Analysis of Covid-19 papers

MAPD-B Project Presentation - Group 9

Belli Luigi, Brocco Luca, Tigan Cristian

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Project overview

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

The idea

We want to analyze papers related to **COVID-19**, **SARS-CoV-2** and related coronaviruses. The dataset we will use is a subset of a still growing dataset that counts over **1'000'000** papers

This dataset is part of a real world ongoing research known as COVID-19 Open Research Dataset Challenge (**CORD-19**). From the related Kaggle website, different versions of the dataset can be downloaded.
We took two different ones:

Structure of the Data

From this link we can view the complete **JSON file structure**

We printed out the first layer of the JSON and its keys:

```
example = filenames[1]
with open(example, 'r') as f:
    data = json.load(f)

print('Each JSON is a dictionary with some keys:')
print(data.keys())
```

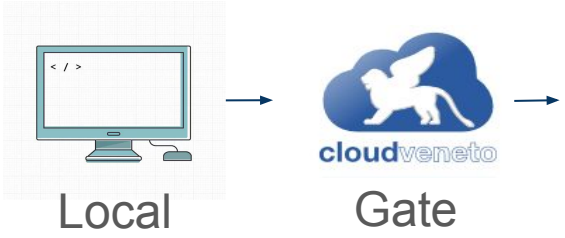
```
Each JSON is a dictionary with some keys:
dict_keys(['paper_id', 'metadata', 'abstract',
'body_text', 'bib_entries', 'ref_entries', 'back_matter'])
```

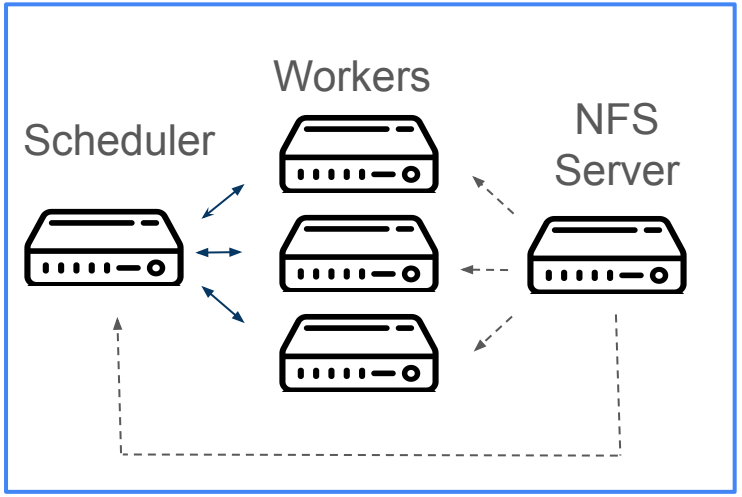
Setup of distributed environment

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Setup

- On CloudVeneto's infrastructure:
  - 5 machines: **1 scheduler**, **3 workers** (1 VCPU, 4GB RAM, 25 GB storage) and an instance as **NFS server** [read only] (2 VCPUs, 4 GB RAM, 25 GB + 60 GB storage)
- Distributed Framework: Dask
- OS : Ubuntu 22.04.5 LTS





Setup

We access the machines via ssh protocols:

```
ssh -L 8888:localhost:8888 -J ncognome@gate.cloudveneto.it -i ~/.ssh/chiave.pem
ubuntu@10.67.22.173
```

We only need to access the **scheduler**, since it is already connected via ssh to the workers, as we set them up as **ssh known hosts**

Scheduler's IP: `ubuntu@10.67.22.173`

Worker's IPs:
```
ubuntu@10.67.22.153
ubuntu@10.67.22.150
ubuntu@10.67.22.183
```

We save all IPs in a Python list

Setup

By launching **Jupyter Notebook** on the right port, we are able to code from our **local browsers**

```
jupyter notebook --no-browser --ip=127.0.0.1 --port=8888
```

To start the cluster we configured Dask with the workers and scheduler's IP addresses. The first one in the list is scheduler's IP. We also specify the **ports** for the scheduler and the **Dashboard**.

```
try: ## This starts the cluster but if it was started and not closed, this will raise error, so
    cluster = SSHCluster(
        [scheduler, worker1, worker2, worker3],
        connect_options={"known_hosts": None},
        scheduler_options={"port": 8786, "dashboard_address": ":8797"},
    )
    client = Client(cluster)
except RuntimeError: # this is how to resume the cluster
    clear_output()
    client = Client(scheduler + ':8786') # restarts the cluster, by simply recalling it
```

