MySQL Assignment

Balbir Lehto

DATA Cohort 1 DEFW4

24th October 2023

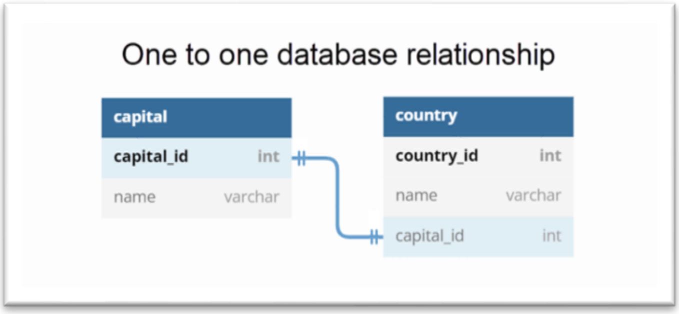
Table of Contents

TASK 1: LIST THE DIFFERENT TYPES OF RELATIONSHIPS IN RELATIONAL DATABASES AND PROVIDE EXAMPLES.

*DEFINITION: A **relational database** is a type of database that focuses on the relation between stored data elements. It allows users to establish links between different sets of data within the database and use these links to manage and reference related data.*

ONE – To – ONE (1:1)

A one-to-one relational database contains data in a form that a single record in one table will relate to only one record in the other table. The diagram below shows this type of relationship:



The above diagram illustrates the concept of one-to-one relationships in databases, using the example of countries and their capital cities. This relationship is typically represented within a single table, as creating a second table for a single value is unnecessary.

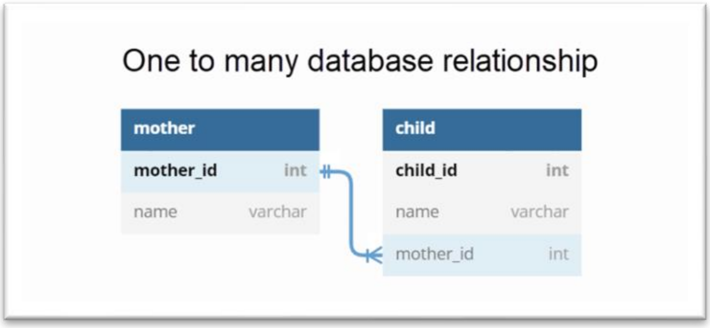
However, there are situations where a one-to-one relational database can be beneficial. It can be thought of as vertically partitioning a table. This approach can save time, memory, and processing power if the data field isn't frequently used, as it doesn't need to be cached.

Also, if the data isn't fully populated or poses a security risk, separating it might be a wise decision. Lastly, for usability purposes, if a table has numerous columns (100), splitting it could make it more manageable.

ONE to ONE RELATIONAL DATABASE	
Advantages	Disadvantages
Data Streamling: Utilizing one-to-one relationships can aid in data normalization. This is achieved by dividing a table with numerous columns into two distinct tables with fewer columns, thereby minimizing data duplication and enhancing data uniformity.	Complexity: One-to-one relationships can introduce complexity into the database schema, making it more challenging to manage.
Security Measures: One-to-one relationships can be used to house sensitive or classified information in a separate table. This table can then be safeguarded with stricter access controls for better security.	Data Retrieval: One-to-one relationships require the use of joins to fetch related data, which could be slower than querying a single table.
Performance Enhancement: One-to-one relationships can boost query performance. By reducing the column count in a table, indexing becomes more efficient, leading to improved performance.	Storage Implications: One-to-one relationships can escalate storage needs by generating additional tables.

ONE – To – MANY (1: N)

A one-to-many entry in a database, has one entry on one side and many entries in the other side of the link. As can be seen in the illustration below, the relationship between mother and child is a one-to-many one. A mother can have many children, but a child only has one mother.



In the example above the mother_id is a primary key in the mother table, but a foreign key in the child table (where it is no longer unique, as many children can be linked to this id). This relationship is mandatory at both ends, to be a mother there must be a child, child will always have a mother.

ONE To MANY RELATIONSHIP	
Advantages	Disadvantages
Data Streamlining: One-to-many relationships can facilitate data normalization by dividing a table into two distinct tables with fewer columns. This can aid in minimizing data duplication and enhancing data uniformity.	Data Access: One-to-many relationships necessitate the use of joins to fetch related data, which could be slower than querying a single table.
Ease of Use: One-to-many relationships are straightforward to comprehend and implement, making them an excellent choice for beginners.	Storage Considerations: One-to-many relationships can increase storage needs by generating additional tables.
Performance Enhancement: One-to-many relationships can boost query performance by reducing the column count in a table, leading to more efficient indexing.	Complexity: One-to-many relationships can introduce complexity into the database schema, making it more challenging to manage.

