Dask Distributed Analysis of Covid-19 Papers

MAPD (mod. B) Final Project

Alessandro Casalino
Michele M. Crudele
Daniele Mellino

What is Dask?

- Dask is a flexible library for parallel computing in Python.

- Dask uses existing Python APIs and data structures making itself very familiar to Python users.

- It allows to parallelize the code on a single machine but also to distribute it to a cluster of hundreds of machines.

- In the first case the operations are executed in parallel over the CPUs of a single machine (vertical scaling), while in a distributed cluster they are performed over several CPUs that reside on different computers that compose the cluster (horizontal scaling).

- The main idea is to exploit the huge computation power of the cluster given by a larger number of CPUs and a larger amount of memory than a simple machine could have.

The Dask Cluster

Every cluster in Dask is made up of:

Client: is the machine from which a user has submitted a job or has distributed a program.

Scheduler: is the head/master node of the cluster and decides how to schedule the execution of the processes on the pool of resources**.**

Worker: represents a *computing node* of a cluster that receives the task that should be executed. A pool of resources is a set of workers.

Set-Up the Cluster

- In order to create the distributed cluster for this project, we exploit the script **dask-ssh**, that opens several SSH connections to our target computers. It works as follows:

```
$ dask-ssh 10.67.22.190  10.67.22.160  --nthreads < #cores / #workers >
```

  where in our case #cores = 8.

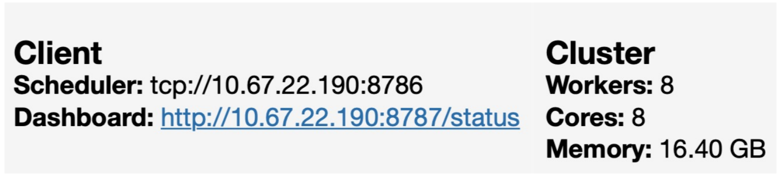
- The version of the libraries has to be the same in every machine of the cluster and all the data need to be downloaded in the same path.

- We decide to use a cluster made up of two VMs, with one worker per core we have (i.e. 8 workers, 4 per VM).

- To connect to the cluster, we simply use the Python command:

```
>>  client = Client( '10.67.22.190:8786' )
```

  that creates the client.



```
Client                                              Cluster
Scheduler: tcp://10.67.22.190:8786                  Workers: 8
Dashboard: http://10.67.22.190:8787/status          Cores: 8
                                                    Memory: 16.40 GB
```

Tasks

1 Implement word counter algorithm to check the more recurrent words in the papers;

2 Discover the best and worst represented countries in the research;

3. Get the embedding of the papers;

4. Compute the cosine similiarity between each title of the papers and find out a couple of papers with the highest cosine similiarity score.

The Dataset

The dataset consists of 1000 papers in json format randomly chosen from the ones in kaggle. The json downloaded are in a multi-lines format, that we transform into a single-line format for a reason that will be clear with the next slide.
To this end, we use the following script :

```
>>  for filename in *.json; do
>>       echo $filename
>>        jq -c . $filename > ../json_singleline/$filename
>>  done
```

Which API?

We can think to each job that we send to the scheduler as a DAG (Directed Acyclic Graph). Dask have different APIs which create and optimize this kind of graph. Bag and DataFrame are some of them.

We are dealing with json data, whose format does not really enforce strict structure and datatypes. That is why we choose to use dask bag to read and preprocess our data.

Most Frequent Words Inside the Papers

1. We load the data into a dask.bag with the function *read_text.* This function interprets every line of our files as a separate element. This is why we converted the data into a line-delimited json format.
2. The resulting items in the bag are strings. We convert each string (that encodes a json) in a dictionary thanks to the function json.loads.
3. At this point, dask creates one partition per file by default: this is the most sensible choice: in this way indeed, we avoid to group all the files together and then split them in partitions. This operation would require a lot of time and resources, so it is better to read each file as a single partition and to repartition them later, as we will do.

Where is the text of each paper?

The text we want to use for accomplish the first task is inside the field *body_text* under the key *text*. We select only this field using the *pluck()* function.
However, we have more than one *text* for each paper. So we need to merge them:

```
>>  texts = js.pluck("body_text").repartition(100).map(merge_text)
```

Once we plucked the *body_text* field, we decrease the number of partitions in order to save computation time. In fact there is no more need of 1000 partitions since now the dimension of the files is reduced. Therefore we repartition the bag to 100 partitions. We will see later that this is a good choice.

Text Cleaning

The text of the papers has to be sanitized: it presents punctuations, single numbers and stopwords (i.e. the most common words in a language). We use a pre-build set of stopwords from the *wordcloud* package to which we add a list of common words that are missing which we remove a posteriori.

