The aim of this exercise is to learn how to write queries in relational algebra (using `Rel`) and SQL (using `SQLite`). This first mission is splitted in two phases :

- Firstly, a list of 15 queries is given, for which you will have to provide code in both `Tutorial D` and `SQL`.
- Then, a list of 7 queries is given, for which you will have to find the most optimized version of the queries only in `Tutorial D` to improve the performance of your queries.

The queries need to be executed on a database inspired by the *Mondial* database [1] and is adapted for this course; this database is available on Moodle. Figure 1 represents a graphical representation of the database schema you will be using for the project. To help you gain a better understanding of the schema, we used Crow's foot notation[2] to represent the various relations.

You need to submit your answers on the INGInious platform; however, we believe you should be able to use database systems on your own machine as well, and wish to encourage you to think carefully about your queries. Hence, you must first evaluate your queries on your local system. The deadline for this project is **Friday, March 4th, 2022, at 11.55pm**. **Late submissions will *not* be considered** and we therefore kindly advise you to get started in a timely manner. The exercise needs to be made individually.

To test your queries on your local machine, the binaries of the two databases are provided. You can find the binaries for these databases, for both `RelDB` and `SQLite`, on Moodle.

We would also like to draw your attention to the following facts:

- The result of every query should be a set. While this is the default in `RelDB`, make sure that you use the appropriate statements in SQL to ensure that duplicate tuples are removed.
- Pay attention to the attributes of your results.
- The datasets that are provided only serve the purpose of giving you a basis to work on. It is by no means exhaustive nor representative of all the cases that can potentially arise using the aforementioned schema. (You are *heavily* encouraged to think of corner cases and adapt the datasets to your needs to be able to validate your queries). To allow you easily to add modifications in `RelDB`, you can find a summary of the different statements to add/update data on a relation.

1 Preparing your machine

1.1 Rel

`Rel` is an implementation of a database system that allows the execution of statements expressed in the `Tutorial D` language, which closely resembles the relational algebra discussed during the lectures. It is a tool implemented in Java. As a first step, please make sure that a Java Virtual Machine (JVM) is installed on your machine before proceeding with the following instructions. Should it be absent

[1] You can find more information on `https://www.dbis.informatik.uni-goettingen.de/Mondial/`
[2] `http://www.vertabelo.com/blog/technical-articles/crow-s-foot-notation`