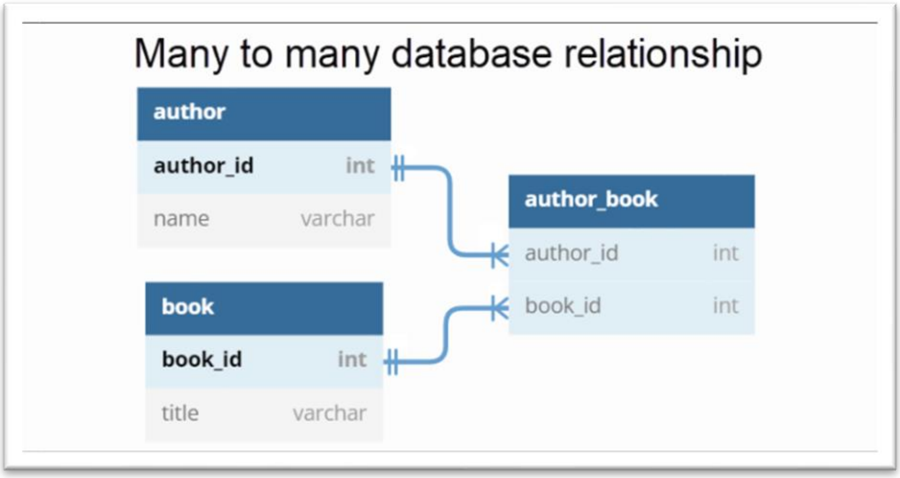
MANY-To-MANY (N: N)

Many-to-many relationships in a database have **multiple entries on both ends** of the relationship. Since many entries can exist on both ends, a common solution is to create a **bridging** table with foreign keys from both tables.

Many-to-many relationships are commonly used with relational databases and in web technologies, such as ecommerce websites.

A many-to-many relationship exists between books and authors. For example, a single book can have many authors and a single author can have numerous books.

If there is a table with books and another with authors, the best way to make a relationship between the two is through a **new table**. The new table has foreign keys from both parent tables, creating a many-to-many relationship.



Performing different types of JOIN queries fetches data from both tables efficiently while protecting the original tables from redundancies.

MANY to MANY Relationship	
Advantages	Disadvantages
Data Organization: Many-to-many relationships can aid in data normalization by splitting a table with many columns into two separate tables with fewer columns. This can help reduce data redundancy and improve data consistency.	Complexity: Many-to-many relationships can increase the complexity of the database schema, making it more difficult to manage.
Comprehensive Support: Relational databases and SQL provide extensive support for many-to-many relationships.	Data Retrieval: These relationships require joins to access related data, which could be slower than querying a single table
User Accessibility: Relationships improve usability for end users and help uphold data integrity.	Data Uncertainty and Replication: These relationships can lead to uncertainty and duplications in data that may result in incorrect query statements.

TASK 2: NORMALISATION – WHAT IS IT AND WHY IS IT IMPORTANT TO DATABSE DEVELOPMENT

Data normalization is a method of structuring data in a database to minimize redundancy and enhance data consistency. The steps to normalize data for a database:

1. **Entity Identification**: Recognise the entities that you wish to store in the database. An entity is an object that holds significance to your application, such as customers, orders, or products.
2. **Table Creation for Each Entity**: Create a table for each entity and define the columns that you wish to store in the table. Each column should represent a single attribute of the entity.
3. **Primary Key Definition**: Assign a primary key for each table. A primary key is a column or set of columns that uniquely identifies each record in the table.
4. **Relationship Identification**: Determine the relationships between the tables. Relationships are formed through primary keys and foreign keys.
5. **Table Normalization**: Normalize the tables by applying normalization rules to eliminate redundancy and enhance data consistency. There are several normal forms, such as 1NF to 5NF (1NF to 3NF are illustrated with examples later)
6. **View Creation**: Construct views to simplify complex queries and enhance performance.
7. **Database Testing**: Examine the database to ensure that it functions as expected and meets your requirements.

Normalisation is a process that optimizes database structure by reducing data redundancy and enhancing data integrity. It is a set of rules and guidelines that help organize data efficiently and prevent common data anomalies like update anomalies, insertion anomalies, and deletion anomalies. Normalization is typically divided into several stages.

Normalization is typically divided into several levels, referred to as normal forms. The typical normal forms are:

1. **First Normal Form (1NF)**: Ensures that each column in a table contains atomic, single(indivisible) values. There should be no repeating groups, and each column should have a unique name.
2. **Second Normal Form (2NF)**: Building on 1NF, 2NF eliminates partial dependencies. A table is in 2NF if it's in 1NF and all non-key attributes are functionally dependent on the entire primary key.
3. **Third Normal Form (3NF)**: Building on 2NF, 3NF eliminates transitive dependencies. A table is in 3NF if it's in 2NF and all non-key attributes are functionally dependent on the primary key, but not on other non-key attributes.
4. **Boyce-Codd Normal Form (BCNF)**: A stricter version of 3NF, BCNF ensures that every non-trivial functional dependency is a super key. This means that no partial dependencies or transitive dependencies are allowed.
5. **Fourth Normal Form (4NF)**: 4NF deals with multi-valued dependencies, where an attribute depends on another attribute but is not a function of the primary key.
6. **Fifth Normal Form (5NF)** (or Project-Join Normal Form, PJNF**)**: These forms deal with cases where a table is in 4NF, but there are join dependencies that can be further optimized.

1NF (First Normal Form): A table is in 1NF if it contains atomic values, meaning each cell holds only one value. There are no duplicate rows. For example:

This Table not in 1NF because it has multiple values in each cell of the subject field.