So, we create a function that performs this cleaning procedure and we map it to the bag that contains the texts of all the papers:

```
>>  text_clean = texts.map(clean_func)
```

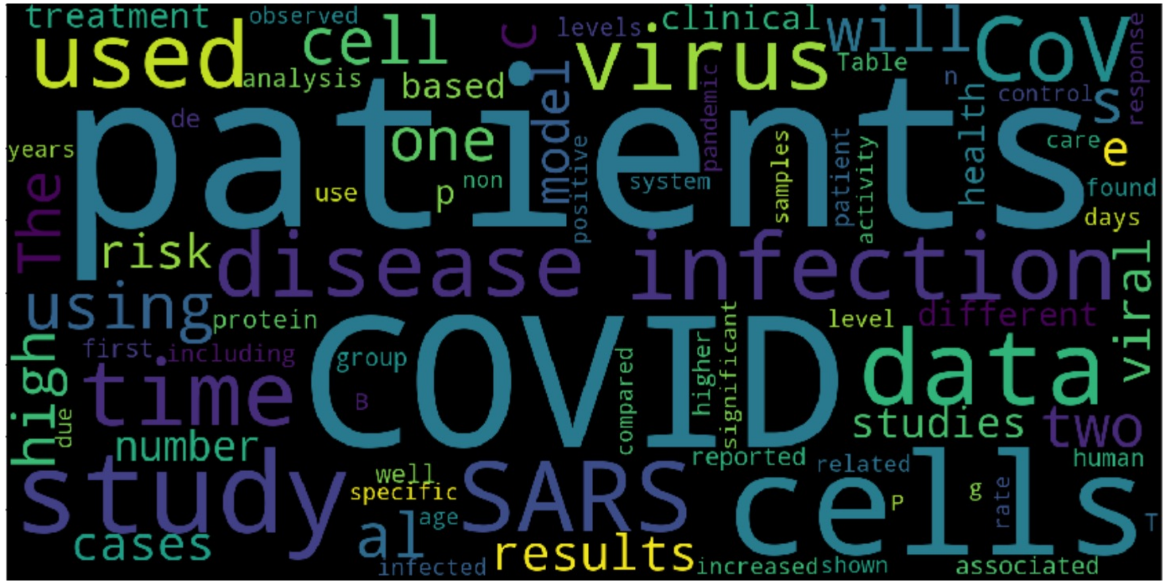
Map-Reduce

1. We find the frequencies of the words inside each paper:

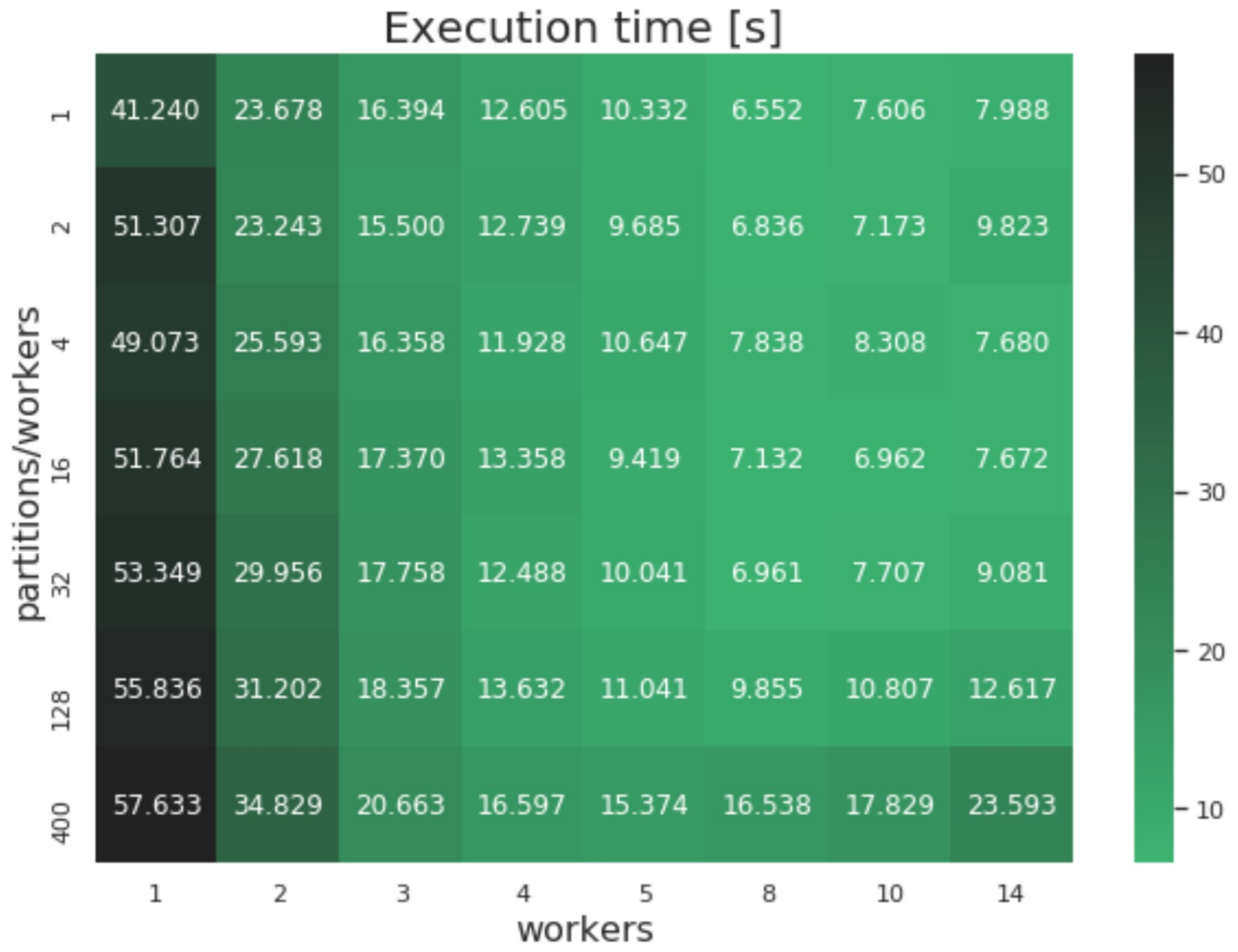
```
>>  words = text_clean.map(count_words)
```