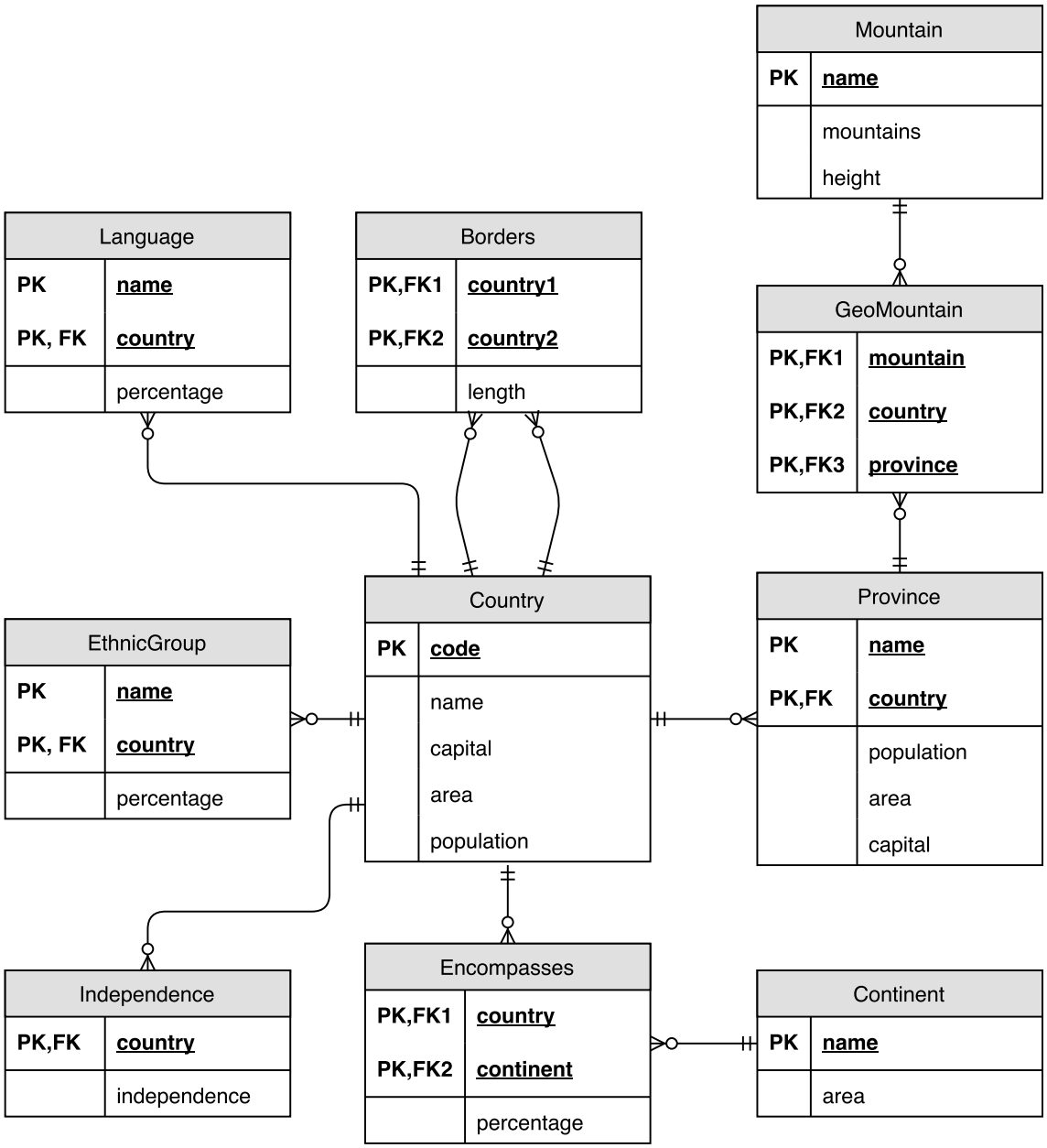

Figure 1: Relational scheme based on the Mondial database

from your system, download it and install it. To setup `Rel` on your platform, follow these instructions[3] :

1. Go to `https://sourceforge.net/projects/dbappbuilder/files/Rel/` and select the stable version (Rel version 3.014).
2. Download the archive file depending on your operating system.
3. Extract it.

[3] `https://sourceforge.net/projects/dbappbuilder/files/Rel/Rel%20version%203.014/`

4. Launch the executable file *Rel*.
5. Choose the command line interface into `Rel` to write your statements.

1.2 SQLite

`SQLite` is a very lightweight database system that allows the execution of statements expressed in `SQL`[4]. It only implements a subset of all the features usually expected from a full fledged SQL server such as for example Oracle or DB2. However its lightweight and responsiveness have gained it a huge worldwide adoption[5] despite of its inability to tackle large databases. Besides that, it is very easily installed on any platform and is an excellent vehicle for getting a basic understanding of the various SQL constructs. To setup `SQLite` on your laptop, proceed as follows[6]:

1. Go to `https://www.sqlite.org/download.html`.
2. Download the archive file depending on your operating system.
3. Extract it.
4. Launch the executable file *sqlite3*.

To get a GUI for `SQLite`, follow these additional instructions:

1. Go to `https://sqlitestudio.pl/index.rvt?act=download`.
2. Download the archive file depending on your operating system.
3. Extract it.
4. Launch the executable file *sqlitestudio*.

2 Querying in `Rel` and in `SQL`

In this first part of the project, we ask you to write two versions of each of the following 15 queries: one using the `Tutorial D` syntax and the other in `SQL`.

1. List the names of all countries.
2. List the names of all mountains with height (strictly) between 3000 and 4000 meters..
3. List all provinces in Europe whose area is less than 200. The results must be a relation composed of 2-tuples with attributes {province, country}, corresponding to the full province name and the full country name, respectively.
4. List the names of all languages in the database. The result must be a relation with one attribute called *language*.

[4]Did you know that *SQL* stands for Structured Query Language ?
[5]It is for instance used on iOS and Android devices
[6]http://www.sqlitetutorial.net/download-install-sqlite/