StudentID	name	subject
1	Sally	Maths, Science
2	Graham	Maths, History

This table is now in 1NF as each cell only holds one value

StudentID	name	subject
1	Sally	Maths
1	Sally	Science
2	Graham	Maths
2	Graham	History

<u>2NF (Second Normal Form):</u> In the table below, "Name" is dependent on "StudentID", but "Teacher" is dependent on "Subject", not "StudentID". For example:

StudentID	name	subject	teacher
1	Sally	Maths	Mr Blue
1	Sally	Science	Ms Green
2	Graham	Maths	Mr Blue
2	Graham	History	Mr Grey

A table is in 2NF if it is in 1NF, and all non-key attributes are fully functionally dependent on the primary key. To convert it to 2NF, we would move the “Teacher” column to a separate table that relates subjects to teachers.

To Normalise this to 2NF we need to split the table into two tables:

These 2 tables now are in 2NF, each non-key attribute is dependent on a primary key

StudentID	name	subject
1	Sally	Maths
1	Sally	Science
2	Graham	English
2	Graham	Maths

subject	teacher
Maths	Mr. Blue
Science	Ms. Green
History	Mr. Grey

<u>3NF (Third Normal Form):</u> A table is in 3NF if it satisfies the rules for 2NF, and no transitive dependency exists. An example of a transitive dependency would be:

employeeID	name	departmentID	departmentName
1	Mr Blue	101	Maths
2	Ms Green	102	Science
3	Ms Pink	101	Maths

In this table, the “DepartmentName” is dependent on the “DepartmentID”, not directly on the “EmployeeID”. This is an example of a transitive dependency because “DepartmentName” is not directly dependent on the primary key (“EmployeeID”), but rather on another column in the table.

To convert this table to 3NF, we move the “DepartmentName” to a separate table that relates departments to their IDs.

Employee Table		
employeeID	name	departmentID
1	Mr Blue	101
2	Ms Green	102
3	Ms Pink	101

Department Table	
DepartmentID	Department name
101	Maths
102	Science

Each non-key attribute in each table (Name and DepartmentID in the Employee table, and DepartmentName in the Department table) is dependent on the primary key. Therefore, both these tables are in 3NF.

In summary, normalisation is very important to database development because it helps to reduce duplicated data, which reduces storage overhead. Having only one piece of data in one place ensures that if you need to update or delete it then it can be done easily and accurately. Rather than having to go to many places to make these changes. A well defined and organized database can improve query performance as well as the speed and simplifying the query process. This is especially crucial to databases with very large sets of data.

MYSQL TASKS:

TASK 1: Using count, get the number of cities in the USA

```
#Task 1: Using count, get the number of cities in the USA
USE world
SELECT COUNT(*) AS Number_of_Cities_in_USA FROM City
    WHERE CountryCode = 'USA';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Number_of_Cities_in_USA
274

← Count * is used on the city table in the USA. I have Used 'AS' to make the label on the result table more meaningful.

COUNT * is used preferably to say COUNT (CountryCode) because a database can often count rows by accessing an index which is faster than accessing the table-column or just return a row count. Accessing the table-column is slower because it will first check for nulls and only return a count of non-null values.

In this example there was no difference as CountryCode is defined as NOT NULL

As an extension to this task, Count has been used on the table 'city' and grouped by district to show the number of cities in each district.

```
# EXTENSION to Task 1: Using COUNT and GROUP BY to get the number of Cities per
# District in the USA represented in the data, then ordered by Descending numbe
# of cities in each District
SELECT COUNT(*) AS Number_of_cities, District, CountryCode FROM city
    WHERE CountryCode = 'USA'
    GROUP by District
    ORDER BY Number_of_cities DESC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Number_of_cities	District	CountryCode
68	California	USA
26	Texas	USA
15	Florida	USA
10	Virginia	USA
9	Arizona	USA
9	Colorado	USA
8	Illinois	USA
8	Michigan	USA
8	Massachusetts	USA
7	North Carolina	USA
6	New York	USA
6	Ohio	USA

Task 2: Find out what the population and life expectancy for people in Argentina (ARG) is

```
# Task 2: Find out what the population and
# life expectancy for people in Argentina (ARG) is
SELECT NAME, Population, LifeExpectancy FROM country
WHERE Code ='ARG';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Name	Population	LifeExpectancy
Argentina	37032000	75.1

Here a simple SELECT statement is used to return Population and LifeExpectancy from the country table (which holds the Population data for the whole country). The WHERE command allows us to limit the results so that the only results returned are those where the Code value matches the string 'ARG'. In this case there was only one entry for Argentina.

TASK 3: Using ORDER BY, LIMIT, what country has the highest life expectancy?

```
# TASK 3: Using ORDER BY, LIMIT, what country has the
# highest life expectancy?
SELECT Name, LifeExpectancy FROM country
ORDER BY LifeExpectancy DESC
LIMIT 1;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Name	LifeExpectancy
Andorra	83.5

This query returns Name and LifeExpectancy from the country table but uses a descending sort (DESC) to show the Highest life expectancy at the top of the results returned and then LIMIT is used to only return a certain number of results, in this example only 1 result is returned.