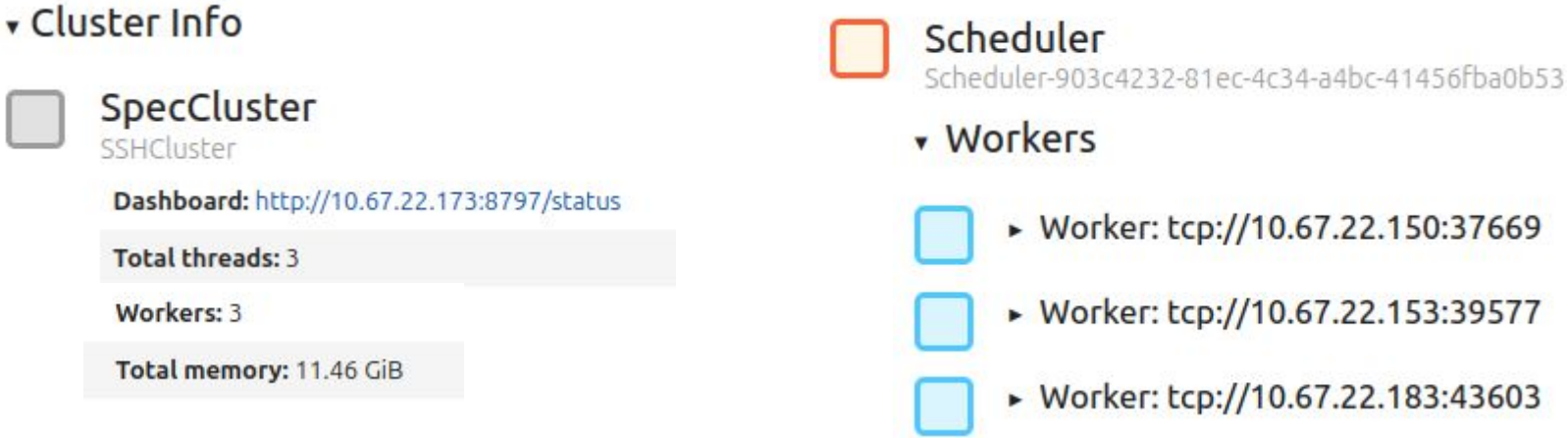
Setup

To access the **Dask Dashboard**, we open up another port via **ssh**:

```
ssh -L 8797:localhost:8797 -J ncognome@gate.cloudveneto.it -i ~/.ssh/chiave.pem
ubuntu@10.67.22.173
```

Then we just need to click http://localhost:8797 from our local pc to access it

And we are ready to go!



Task 1:
Word Counter Distributed Algorithm

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Word Counter Distributed Algorithm

We will program an algorithm that lets us know which words appear in a document and how many times they occur. This will be articulated in two separate phases:

Map Phase

For each document D_i we want to produce a set of pairs $(w, cp(w))$ where w is every word that is found in that document

$$w \in D_i$$

and $cp(w)$ is the number of occurrences of w in D_i

Reduce Phase

For every word w found in Map Phase, compute $(w, c(w))$ where $c(w)$ denotes all occurrences of w in all documents:

$$c(w) = \sum_{k=1}^{n} cp_k(w)$$

Word Counter Distributed Algorithm

First off, we import the **JSON** files:

```
# get all filenames in the selected folder
filenames = glob.glob(os.path.join(data_path, '*.json'))
filenames = [os.path.abspath(f) for f in filenames]
```

For this task we decided to create a **Dask Bag** containing all the JSON files.

```
# create a bag
partition_size = 10

def load_json_file(path):
    with open(path) as f:
        return json.load(f)

json_bag = db.from_sequence(filenames, npartitions=partition_size).map(load_json_file)
```

We will explore later the best configuration in terms of speed

Word Counter Distributed Algorithm

We then setup the **word counting algorithm** itself:

Given a ‘text_body’, we want to **join** all of its parts in **lowercase**:

```
string = " ".join([txt['text'] for txt in body_text]).lower()
```

We then remove all the **numbers** and **punctuations**

```
string = re.sub(r"[^a-z\s]", " ", string)
```

We also want to remove **english stopwords**, **common paper language words** (which are not meaningful for our goal) and **single letters**:

```
stop_words = set(en_stopwords) | {"fig","figure","et","al","results","also",
                                  "used","using","may","one","two","de","however"}

stop_words |= set("p o i u y t r e w q l k j h g f d s a z x c v b n m".split())
filtered_words = [w for w in tokens if w not in stop_words]
```

