Wttj assignment - answers and explanations

Guillaume Baelen

Question 1:

In order to implement the question 1 and create a script that will count all the jobs per continent per profession category, I worked in two main step that I will try to explain here: First creating an initial script easy enough to load the data directly from the csv and parse them. Second create a phoenix app that would allow us to load the data into a database and prepare a more realistic application to answer the question 2 and 3

The initial script

The initial goal was simply to find a functional way to process the data in order to get to the desired table. The idea of the process can be recap like this:

1. Load the data from the csv using CSV.decode (Tried with simple Stream.map to parse the data but that proved to add more code for a very similar result)

2. Loop through the data to extract the location and the category name to pass the list to Enum.frequencies to get a map of the occurrences of each key that is the combination of the category and the continent.
    a. The continent is determined by passing the latitude and longitude to a plugin called geocode that will return the country code then a map that has the country code as a key and the continent as a value will allow us to replace the country code by the continent.

3. When we have the frequencies, we extract the categories and continents that are in the map and loop through the frequencies to compute the total of each category and continent and add them to the frequencies.

4. Lastly we take the final frequencies and make them go through a function that will format the data into a big string that will draw the table with all the values and display them to the console.

One thing that could be improved in the script is to make a join between the profession and the jobs so that we don't have to load the profession category separately from the offers. The main reason that I didn't do it here is for time management as I do not know yet how to do that kind of thing with Ecto. I could look for it but that might take a bit of time.

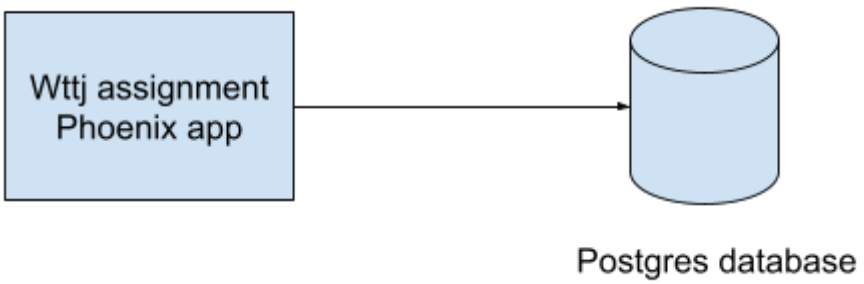
Second step: the rebirth of the phoenix

The second step which was simpler in terms of code was to set up a phoenix application. The process went like this:

1. Created a phoenix application to be able to use the database and prepare the third question.
2. Created the schema and seeds to load the data into database
3. Import the original script and refactored the part that loaded the csv to load the data from the database directly
4. Set up docker-compose to not have to install the database on the computer and be more flexible in the development

Question 2:

So now that we have an application with a database that can constantly be filled of data by a potential third party service, we can look more in detail on how we would scale our application to still be capable of counting in real-time the jobs offer per continent and professional categories.



Our current application

Little note: Here the addition of new entries in the database is considered to be separated from the answer and cannot be change. Otherwise another solution would be to change the tool that adds the jobs in the database so that it also stores the continent with the latitude and longitude which would make the computation of the counts per continent faster.