TASK 4: Select 25 cities around the world that start with the letter 'F' in a single SQL query.

```
# TASK 4: Select 25 cities around the world that start
# with the letter 'F' in a single SQL query.
SELECT Name, CountryCode FROM city
WHERE Name LIKE 'F%'
LIMIT 25;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Name	CountryCode
Fagatogo	ASM
Florencio Varela	ARG
Formosa	ARG
Francistown	BWA
Fortaleza	BRA
Feira de Santana	BRA
Franca	BRA
Florianópolis	BRA
Foz do Iguaçu	BRA
Ferraz de Vasconcelos	BRA
Francisco Morato	BRA
Franco da Rocha	BRA
Fuenlabrada	ESP
Faridabad	IND
Firozabad	IND
Farrukhabad-cum-Fat...	IND
Faizabad	IND
Fatehpur	IND
Firenze	ITA
Foggia	ITA
Ferrara	ITA
Forlì	ITA
Fukuoka	JPN
Funabashi	JPN
Fukuyama	JPN

Here the LIKE operator is used when you only want to specify a subset of letter(s) from that string to find the values needed. LIKE can be used in conjunction with '%' symbol to allow any characters to follow 'F'

TASK 5: Create a SQL statement to display columns Id, Name, Population from the city table and limit results to first 10 rows only.

```
# TASK 5: Create a SQL statement to display columns Id,
# Name, Population from the city table and limit results
# to first 10 rows only.
SELECT ID, Name, Population FROM city
LIMIT 10;
```

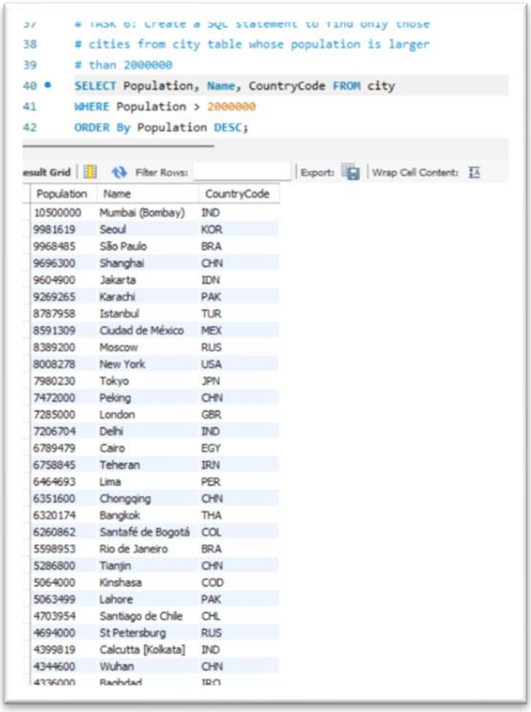
Result Grid | Filter Rows: | Edit: | Export/Import:

ID	Name	Population
1	Kabul	1780000
2	Qandahar	237500
3	Herat	186800
4	Mazar-e-Sharif	127800
5	Amsterdam	731200
6	Rotterdam	593321
7	Haag	440900
8	Utrecht	234323
9	Eindhoven	201843
10	Tilburg	193238
NULL	NULL	NULL

Here LIMIT restricts the results returned to the first 10 rows,

The rows that are returned here can be altered by ordering the results in a different way, here they have been ordered by ID by MYSQL (which tends to order by the first table column values regardless of whether the first column in the table is returned as part of the SQL statement). The only way to guarantee the table is ordered by a certain field is to use ORDER BY.

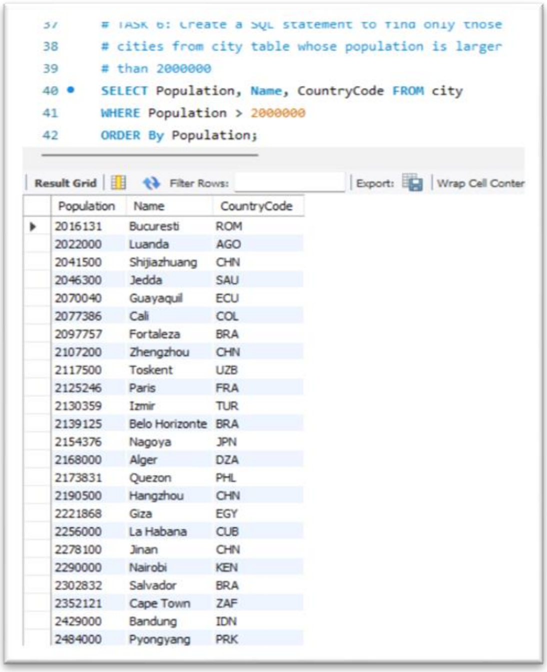
TASK 6: Create a SQL statement to find only those cities from city table whose population is larger than 2000000