2. We reduce them to one single list using the *foldby* method, that is more performant than *groupby*

```
>>  word_count = words.flatten().foldby('word', binop=incr,
                        initial=0, combine = lambda x,y: x+y,
                        combine_initial=0).compute()
```



How do the workers and the partitions influence the computation time?



partitions/workers	1	2	3	4	5	8	10	14
1	41.240	23.678	16.394	12.605	10.332	6.552	7.606	7.988
2	51.307	23.243	15.500	12.739	9.685	6.836	7.173	9.823
4	49.073	25.593	16.358	11.928	10.647	7.838	8.308	7.680
16	51.764	27.618	17.370	13.358	9.419	7.132	6.962	7.672
32	53.349	29.956	17.758	12.488	10.041	6.961	7.707	9.081
128	55.836	31.202	18.357	13.632	11.041	9.855	10.807	12.617
400	57.633	34.829	20.663	16.597	15.374	16.538	17.829	23.593

Most and Less Productive Countries and Universities

The two following tasks are about finding which are the worst and best represented countries and universities in the research.

We load the data in a dask.bag in the very same way as the previous point, but now we pluck the *authors* field, since it is the one that contains the country and the university the authors belong to.

Also in this case, the data have to be sanitized: there are cases in which the same country is called in different ways (e.g. UK, U.K., U. K., United Kingdom) or the authors are not all of the same country. Moreover, in some cases the fields are missing or they are empty. We define a function to solve all of those problems and we map it to the bag in which we loaded the data.

From Bag to DataFrame

Once we have pre-processed the data, we can load them in a dataframe structure, that is more performant with structured data with respect to dask bags. They are also very convenient and easy to use since they can be treated like they were pandas.dataframe.

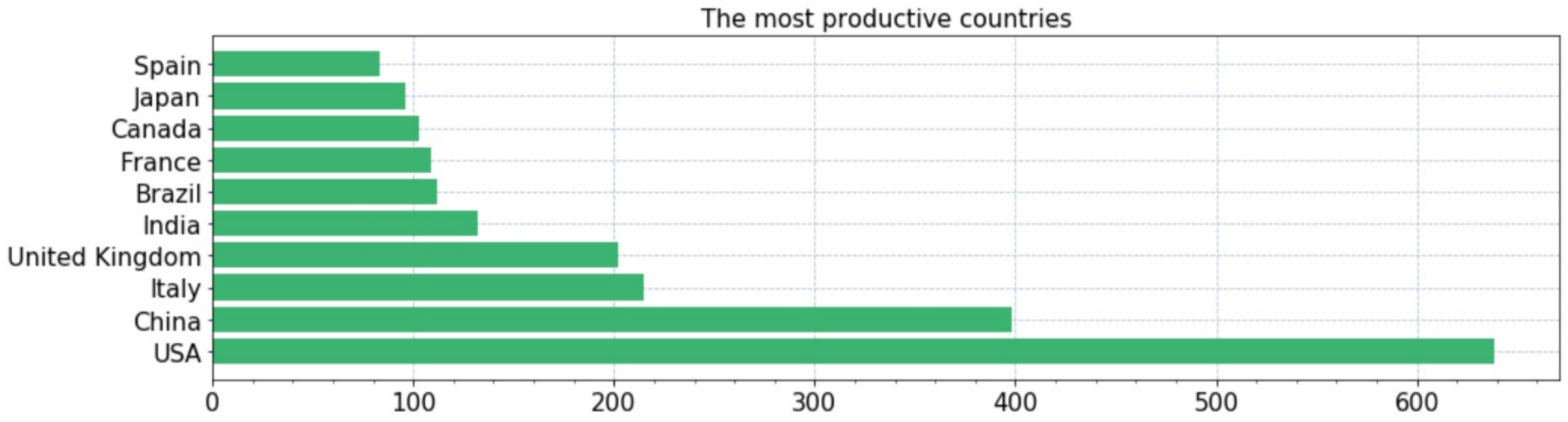
```
>>  authors_df = (authors.repartition(100)
                    .flatten().map(flatten)
                    .filter(lambda record: tokenizer.tokenize(record["Country"]) ==
                          word_tokenize(record["Country"], language = 'english'))
                    .to_dataframe(meta = {"name": str, "surname": str,
                                         "University": str, "Country": str}))
```

We specify the type of each column manually since this improves the performance and prevents dask to infer the wrong type for the columns.
With the data collected in a dataframe, the following tasks are very easy to be solved.