Word Counter Distributed Algorithm

Let's see how the algorithm behaves on a single file. We will print the most occurring words first:

```
words_in_body(load_json_file(example_file)['body_text'])
```

```
[('ace', 45),
 ('tissues', 43),
 ('oral', 42),
 ('cells', 39),
 ('cell', 26),
 ('expression', 25),
 ('ncov', 25),
 ('cavity', 16),
 ('rna', 16),
 ('single', 14),
 ('tongue', 14),
 ('data', 13),
 ('seq', 13),
 ('epithelial', 12),
 ('including', 11),
 ('infection', 11),
 ('study', 11),...
```

The output **looks promising**: let's configure a **Dask Graph** and perform the task on a **parallelized** setting!

Word Counter Distributed Algorithm

Map phase:

```
# extract each document's body text
body_texts = json_bag.pluck("body_text")

# count words (map phase)
words_counts = body_texts.map(words_in_body)
```

Reduce phase:

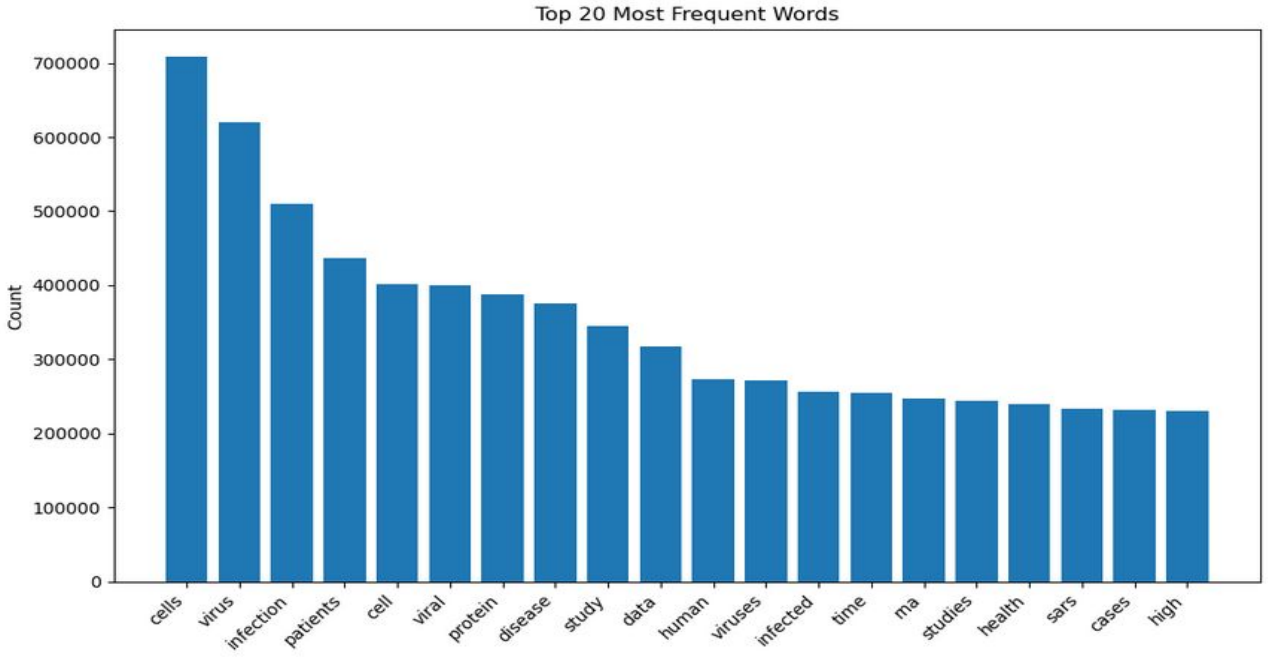
```
# increment counters through all documents
def increment(tot, x):
    return tot + x['n_counts']

# reduce phase with foldby method
words_counted_reduce = words_counts.flatten().foldby('word',
                                                     binop=increment,
                                                     initial=0,
                                                     combine = lambda x,y: x+y,
                                                     combine_initial=0
                                                    )
```

Word Counter Distributed Algorithm

Dask's map operation is **lazy**: the task itself will be performed only when you ask to compute it:

```
word_counts = words_counted_reduce.compute()
```

We plot the results in a histogram:



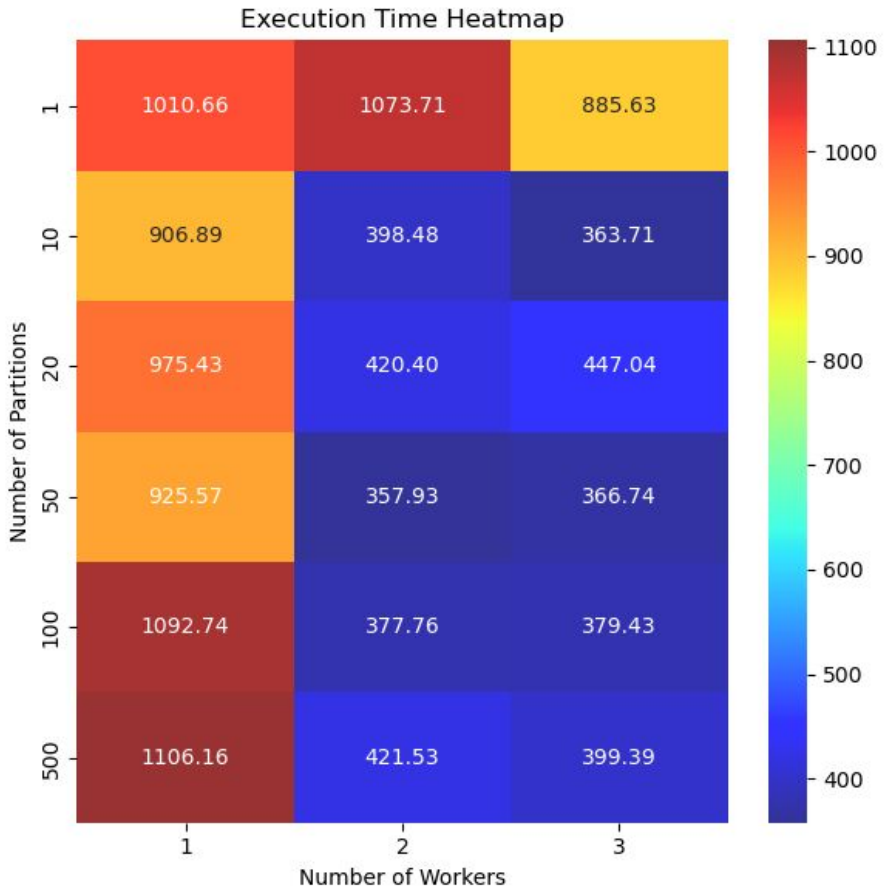
Grid search on speed

We want to make sure that the tasks are done **as quickly as possible**

We tested the previous task in different configurations in terms of **Partitions** and **Workers** (time is in seconds)

It looks like that for this task's purposes it's best to have **2-3 Workers** and a number of **Partitions** between **10** and **100**

<u>We could scale this task easily!</u>



Task 2:
Most and least represented countries & institutes

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

The Goal

Objective: Identify the most and least represented countries and institutes in COVID-19 research.

Approach:

1. **Extract** country and institution of affiliations for each author
2. **Construct** a Dask DataFrame for efficient grouping and counting
3. **Clean** data: remove non-alphabetical characters, unify country names
4. **Visualize** results as bar plots

Extract country and institution

The information about the author's country and institution of affiliation is located under *metadata/authors/affiliation/location* in the nested **JSON structure**.

```
def extract_country(filenames):
    with open(filenames, 'r') as f:
        data = json.load(f)
    authors = (
        data
        .get("metadata", {})
        .get("authors", [])
    )

    countries = []
    for author in authors:
        country = (author.get("affiliation", {})
                .get("location", {})
                .get("country"))
        if country is not None and country != "":
            countries.append(country)
```

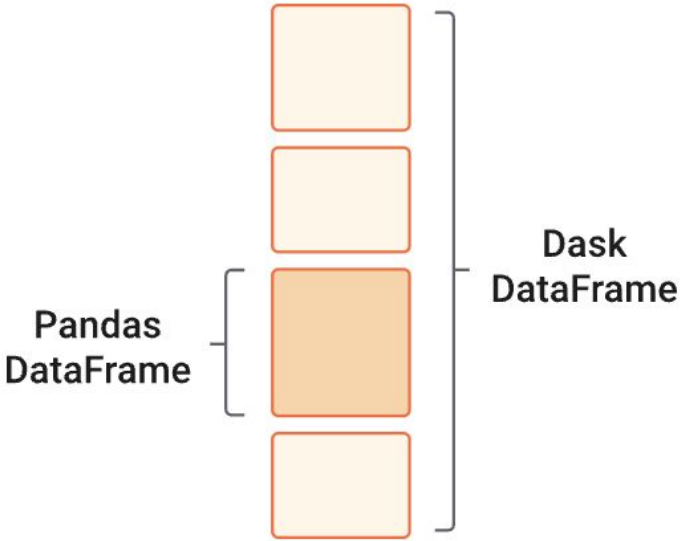
We iterate over all authors in a paper to extract their country and institution of affiliation.