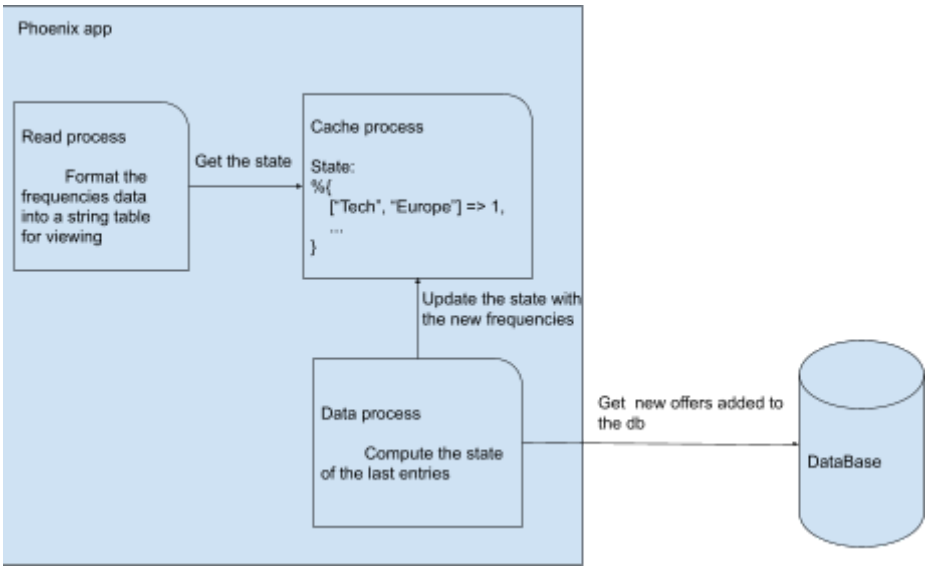
Let's start with the perfect scenario

If we start with the best case scenario where we can have our script to process the 1000 entries in the database in less than 1 secondes, then we only have one problem to handle: the constant growth.

Indeed with 1000 entries to handle every second, the size of the database will have doubled after a day when it would already take a day to calculate the 100 000 000 entries.

Recalculating the full database would make it impossible for us to have the data in real-time. This is why our goal should be to calculate only the newly added entries in the database and not the entire thing everytime.

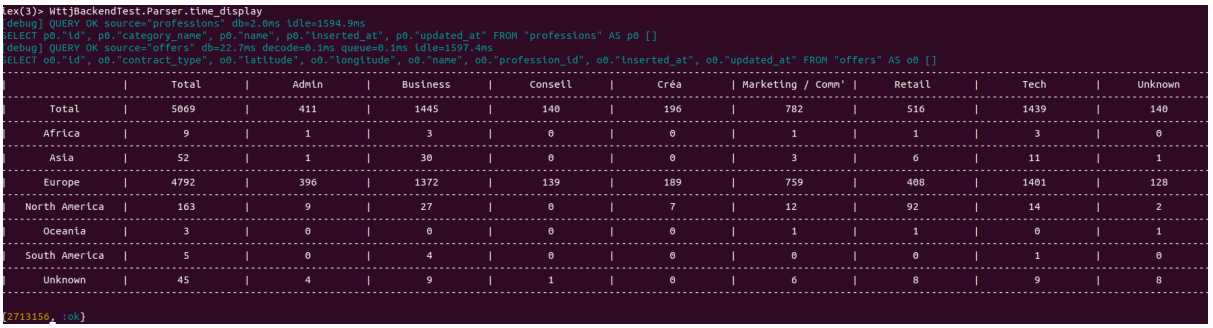
For that we can change our application by splitting the application in two parts that would be connected by a cache. First we would need to create a GenServer module that would basically use its state as a cache by keeping the frequency map and update it when needed. Then we would spawn a process that would continuously read the database and when each second 1000 new entries will be added, the process will get those entries and only those and add each category/continent with the according fields in the cache process. Finally we would just spawn a new process when we want to display the data that would just request the state of the cache process that should return an up to date map of frequencies that will only need to be parsed to have the table of jobs per category per continent as a string.



Now going back to the real world

The solution in an ideal world would work just fine assuming that we can wait a long time for the application to start (in the question we actually start with 100 000 000 entries already in place that need to be calculated).

If we make some measure on the current version of the application that compute the jobs count, we are far from the 1000 entries in 1 second as we can see on the screenshots below:

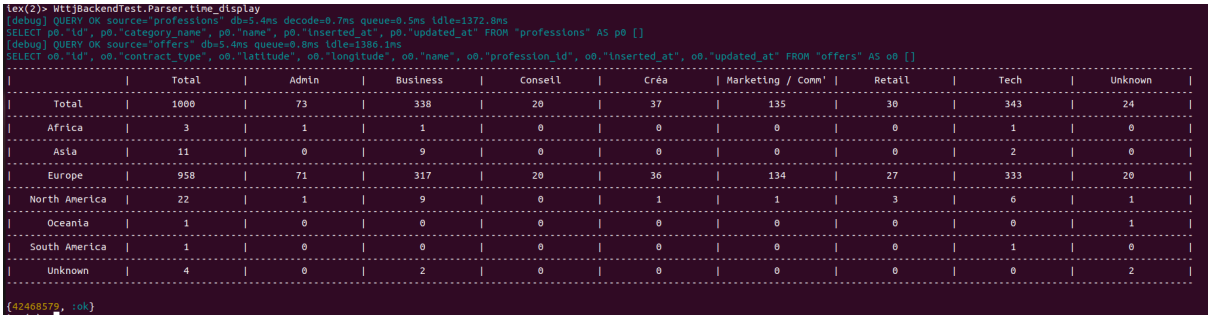


```
iex(3)> WttjBackendTest.Parser.time_display
[debug] QUERY OK source="professions" db=2.6ms idle=1594.9ms
SELECT p0."id", p0."category_name", p0."name", p0."inserted_at", p0."updated_at" FROM "professions" AS p0 []
[debug] QUERY OK source="offers" db=22.7ms decode=0.1ms queue=0.1ms idle=1597.4ms
SELECT o0."id", o0."contract_type", o0."latitude", o0."longitude", o0."name", o0."profession_id", o0."inserted_at", o0."updated_at" FROM "offers" AS o0 []
```