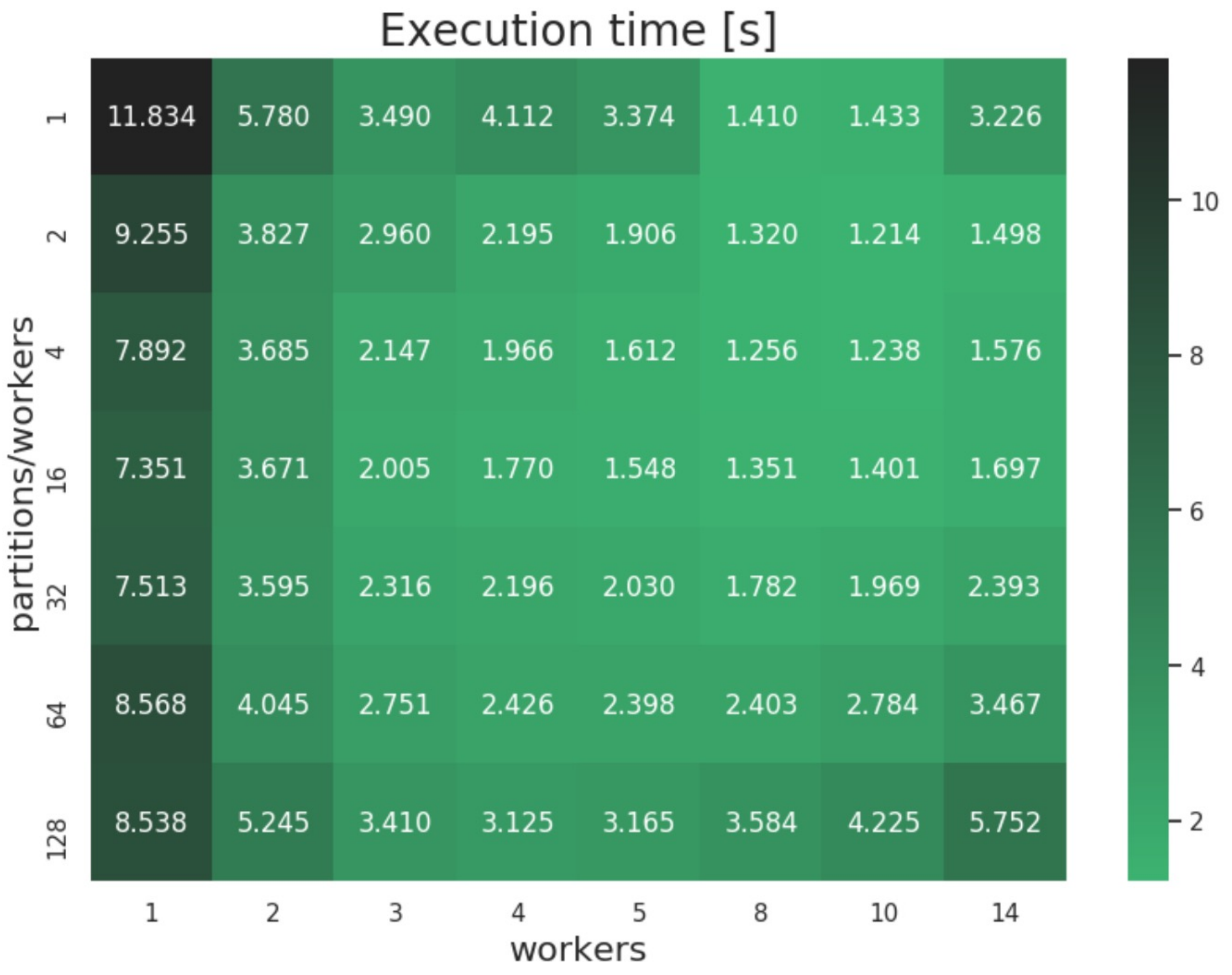
The countries

We select from the dataframe only the column related to the countries, then we use the *value_counts()* function to compute the occurrence of each of them. Then, we persist the result since we will use those data more than once, so we avoid them to be distributed around the cluster multiple times:

```
>>  countries = authors_df.Country.value_counts().persist()
```