```sql
# TASK 6: Create a SQL statement to find only those
# cities from city table whose population is larger
# than 2000000
SELECT Population, Name, CountryCode FROM city
WHERE Population > 2000000
ORDER By Population DESC;
```

Population	Name	CountryCode
10500000	Mumbai (Bombay)	IND
9981619	Seoul	KOR
9968485	São Paulo	BRA
9696300	Shanghai	CHN
9604900	Jakarta	IDN
9269265	Karachi	PAK
8787958	Istanbul	TUR
8591309	Ciudad de México	MEX
8389200	Moscow	RUS
8008278	New York	USA
7980230	Tokyo	JPN
7472000	Peking	CHN
7285000	London	GBR
7206704	Delhi	IND
6789479	Cairo	EGY
6758845	Teheran	IRN
6464693	Lima	PER
6351600	Chongqing	CHN
6320174	Bangkok	THA
6260862	Santafé de Bogotá	COL
5598953	Rio de Janeiro	BRA
5286800	Tianjin	CHN
5064000	Kinshasa	COD
5063499	Lahore	PAK
4703954	Santiago de Chile	CHL
4694000	St Petersburg	RUS
4399819	Calcutta [Kolkata]	IND
4344600	Wuhan	CHN
4336000	Baghdad	IRQ

This table shows the highest population at the top as it has been sorted DESC.

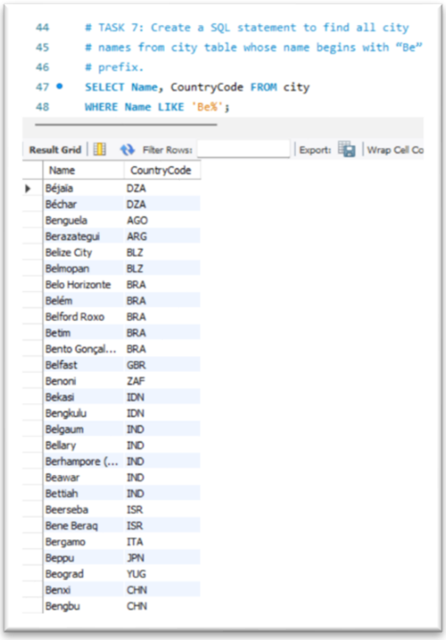


```sql
# TASK 6: Create a SQL statement to find only those
# cities from city table whose population is larger
# than 2000000
SELECT Population, Name, CountryCode FROM city
WHERE Population > 2000000
ORDER By Population;
```

Population	Name	CountryCode
2016131	Bucuresti	ROM
2022000	Luanda	AGO
2041500	Shijiazhuang	CHN
2046300	Jedda	SAU
2070040	Guayaquil	ECU
2077386	Cali	COL
2097757	Fortaleza	BRA
2107200	Zhengzhou	CHN
2117500	Toskent	UZB
2125246	Paris	FRA
2130359	Izmir	TUR
2139125	Belo Horizonte	BRA
2154376	Nagoya	JPN
2168000	Alger	DZA
2173831	Quezon	PHL
2190500	Hangzhou	CHN
2221868	Giza	EGY
2256000	La Habana	CUB
2278100	Jinan	CHN
2290000	Nairobi	KEN
2302832	Salvador	BRA
2352121	Cape Town	ZAF
2429000	Bandung	IDN
2484000	Pyongyang	PRK

This table shows the lowest population at the top as it has been sorted ASC by default

TASK 7: Create a SQL statement to find all city names from city table whose name begins with “Be” prefix.



```sql
# TASK 7: Create a SQL statement to find all city
# names from city table whose name begins with "Be"
# prefix.
SELECT Name, CountryCode FROM city
WHERE Name LIKE 'Be%';
```

Name	CountryCode
Béjaïa	DZA
Béchar	DZA
Benguela	AGO
Berazategui	ARG
Belize City	BLZ
Belmopan	BLZ
Belo Horizonte	BRA
Belém	BRA
Belford Roxo	BRA
Betim	BRA
Bento Gonçal...	BRA
Belfast	GBR
Benoni	ZAF
Bekasi	IDN
Bengkulu	IDN
Belgaum	IND
Bellary	IND
Berhampore (...	IND
Beawar	IND
Bettiah	IND
Beerseba	ISR
Bene Beraq	ISR
Bergamo	ITA
Beppu	JPN
Beograd	YUG
Benxi	CHN
Bengbu	CHN

Here the LIKE command looks for anything starting ‘Be’ followed by any character which is denoted by the % symbol.

TASK 8: Create a SQL statement to find only those cities from city table whose population is between 500000-1000000.

```sql
# TASK 8: Create a SQL statement to find only those
# cities from city table whose population is between
# 500000-1000000.
SELECT Name, Population FROM city
WHERE Population BETWEEN 500000 AND 1000000;
```

Name	Population
Amsterdam	731200
Rotterdam	593321
Oran	609823
Dubai	669181
Rosario	907718
Lomas de Zamora	622013
Quilmes	559249
Almirante Brown	538918
La Plata	521936
Mar del Plata	512880
Adelaide	978100
Khulna	663340
Cotonou	536827
Santa Cruz de la...	935361
La Paz	758141
El Alto	534466
Campinas	950043
São Gonçalo	869254
Nova Iguaçu	862225
São Luís	837588
Maceió	786288
Duque de Caxias	746758
São Bernardo do...	723132
Teresina	691942

Here the BETWEEN operator returns all values inclusive of the boundary values in the conditional statement.

TASK 9: Create a SQL statement to find a city with the lowest population in the city table

Below are two different queries to get the same result:

```sql
# TASK 9: Create a SQL statement to find a city
# with the lowest population in the city table
SELECT Name, Population AS Lowest_Population, CountryCode FROM city
ORDER BY Population ASC
LIMIT 1;
```

Name	Lowest_Population	CountryCode
Adamstown	42	PCN

This query orders by the population (from lowest to highest) but only returns one value due to the LIMIT operator. This query is faster than the alternative on the right if the data is already sorted and the data set is very big, for small data sets the difference in time taken is negligible.