	Total	Admin	Business	Conseil	Créa	Marketing / Comm'	Retail	Tech	Unknown
Total	5069	411	1445	140	196	782	516	1439	140
Africa	9	1	3	0	0	1	1	3	0
Asia	52	1	30	0	0	3	6	11	1
Europe	4792	396	1372	139	189	759	408	1401	128
North America	163	9	27	0	7	12	92	14	2
Oceania	3	0	0	0	0	1	1	0	1
South America	5	0	4	0	0	0	0	1	0
Unknown	45	4	9	1	0	6	8	9	8

```
{2713156, :ok}
```

In this screenshot, we can see in the bottom the time in microseconds it took to process 5000+ entries and that took almost 2 minutes.



```
iex(2)> WttjBackendTest.Parser.time_display
[debug] QUERY OK source="professions" db=5.4ms decode=0.7ms queue=0.5ms idle=1372.8ms
SELECT p0."id", p0."category_name", p0."name", p0."inserted_at", p0."updated_at" FROM "professions" AS p0 []
[debug] QUERY OK source="offers" db=5.4ms queue=0.8ms idle=1386.1ms
SELECT o0."id", o0."contract_type", o0."latitude", o0."longitude", o0."name", o0."profession_id", o0."inserted_at", o0."updated_at" FROM "offers" AS o0 []
```

	Total	Admin	Business	Conseil	Créa	Marketing / Comm'	Retail	Tech	Unknown
Total	1000	73	338	20	37	135	30	343	24
Africa	3	1	1	0	0	0	0	1	0
Asia	11	0	9	0	0	0	0	2	0
Europe	958	71	317	20	36	134	27	333	20
North America	22	1	9	0	1	1	3	6	1
Oceania	1	0	0	0	0	0	0	0	1
South America	1	0	0	0	0	0	0	1	0
Unknown	4	0	2	0	0	0	0	0	2

```
{42468579, :ok}
```

In this second screenshot, we took only 1000 entries to ease the calculations and we still need 42 seconds to process them.

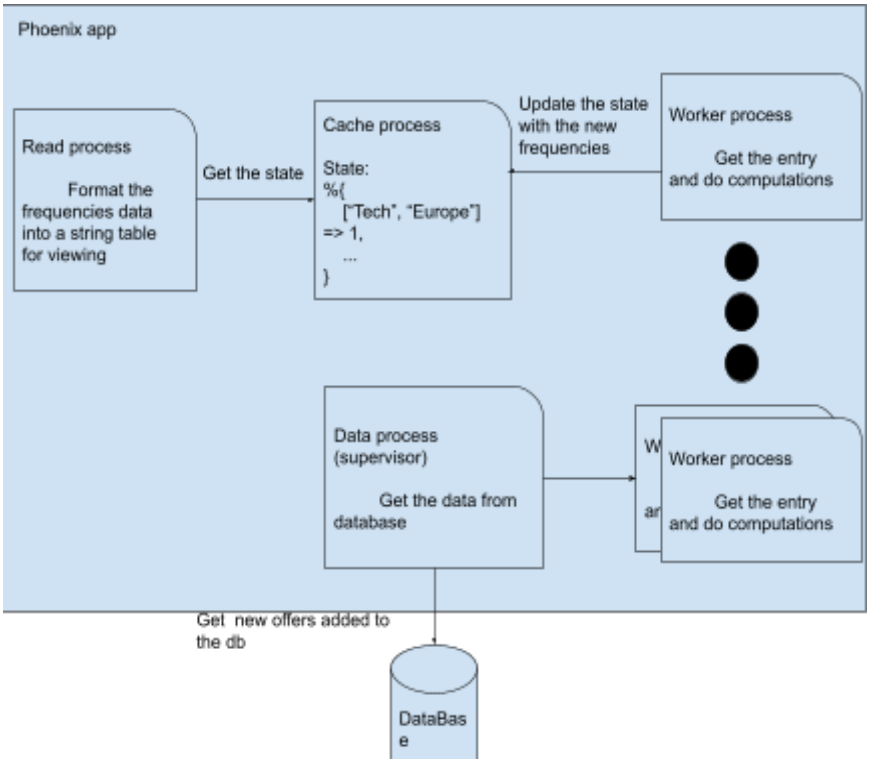
At this rate it would take an entire day to process the 100 000 000 entries and with the 1000 new entries per second coming, by the time the first batch of 1000 entries would have finished to be processed, 42k new entries would be waiting. Thus, making it impossible for the data to be viewed in real-time, and even worse, the more time passes the bigger the delay would be.