Timing analysis of the code for the countries

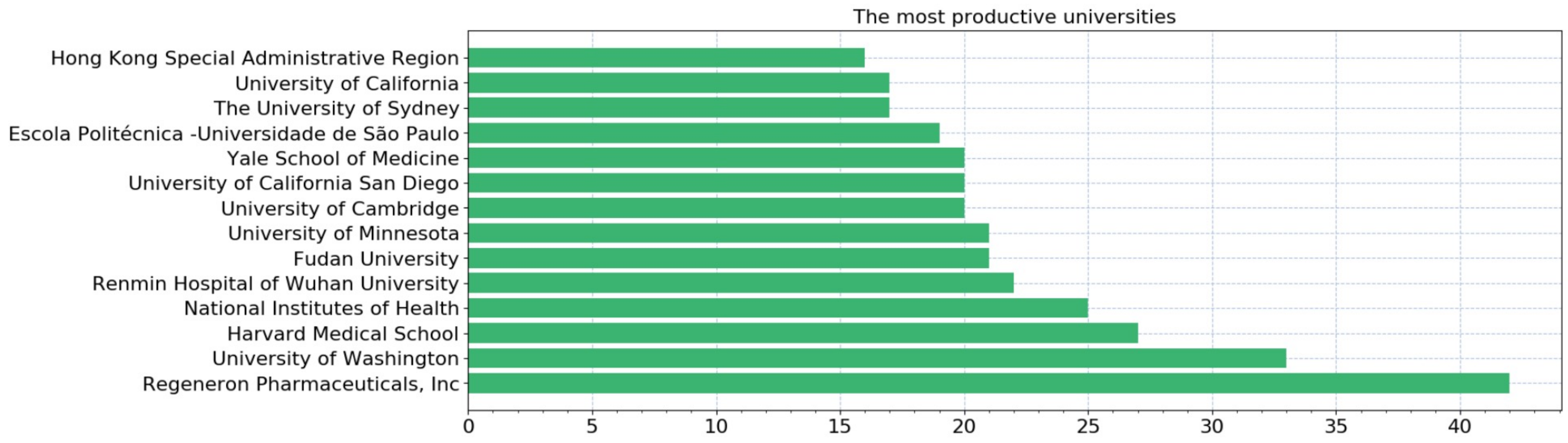


partitions/workers	1	2	3	4	5	8	10	14
1	11.834	5.780	3.490	4.112	3.374	1.410	1.433	3.226
2	9.255	3.827	2.960	2.195	1.906	1.320	1.214	1.498
4	7.892	3.685	2.147	1.966	1.612	1.256	1.238	1.576
16	7.351	3.671	2.005	1.770	1.548	1.351	1.401	1.697
32	7.513	3.595	2.316	2.196	2.030	1.782	1.969	2.393
64	8.568	4.045	2.751	2.426	2.398	2.403	2.784	3.467
128	8.538	5.245	3.410	3.125	3.165	3.584	4.225	5.752

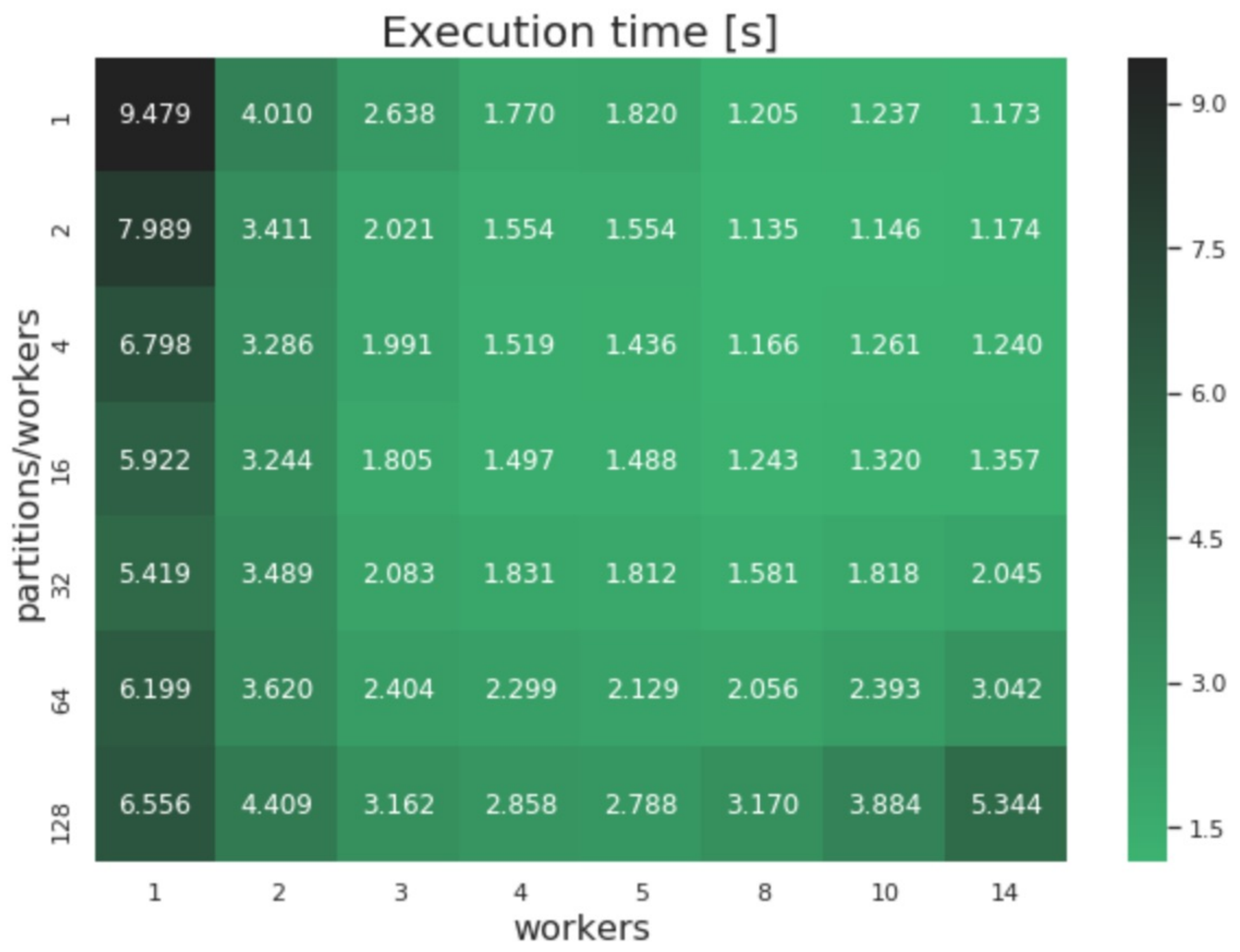
The universities

We select from the dataframe only the column related to the universities, then we use the *value_counts()* pandas function to compute the occurrence of each of them. Also in this case we persist the result.

```
>>  univs = (authors.repartition(100)
                  .flatten().map(flatten)
                  .to_dataframe(meta = {"name": str, "surname": str,
                                        "University": str, "Country": str})
                  .University.value_counts().persist())
```