5. For each country that has declared independence, give its full name and its independence date.
6. List country codes and full names of countries that share borders with both countries China and India. The result relation is composed of 2-tuples with attributes {code, name}.
7. List the Swiss mountains that have a height between (and including) 4400 and 4500. The result must be a relation composed of 2-tuples with attributes *{mountain, height}*.
8. List all pairs of neighbouring countries. The result must be a relation composed of 2-tuples with attributes *{name1, name2}*, i.e., the full country names must be given. (For instance, a result could be *{Belgium, France}*.) Make sure that every pair of neighbouring countries is only included once.
9. List the name of the countries that do not have an island.
10. List all names that are shared by a province and a country's capital. (Hint: Your Tutorial D query may not have more than 50 characters.)
11. List the full names of the countries that border the United States, as well as the countries that border those bordering countries: i.e., the output should contain Mexico, but also Belize, which borders Mexico. The United States itself should not be part of the output.
12. Count the number of different ethnic groups. The result must be a relation composed of a 1-tuple with attribute *{cnt}*.
13. List the names of the countries with 3 borders or less — including those with no borders! The result must be a relation composed of 2-tuples with attributes *{name, cnt}*, where *name* is the name of the country and *cnt* is the number of borders of the country.
14. List the most prominent language(s) for each country; i.e., list for each country those languages for which the percentage is maximal. You are not allowed to use *SUMMARIZE*. The result must be a relation composed of 2-tuples with attributes *{country, name}*, where *country* is the code of the country and *name* is the name of the language.
15. In the given database the Borders relation is symmetric: if the country 1 (*c1*) borders country 2 (*c2*), country 2 also borders country 1. One may wish to check this property on a given database. Write a query that determines all tuples *{c1, c2}*, where *c1* and *c2* are the codes of the countries, for which the inverse direction is missing. (Note: on the given database, the output of this query should be empty; on a database in which the Border relation is not symmetric, however, it should return the violating tuples. Create a database yourself to check this.)
16. Some countries have incomplete information about their dialects and languages. List the name of countries for which the total percentage of reported languages does not reach 99% together with the proportion of the population for which this information is *unknown*. The result should have the form {name, unknown_lang_p}.

3 Optimization in Rel

Now that you have some basis in querying with `Tutorial D`, you will learn how to optimize your queries in `Rel`. For the 7 following queries, find the most optimized version of the queries only in `Tutorial D` to improve the performance of your queries. In a later mission, you will learn how to improve the queries in `SQL`.

1. For each continent, count the number of independent countries and sum all their populations. The result must be a relation composed of a 3-tuple with attributes *{continent, cnt_countries, sum_population}*.

2. List all mountains that are on a border. The result must be a relation composed of a 3-tuple attributes *{mountain, country1, country2}* where *country1* and *country2* are the names of the countries for which a border exists and *mountain* is the mountain located on the border.

3. List all countries that have a border of length at least 50, but no border of length longer than 700. The result must be a relation composed of a 1-tuple attribute *{name}* where *name* is the name of the country. Assume that the borders in the Borders relation might not be stored in a symmetric manner. It can be assumed there are no two countries with the same name.

4. For all countries that have a bordering country with a province with an area of less than 5000.0, list the languages; provide the country code and the language name. The result must be a relation composed of a 2-tuple with attributes *{country, language}* where *country* is the code of the country and *language* is the name of the language.

5. Luxembourg is in a specific topographic situation: it has exactly three neighboring countries (Belgium, Germany, and France), each of which are pairwise neighbors of each other as well. As a result, Luxembourg is surrounded by exactly three countries. List the abbreviations of all countries that are in a similar situation as Luxembourg. The result must be a relation composed by a 1-tuple attribute *{country}* where *country* is the code of the country.

6. List all codes for countries that have a total border length of at most 100.0, and a population of at least 5 million; in the calculation of border lengths, only borders of at least 50.0 should be considered.

7. List all capitals of provinces that have at least twice the population of capitals of other provinces in the same country, or any neighboring country. The result must be a relation composed by a 3-tuple attributes *{name, capital, country}* where *name* is the name of the province, *capital* is the capital of the province and *country* is the code of the country.

Appendix: Statements in Tutorial D

Creating a relation

VAR `name_of_relation` BASE RELATION
    { *name_of_attribute type*, *name_of_attribute type* }
KEY { *name_of_attribute*, *name_of_attribute* };

Note that only later in the lectures we will discuss what keys are in detail; for now, just assume that for technical reasons you need to list the set of attributes twice.

Inserting data in a relation

INSERT `name_of_relation` RELATION {
    TUPLE { *name_of_attribute value*, *name_of_attribute value* },
    TUPLE { *name_of_attribute value*, *name_of_attribute value* }
};

Querying a relation

`name_of_relation`

With a projection on the relation
`name_of_relation` {*name_of_attribute*}

With a selection on the relation
`name_of_relation` WHERE *condition*

Updating data in a relation

UPDATE `name_of_relation`
    WHERE *name_of_attribute* = *value*: { *name_of_attribute* := *new_value* };

Removing a relation

DROP VAR `name_of_relation`;