```sql
# Alternative way to get the same result of lowest population from city table
SELECT Name, Population
FROM city
WHERE Population = (SELECT MIN(Population) FROM city);
```

Name	Population
Adamstown	42

In the second SELECT statement we assign to Population a value that is the minimum in the table, then the first SELECT statement returns that as a result. This has the advantage that if there were more than one country with the same minimum Population value, you would see them all. This method also does not need to sort the data and is more efficient for a large data set.

TASK 10: Create a SQL statement to show the population of Switzerland and all the languages spoken there.

Here a table join is made to access data from both tables.

The GROUP_CONCAT allows the languages all to be shown in one row. The second example does not use GROUP_CONCAT but shows the languages in separate table rows, which allows clear display of the number of people speaking that language by population.

```sql
# TASK 10: Create a SQL statement to show the population of
# Switzerland and all the languages spoken there.
SELECT co.Name, cl.CountryCode, GROUP_CONCAT(DISTINCT Language SEPARATOR ', ')
    AS Languages_Spoken_in_Switzerland, co.Population AS Total_Population
    FROM country AS co
INNER JOIN countrylanguage AS cl
    ON co.Code=cl.CountryCode
    WHERE cl.CountryCode = 'CHE';
```

Name	CountryCode	Languages_Spoken_in_Switzerland	Total_Population
Switzerland	CHE	French, German, Italian, Romansh	7160400

```sql
#Alternative way to show languages spoken in Switzerland:
 SELECT co.Name,  Language
    AS Languages_Spoken_in_Switzerland, ROUND(co.Population*(Percentage/100)) AS Population_speaking_language
    FROM country AS co
INNER JOIN countrylanguage AS cl
    ON co.Code=cl.CountryCode
    WHERE cl.CountryCode = 'CHE';
```

Name	Languages_Spoken_in_Switzerland	Population_speaking_language
Switzerland	French	1374797
Switzerland	German	4554014
Switzerland	Italian	551351
Switzerland	Romansh	42962

OTHER EXTENSION (FURTHER LEARNING) TASKS WITH DIFFERENT SQL QUERIES:

This query shows Language and the countries that

Language is spoken in. A table join allows data from two tables and GROUP_CONCAT allows countries to be showed in one row.

```
# EXT TASK: Create a SQL statement to show Languages
# and what countries they are spoken in.
SELECT  cl.Language, GROUP_CONCAT(DISTINCT co.Name SEPARATOR ', ')
AS Countries_speaking_this_language FROM country AS co
INNER JOIN countrylanguage AS cl
    ON co.Code=cl.CountryCode
GROUP BY cl.Language;
```

Language	Countries_speaking_this_language
[South]Mande	Côte d'Ivoire
Abhyasi	Georgia
Acholi	Uganda
Adja	Benin
Afar	Djibouti, Eritrea
Afrikaans	Namibia, South Africa
Aimará	Bolivia, Chile, Peru
Ainu	Japan
Aizo	Benin
Akan	Côte d'Ivoire, Ghana
Albaniana	Albania, Italy, Macedonia, Yugosl...
Ambo	Angola
Amhara	Ethiopia
Ami	Taiwan

This query uses GROUP BY country Name to show the Official languages spoken in that Country along with the percentage representation of that language in the country. The COUNT gives the number of official languages due to the boolean condition in the WHERE statement as T is True for langauage being official.

```
#EXT TASK:The SQL below is a query to find out how many Official languages are spoken
# and their relative percentages in square brackets, as can be seen the  pepercentages
# do no add up to 100 as there are unofficial languages also spoken
# but not represented below due to the IsOfficial = 'T' condition.To see all
# Languages This condition can be disables with a '#'
# For each country in the list. GROUP_CONCAT allows me to show the countries as
# a list all in one row, rather than over many rows. Within that function some
# formatting is done to make the countries and associated percentage more readable.