Result: 2 lists

```
return countries , institutions
```

Dask DataFrame

A **Dask DataFrame** is a collection of multiple pandas DataFrames called partitions. It is useful when dealing with larger-than-memory datasets.



From Dask Bag to DataFrame

From Dask Bag to DataFrame : We use Dask Bag to ingest data from the JSON files and preprocess it into Dask DataFrames.

```
npartitions = 50
json_df = (
    db.from_sequence(filenames, npartitions=npartitions)
      .map(extract_country)
      .to_dataframe(columns=["country", "institutions"])
)
```

We **map** the extract_country function over all the elements of the Bag. Then, we convert the Dask Bag to a DataFrame

Data Cleaning

We perform **data cleaning** to unify different country name variations and remove non-alphabetical characters.

We create a **dictionary** that maps alternate country names to standardized ones.

country_clean	count
United States	23682
United States of America	21021
The United States of America	65
Unites States	40
United States of America A R	15
United States A R	14

```
# country mapping
country_mapping = {
    'USA': 'United States',
    'United States of America': 'United States',
    'US': 'United States',
    'USA, USA': 'United States',
    'Usa': 'United States',
    'Alabama': 'United States',
    'New Jersey': 'United States',
```

Data Cleaning & Transformation

We define a function to **explode** the two lists, **clean** the country names and **remove** whitespace from the institutions column.

```
def process_and_clean(pdf):
    # Explode both columns
    pdf = pdf.explode('country')
    pdf = pdf.explode('institutions')
    pdf = pdf[pdf['institutions'].fillna('').str.strip() != '']
    pdf['country_clean'] = pdf['country'].replace(country_mapping)
    pdf['country_clean'] = pdf['country_clean'].astype(str)
    mask = (
        pdf['country_clean'].notna() &
        (pdf['country_clean'] != 'nan') &
        pdf['country_clean'].str.match(r'^[a-zA-Z\s]+$', na=False)
    )
    pdf = pdf[mask]

    return pdf
```

Apply & Persist

We **apply** all transformations at once on all the partitions for better graph optimization.

```
json_df = json_df.map_partitions(process_and_clean)
```

Then we use **persist** to start the execution in the background and store the results in memory, reducing computational time for subsequent operations.

```
json_df = json_df.persist()
```

Count & GroupBy

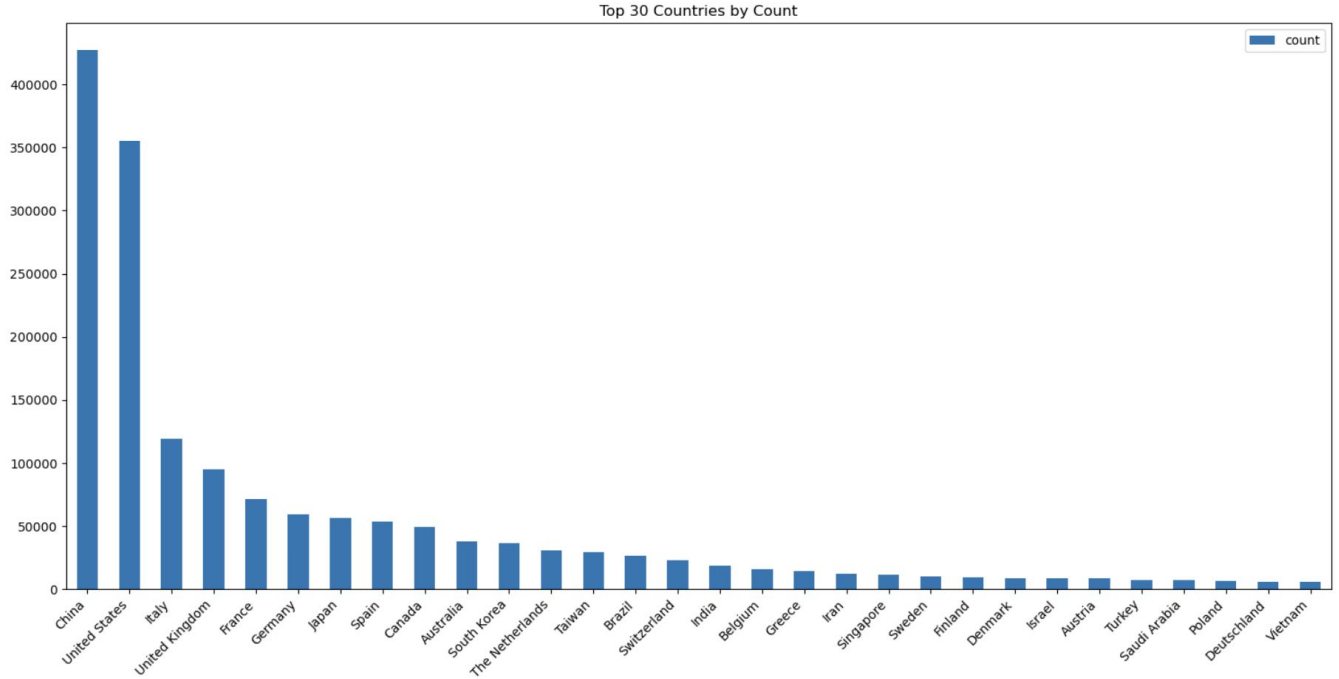
We perform a **count** and a **group by** for institutions and country columns.

```
country_counts = (
    json_df.groupby("country_clean")
           .size(split_out=3)
           .to_frame('count')
           .reset_index()
           .sort_values('count', ascending=False)
           .compute()  # Compute
)
```