Timing analysis of the code for the universities



partitions/workers	1	2	3	4	5	8	10	14
1	9.479	4.010	2.638	1.770	1.820	1.205	1.237	1.173
2	7.989	3.411	2.021	1.554	1.554	1.135	1.146	1.174
4	6.798	3.286	1.991	1.519	1.436	1.166	1.261	1.240
16	5.922	3.244	1.805	1.497	1.488	1.243	1.320	1.357
32	5.419	3.489	2.083	1.831	1.812	1.581	1.818	2.045
64	6.199	3.620	2.404	2.299	2.129	2.056	2.393	3.042
128	6.556	4.409	3.162	2.858	2.788	3.170	3.884	5.344

Comment on the timings

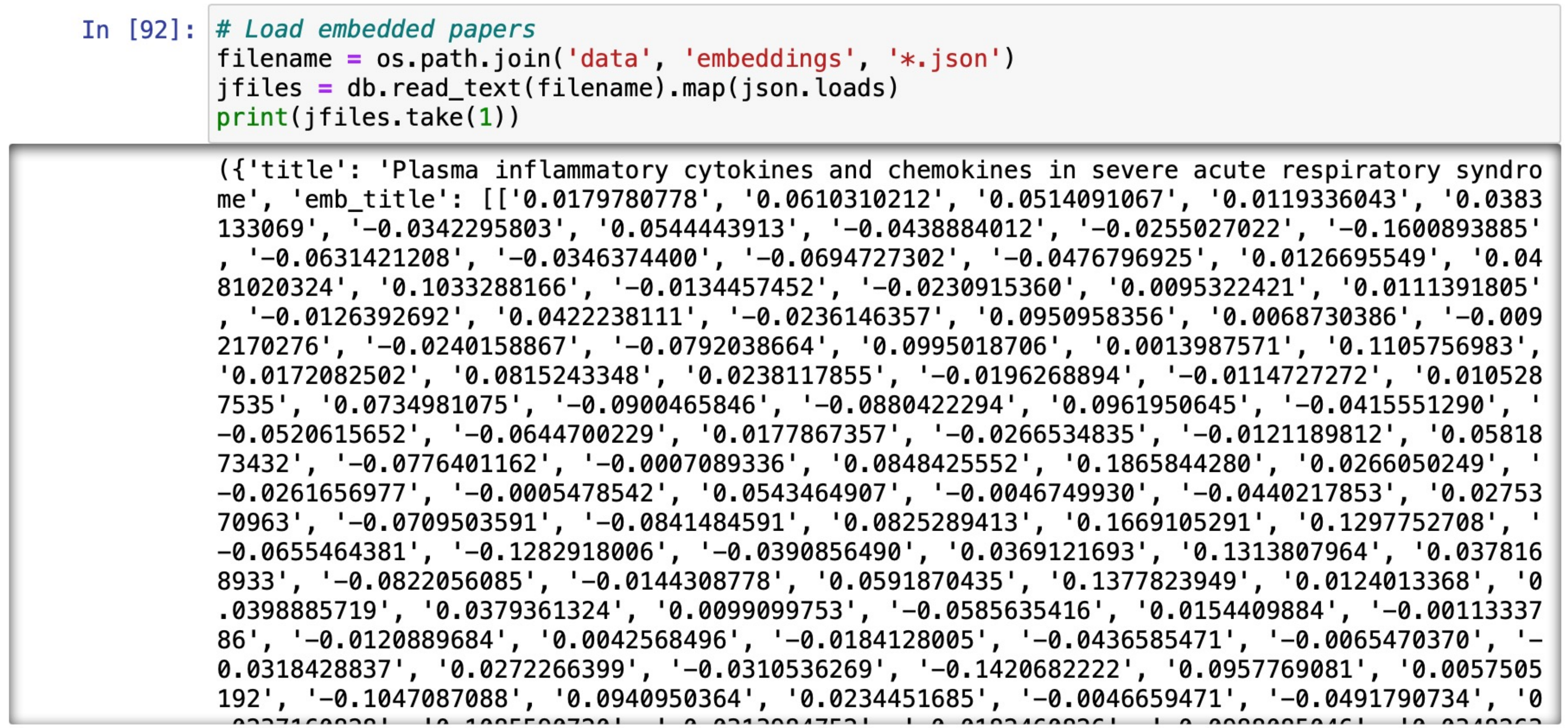
- All the previous heatmaps about the timings show that both the workers and the partitions should not be neither too many nor too few.
- As far as the partitions, their size should fit comfortably in memory but also not be too many, since every operation on every partition takes the central scheduler a few hundred microseconds to process. Moreover, having too many partitions means that a lot of chunks of data has to be transmitted through the cluster and this cause a lot of overhead.
- As far as the workers, having more than one worker is good to distribute the code, but too many of them can cause overhead too, since the workers have to exchange data between each other and this operation takes a lot of time if the workers are too many. Another consideration that we can do about this aspect, is that it is not sensible to use a number of workers that is higher than the number of cores we have in our cluster, since the workers are supposed to execute tasks in parallel: looking at the heatmaps indeed, we can see that the computation time starts increasing when we set a number of workers greater than 8, that is the number of cores we have in our cluster.

Embeddings for the titles of the papers

In NLP a common technique to perform analysis over a set of texts is to transform the text to a set of vectors, each one representing a word inside the document.

We transform the titles of the papers into their embedding version by using the pre-trained model available on FastText page.

- We load the data plucking only the metadata field of each paper, since it contains the title.
- We load the english FastText model using the *fasttext* library.
- We create one dictionary per paper, each one with its title and the embedding for it.
- We save each dictionary in a .json file that we load using a dask.bag.