SELECT Name,
GROUP_CONCAT(' ', Language, ' [', Percentage, '%', ']' ORDER BY Percentage DESC)
AS 'Languages', COUNT( DISTINCT Language) AS Number_of_Official_languages_Spoken FROM world.countrylanguag
join country ON country.Code = countrylanguage.CountryCode
WHERE Percentage>0 AND IsOfficial = 'T'
GROUP BY country.Name
Order BY Number_of_Official_languages_Spoken DESC, Name;
```

Name	Languages	Number_of_Official_languages_Spoken
South Africa	Zulu [22.7%], Xhosa [17.7%], Afrikaans [14.3%], English [8.5%]	4
Switzerland	German [63.6%], French [19.2%], Italian [7.7%], Romansh [0.6%]	4
Belgium	Dutch [59.2%], French [32.6%], German [1.0%]	3
Bolivia	Spanish [87.7%], Ketšua [8.1%], Aimará [3.2%]	3
Luxembourg	Luxembourgish [64.4%], French [4.2%], German [2.3%]	3
Peru	Spanish [79.8%], Ketšua [16.4%], Aimará [2.3%]	3
Singapore	Chinese [77.1%], Malay [14.1%], Tamil [7.4%]	3
Vanuatu	Bislama [56.6%], English [28.3%], French [14.2%]	3
Afghanistan	Pashto [52.4%], Dari [32.1%]	2
American Samoa	Samoan [90.6%], English [3.1%]	2
Belarus	Belorussian [65.6%], Russian [32.0%]	2
Canada	English [60.4%], French [23.4%]	2

This query shows the number of languages spoken in each country

```
#This query shows how many different languages are spoken in each country
SELECT Name, COUNT( DISTINCT Language) AS Number_of_languages_spoken FROM world.countrylanguage
JOIN country
ON country.Code=countrylanguage.CountryCode
GROUP BY Name
ORDER BY Number_of_languages_spoken DESC;
```

Name	Number_of_languages_spoken
Canada	12
China	12
India	12
Russian Federation	12
United States	12
South Africa	11
Tanzania	11
Congo, The Democratic Republic of the	10
Iran	10
Kenya	10
Mozambique	10
Nigeria	10
Philippines	10
Sudan	10
Uganda	10
Angola	9
Indonesia	9
Vietnam	9
Australia	8
Austria	8
Cameroon	8

This query calculates the number of people who speak each language around the world. This is calculated from the percentage of each population speaking that language which is then aggregated, divided by 1000 to give a smaller number with K added to the end to denote' thousands'. Ideally this query should return results in billions, millions or thousands.

```
#This query returns the number of people speaking a certain language around the world
# The number of people speaking that language per country is aggreagated to give a whole world figure.
SELECT Language, CONCAT(ROUND(SUM((Percentage/100)*Population/(1000))), ' K') AS number_speaking_language_around_the_world  FROM country
join countrylanguage ON country.Code = countrylanguage.CountryCode
WHERE Percentage >0
GROUP BY Language
ORDER BY SUM((Percentage/100)*Population) DESC;
```

Language	number_speaking_language_around_the_world
Chinese	1191844 K
Hindi	405633 K
Spanish	355029 K
English	347078 K
Arabic	233839 K
Bengali	209305 K
Portuguese	177595 K
Russian	160808 K
Japanese	126814 K
Punjabi	104025 K
German	92134 K
Javanese	83570 K
Telugu	79066 K
Marathi	75019 K
Korean	72291 K
Vietnamese	70616 K
French	69981 K

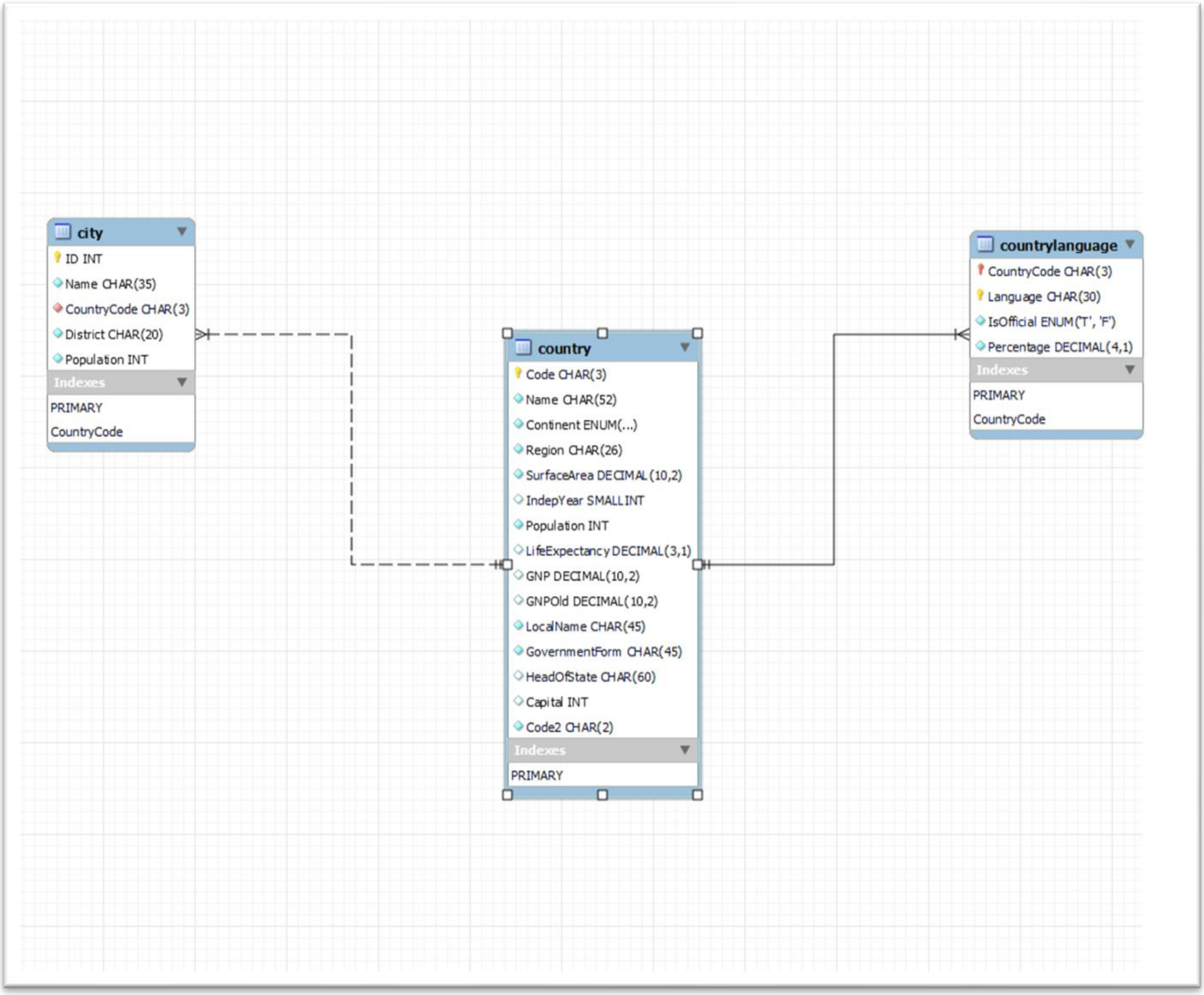
The queries above make use of formatting techniques, concatenation and some simple calculations to show different data, and data from different perspectives such as language or country.