We use **split_out** to enable greater parallelism during the aggregation phase. It divides groups into multiple output partitions, reducing shuffle bottlenecks and balancing the workload.

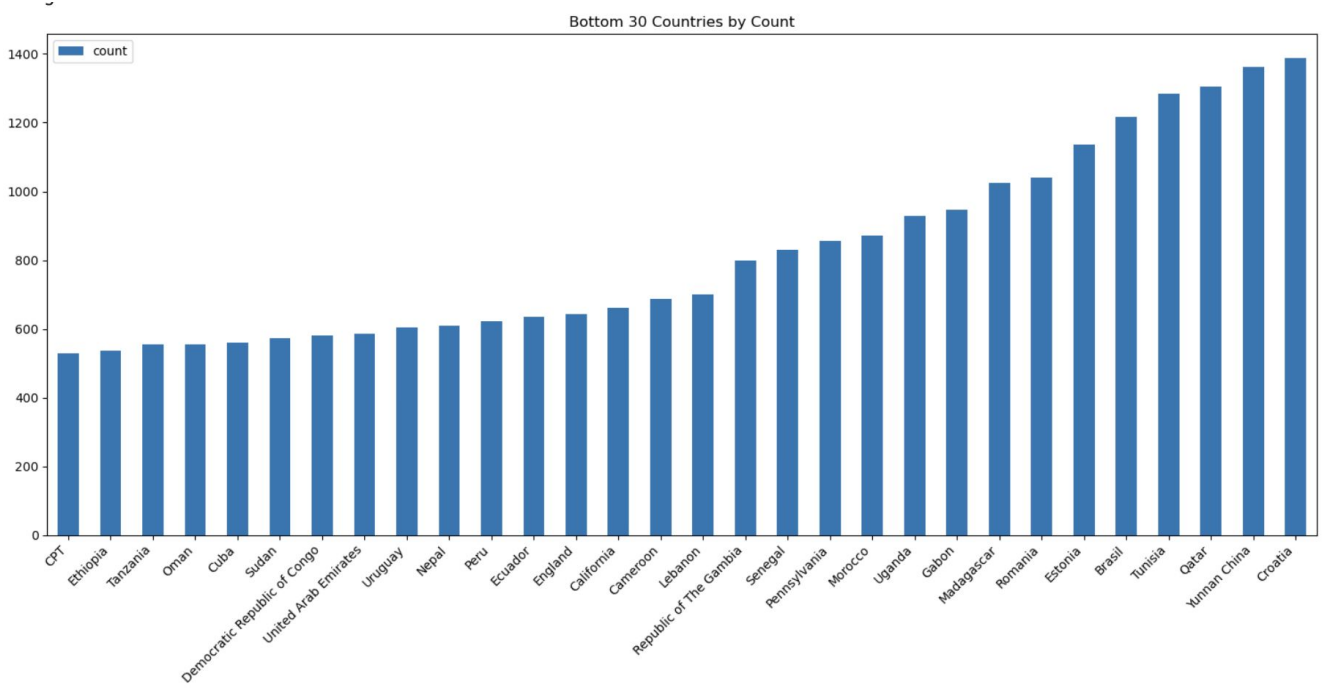
Top 30 Countries

This bar chart displays the top 30 represented institutions. China and the United States lead global research efforts on COVID-19.