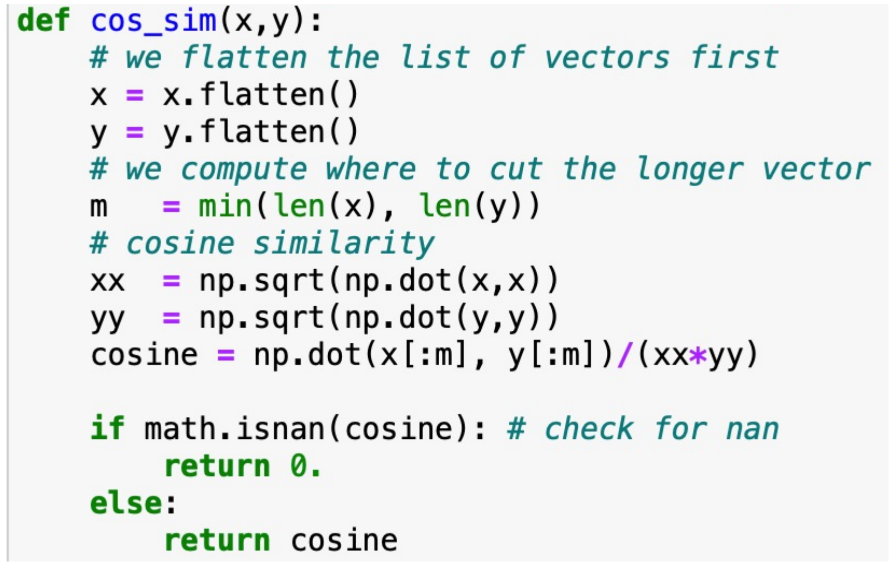
```
In [92]: # Load embedded papers
         filename = os.path.join('data', 'embeddings', '*.json')
         jfiles = db.read_text(filename).map(json.loads)
         print(jfiles.take(1))
```

```
({'title': 'Plasma inflammatory cytokines and chemokines in severe acute respiratory syndro
me', 'emb_title': [['0.0179780778', '0.0610310212', '0.0514091067', '0.0119336043', '0.0383
133069', '-0.0342295803', '0.0544443913', '-0.0438884012', '-0.0255027022', '-0.1600893885'
, '-0.0631421208', '-0.0346374400', '-0.0694727302', '-0.0476796925', '0.0126695549', '0.04
81020324', '0.1033288166', '-0.0134457452', '-0.0230915360', '0.0095322421', '0.0111391805'
, '-0.0126392692', '0.0422238111', '-0.0236146357', '0.0950958356', '0.0068730386', '-0.009
2170276', '-0.0240158867', '-0.0792038664', '0.0995018706', '0.0013987571', '0.1105756983',
'0.0172082502', '0.0815243348', '0.0238117855', '-0.0196268894', '-0.0114727272', '0.010528
7535', '0.0734981075', '-0.0900465846', '-0.0880422294', '0.0961950645', '-0.0415551290', '
-0.0520615652', '-0.0644700229', '0.0177867357', '-0.0266534835', '-0.0121189812', '0.05818
73432', '-0.0776401162', '-0.0007089336', '0.0848425552', '0.1865844280', '0.0266050249', '
-0.0261656977', '-0.0005478542', '0.0543464907', '-0.0046749930', '-0.0440217853', '0.02753
70963', '-0.0709503591', '-0.0841484591', '0.0825289413', '0.1669105291', '0.1297752708', '
-0.0655464381', '-0.1282918006', '-0.0390856490', '0.0369121693', '0.1313807964', '0.037816
8933', '-0.0822056085', '-0.0144308778', '0.0591870435', '0.1377823949', '0.0124013368', '0
.0398885719', '0.0379361324', '0.0099099753', '-0.0585635416', '0.0154409884', '-0.00113337
86', '-0.0120889684', '0.0042568496', '-0.0184128005', '-0.0436585471', '-0.0065470370', '-
0.0318428837', '0.0272266399', '-0.0310536269', '-0.1420682222', '0.0957769081', '0.0057505
192', '-0.1047087088', '0.0940950364', '0.0234451685', '-0.0046659471', '-0.0491790734', '0
```

Cosine Similarity

In this section we compute the cosine similarity between the titles of the papers using the embeddings generated in the previous part of the notebook.