Nonetheless, that wouldn't be that difficult to arrive at a real-time application. Even better, the changes that we describe in the perfect scenario above are still very useful for this, only there are a few features that are missing to be perfectly complete!

To be able to approach something that would behave in real-time, we simply need to process more data at once and Elixir is perfect for that. Elixir is designed on very lightweight processes working concurrently meaning that one machine can handle millions of processes! (Well to be fair, the amount of processes will depend on the machine running the application but as nothing on that subject was specifically indicated, I will take the liberty to assume that we have a standard machine that can at least handle a few thousands of processes up to 1 000 000).

If we look at the schema we have in the part of the perfect scenario, the only thing that needs to be changed is the data process. We would need to replace it by a set of processes that would contain one supervisor that would take the entries from the database to redistribute them to a set of worker processes that would do the computation on the received entry (mainly getting the location as this is what takes the most time currently). Eventually if the database is too big, we could add a layer of multiple supervisors that would get assigned

a batch of entries to extract from the database and forward them to their own workers. Of course each worker would add to the cache directly but would still confirm success to the supervisor to avoid losing data in case of issues. The supervisor would then need to use its state to track what was given to each worker.



Another thing that can be done if the number of processes in one machine isn't enough, would be to horizontally scale by adding more copies of the phoenix application. That would mean that the Cache process will have to be replicated on each node to be able to stay consistent (or that one node is dedicated to be a cache and we use the distribution capacity of elixir to get the data but that would create a potential single point of failure that we usually want to avoid to have a reliable application). Ideally we would add an election system so that only one node has a master cache and the other will just copy it so that if the master loses connection, the other copies will choose a new master to control the changes. That would make it not only more scalable to remain as close to a real-time behavior as possible, but it would also make the system more resilient and will prevent wasting time recomputing everything after a crash.

Finally a last change that I would make to make the system more scalable in case of huge volume, would be to replace the Postgres database with a noSQL database that would be more fitted to handle high loads like those given in the subject. More precisely I would look at cassandra that uses consistent hashing to distribute its data that could come handy in the scenario where we want multiple supervisors to get data from the database. In this case each hash of cassandra could be given to a specific supervisor.

Last possible thing to improve scalability but only for when we have to recalculate everything that was already set, is to add a table with the continents and make an association to the job

offers (or add a field in the case of a nosql database). So that when we process a location we also store the continent returned which is what slows the application the most, having it stored into the database will allow us to get the data more quickly but like explained in the introduction that it would be better to get the continent when originally registering the data in the database.

Question 3:

Now that we want to have the data available through an API, I think that making a REST API will be more than enough. Soap and JSON RPC work by producing an action on the data, in our case we only want to return that data and REST is perfect for this. GraphQL would have been an interesting alternative but I do not know enough about it to implement it in this assignment.

Having a phoenix app already ready will make it easier to implement the endpoints with a rate limit. There is 4 endpoints that we could do:

- /api/v1/offers: will return all offers (with a default limit and pagination)
- /api/v1/offers/{offer_id} : will return a specific offer
- /api/v1/professions: that will return all professions available (with a default limit and pagination)
- /api/v1/professions/{professions_id} : that will return a specific profession

The endpoints that list all data of a resources should have the following filters:

- Jobs:
    - Pagination (offset based pagination)
    - Filtering of data
        - keyword in name
        - category id
    - Sorting
        - Name,
        - Contract_type
- Profession:
    - Pagination (offset based pagination)
    - Filtering
        - Category name
        - Keyword in profession_name
    - Sorting:
        - Name
        - Category_name