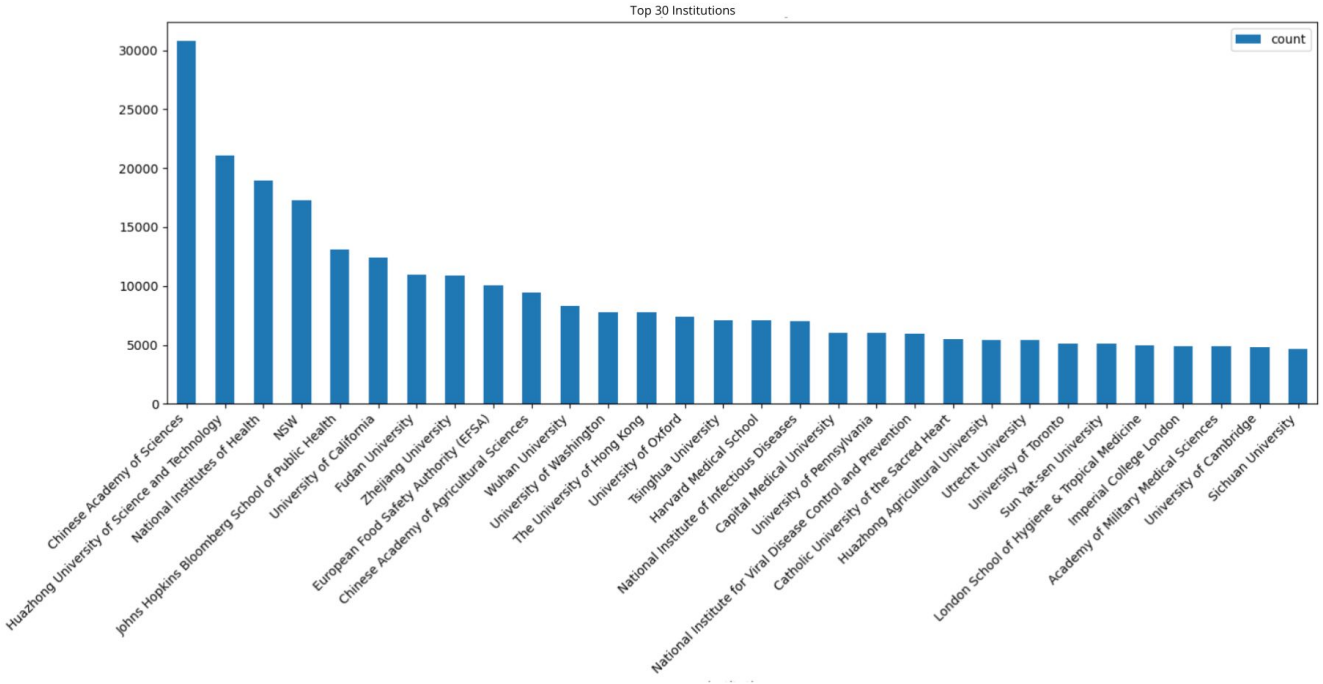
Bottom 30 Countries

Here we show the bottom 30 countries.