```
# This query shows use of three tables joined just to illustrate that it is the
# same procedure as joining two. Here we have the Country, number of Districts and
# number of Languages
SELECT   co.Name AS CountryName, COUNT(DISTINCT c.District) AS Number_Of_Districts,
    COUNT( DISTINCT cl.Language) AS number_of_languages FROM city AS c
INNER JOIN countrylanguage AS cl
    ON c.CountryCode=cl.CountryCode
INNER JOIN country AS co
    ON co.Code=c.CountryCode
    Group BY co.Name
    ORDER BY co.Name;
```

CountryName	Number_Of_Districts	number_of_languages
Afghanistan	4	5
Albania	1	3
Algeria	18	2
American Samoa	1	3
Andorra	1	4
Angola	4	9
Anguilla	1	1
Antigua and Barbuda	1	2
Argentina	20	3
Armenia	3	2

This Query illustrates 3 tables joined together to show country, district count & language count.

EER DIAGRAM:



Key to the different symbols in the EER diagram above:
In MySQL Workbench, the different coloured keys and symbols next to the fields in a database table have the following meanings:

This symbol indicates that the field is only a primary key

A blue lined filled diamond indicates that it's a NOT NULL simple attribute

A blue lined not filled diamond indicates that it's a simple attribute which can be NULL

A red colored filled diamond indicates that it's a NOT NULL foreign key

A red colored key indicates that it's a primary key which is also a foreign key

Task 14:

Identify the primary key in country table: The primary key in the 'country' table is **'Code'** which holds unique values for each row in its own table.

Identify the primary key in city table: The primary key in the 'city' table is **'ID"** – which holds unique values for each row in its own table.

Identify the primary key in countrylanguage table: The primary key in 'countrylanguage' table is a composite of **'Language'** & **'CountryCode'**. The combination of these two fields gives a unique key. Languages and CountryCode are not unique and are in fact both repeated in this table, but together they form a unique composite key.

Identify the foreign key in city table: The foreign key in the 'city' table is **'CountryCode'** which holds unique values for each row in its own table.

Identify the foreign key in countrylanguage table: The foreign key in 'countrylanguage' table is **'CountryCode'** which holds unique values for each row in its table.

REFLECTION:

This assignment has helped me greatly to practice my SQL coding skills from lessons and learn many new features outside of the taught course content. The data set was good to work with as it was possible to perform some mathematical operations within the SQL query, which was good practice. Trying different ways to reach the same result was also helpful to gain better understanding of how different commands work. I also tried new ways of formatting and representing the results. This was all new and was initially a very good exercise in learning how to debug when there were syntax anomalies. I can better understand the SQL reference documentation for different commands and am now confident that I can learn new SQL skills faster.
The assignment has consolidated the course content on the different types of relational database relationships and Normalisation which is a much more complex topic than I had first thought. Moving forward from this assignment I would like to practice my SQL with much larger data sets, as the data set for this assignment and others that we have come across so far have been small. Practicing on a very large data set will make it easier to understand the different ways to optimise a query whilst considering processing speeds, memory use and other aspects, as well as the other difficulties associated with a very large data set. I would also like to learn database testing, which has been outside of the remit of this course. I plan to learn/practice more SQL and possibly get a certification as I have enjoyed the practical problem-solving part of this course module. I did not find the SQL coding part of this assignment stretched my SQL skills but enjoyed including some extension tasks. I did find the topic on normalization quite tricky, and I will be revisiting the topic again especially Normalisation forms 4NF and 5NF.