary key is a unique series of integer numbers.

A close-up of a blue box

Description automatically generated

Again, we see the “golden key” to represent the “ID” column as primary key.

## Identify the primary key in countrylanguage table.

The primary key in the “countrylanguage“ table is the combination of two columns, “CountryCode” and “Language”. These two attributes together for a so called "composite key”, which helps create a unique value for each combination of country and language in the database.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

As we can see, when selecting “PRIMARY” under indexes, both “CountryCode” and “Language” are highlighted as composite primary key.

## Identify the foreign key in city table.

A screenshot of a computer

Description automatically generatedThe foreign key in the “city” table is “CountryCode”. The dashed line represents the fact that the primary key “Code” (in the “country” table) becomes a non-key attribute in the ‘child’ entity. In fact, “CountryCode” in the city table is not used as a key attribute (which is the “ID” PK). This is called a **Non-Identifying** Relationship.

Non-Identifying Relationship (many-to-one)

## 

Primary Key

Foreign Key

The relationship is many-to-one (or one-to-many depending on which table are we analysing), as one country can have multiple city, but for each city there can be one country, and one only.

## Identify the foreign key in countrylanguage table.

As we have seen before in task C, the “countrylanguage” table uses a composite key as its primary key. This composite key is formed by the combination of two attributes, “CountryCode” and “Language”.

The “CountryCode” column, represented by a “red key” symbol , is a foreign key. It plays a dual role by serving both as a foreign key and as part of the composite primary key. As a foreign key, it creates the relationship between the “country” and the “countrylanguage” tables through the country code field (respectively named “Code” and “CountryCode”).

A screenshot of a computer

Description automatically generated

When we hover over the relationship between the two tables, the common elements are highlighted.

In the context of a one-to-many relationship, we can see that each row in the “country” table is linked to multiple rows in the “countrylanguage” table, while each row in the “countrylanguage” table can be linked to only one row in the “country” table.

This helps us identify that the foreign key in the “countrylanguage” table is the attribute responsible for establishing the link between these tables, allowing one-to-many relationships to be established in the database. In this case, the foreign key in the “countrylanguage” table is the “CountryCode” attribute.

# Reflections

This assignment helped me massively to understand the theory behind the databases and SQL; it also pushed me to stretch my knowledge and put into practice what I have learned throughout the course and in my self-study time. It was challenging at times, but overcoming those challenges contributed to solidify my knowledge.

One of the most enjoyable aspects of this assignment was diving into the theory behind relational databases and comprehending how they function. I found fascinating similarities between my archival science background and relational databases, particularly concerning the importance of “data integrity”.

In archival science, safeguarding “data integrity” is paramount. This concept ensures that documents and records, some of which may be centuries old, remain unaltered and authentic, therefore easy to access within massive archives. Similarly, in the world of databases, “data integrity” plays a crucial role in making information retrieval smoother and more efficient. It guarantees that data is correctly stored, organised and uncorrupted, resulting in a more coherent and reliable database system.

Regarding the MySQL Workbench tasks, it was a great experience to put my hands in a real database and I enjoyed retrieving the information to analyse and understand better the data.

Creating more complex queries, such as for “Task 12” and exploring advanced features such as the Stored Procedures in my “Extension” task was extremely rewarding.

Trying out something new forced me to face my technical limitations, such as for the two tasks just citated. Fortunately, patience and determination helped me figure out that even long queries can be broken down into smaller, simpler steps. It only requires taking enough time to really understand deeply the reasons behind each action.

Overall, this was a fantastic experience, not only expanded my technical skills but also reinforced my mental strength. It pushed me to overcome hurdles and this helped me learning more than anything else. Finding links between archives and the theory behind databases had also a great impact in strengthening my motivation to understand how databases work in detail and how valuable they are.

In the future, I will definitely go back to this assignment to remind myself all the lessons that I have learned during this experience.