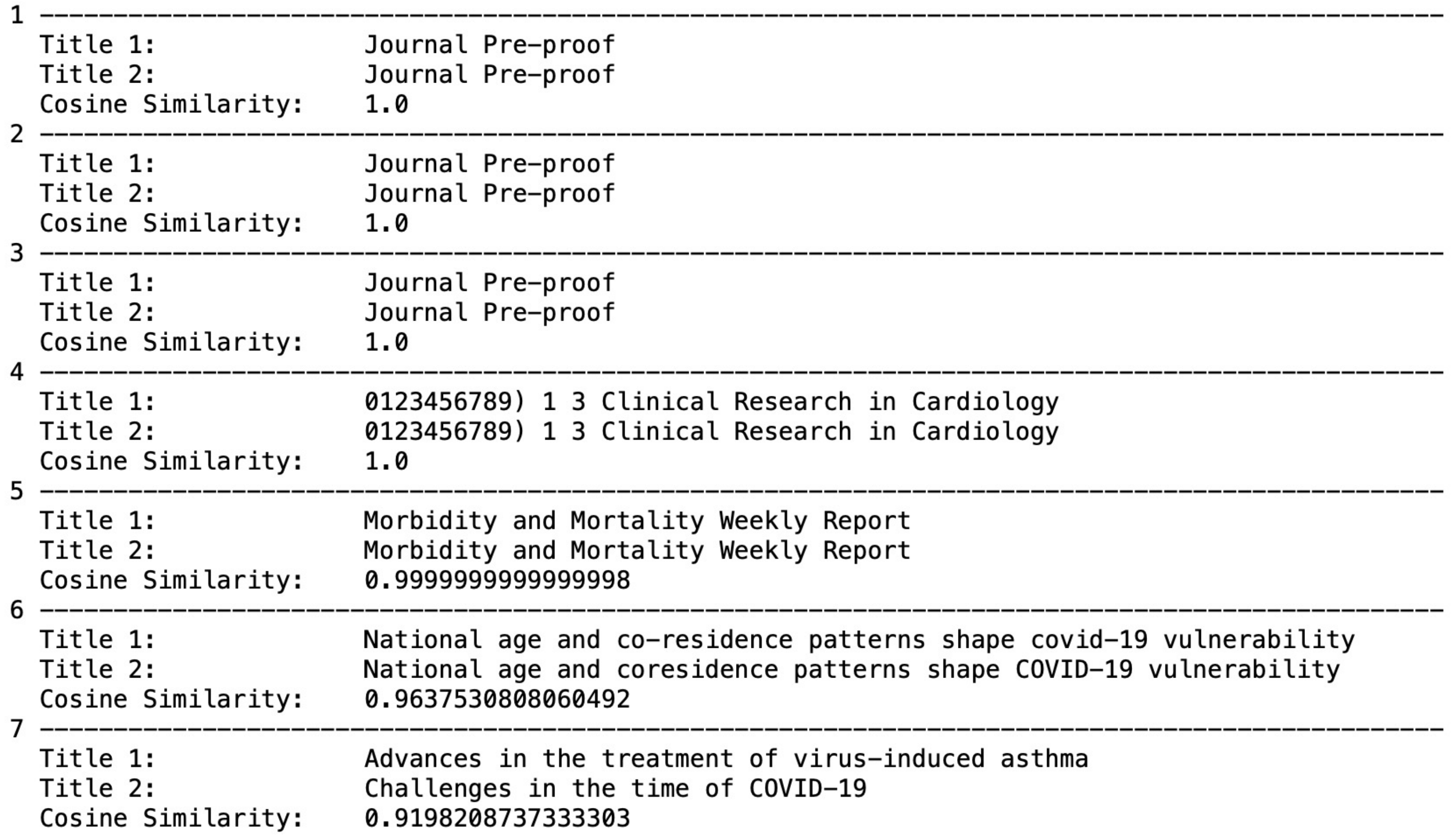
Each embedding is a list of vectors, with one vector for each word of the title. As a consequence, the embeddings are not all of the same length, so we have to cut some of them in order to be able to compute the cosine similarity: to this end, we compute the minimum length between the two titles we are analyzing and cut the larger one. For example, if two titles have 10 and 16 words respectively, we compute the cosine of the first 10 words only.



```python
def cos_sim(x,y):
    # we flatten the list of vectors first
    x = x.flatten()
    y = y.flatten()
    # we compute where to cut the longer vector
    m   = min(len(x), len(y))
    # cosine similarity
    xx  = np.sqrt(np.dot(x,x))
    yy  = np.sqrt(np.dot(y,y))
    cosine = np.dot(x[:m], y[:m])/(xx*yy)

    if math.isnan(cosine): # check for nan
        return 0.
    else:
        return cosine
```

This time we proceed with a sequential code since it is fast and we save the results in a pandas dataframe. We do not use dask dataframe in this case since we have computed all the results and we will not perform other complex operations with them, so there is no point in using a distributed dataset this time.



```
1 -----------------------------------------------------------------------------------------------
  Title 1:              Journal Pre-proof
  Title 2:              Journal Pre-proof
  Cosine Similarity:    1.0
2 -----------------------------------------------------------------------------------------------
  Title 1:              Journal Pre-proof
  Title 2:              Journal Pre-proof
  Cosine Similarity:    1.0
3 -----------------------------------------------------------------------------------------------
  Title 1:              Journal Pre-proof
  Title 2:              Journal Pre-proof
  Cosine Similarity:    1.0
4 -----------------------------------------------------------------------------------------------
  Title 1:              0123456789) 1 3 Clinical Research in Cardiology
  Title 2:              0123456789) 1 3 Clinical Research in Cardiology
  Cosine Similarity:    1.0
5 -----------------------------------------------------------------------------------------------
  Title 1:              Morbidity and Mortality Weekly Report
  Title 2:              Morbidity and Mortality Weekly Report
  Cosine Similarity:    0.9999999999999998
6 -----------------------------------------------------------------------------------------------
  Title 1:              National age and co-residence patterns shape covid-19 vulnerability
  Title 2:              National age and coresidence patterns shape COVID-19 vulnerability
  Cosine Similarity:    0.9637530808060492
7 -----------------------------------------------------------------------------------------------
  Title 1:              Advances in the treatment of virus-induced asthma
  Title 2:              Challenges in the time of COVID-19
  Cosine Similarity:    0.9198208737333303
```