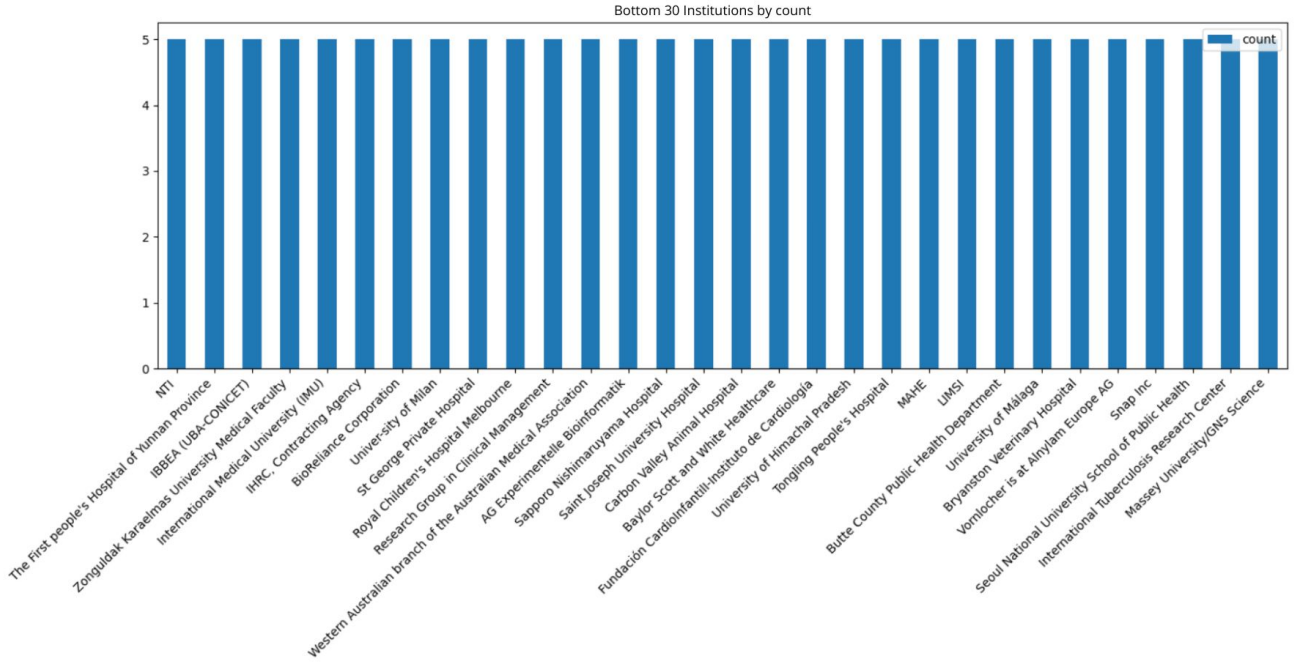
Top 30 Institutions

Here we can view the top 30 Institutions



Bottom 30 Institutions

Then the bottom 30

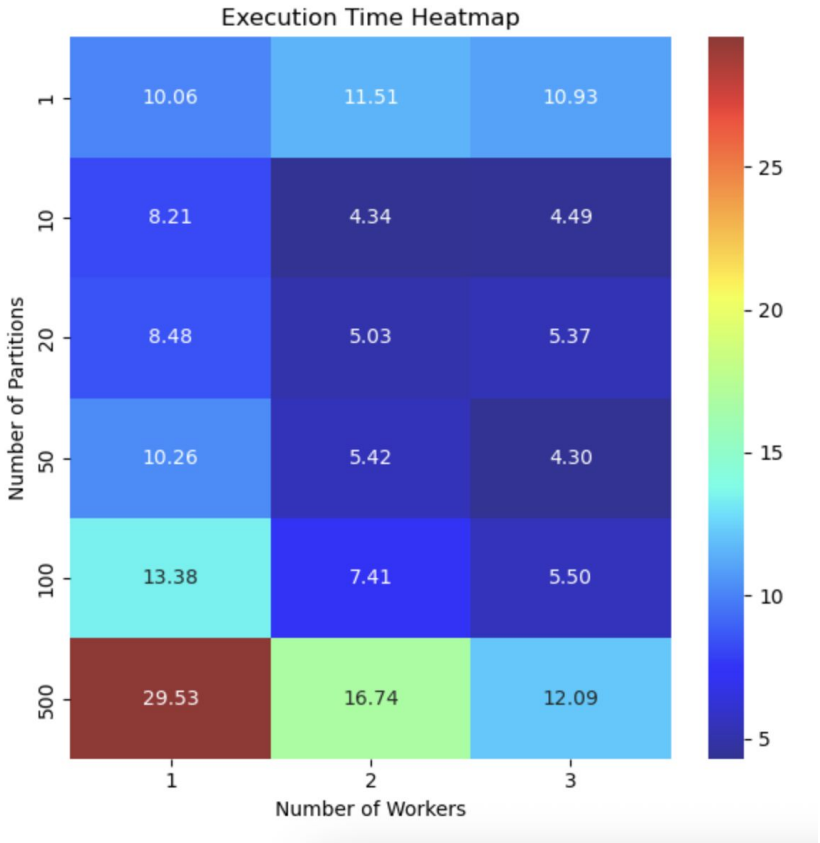


Grid search on speed

We once again performed a Benchmark testing the Map-Reduce phases using different configurations in terms of **Partitions** and **Workers**

We use a subset of **10,000 files** to not **overload** when testing with just a **single worker.**

The results confirm that the cluster works best in the range **10-100 partitions** and **3 workers.**



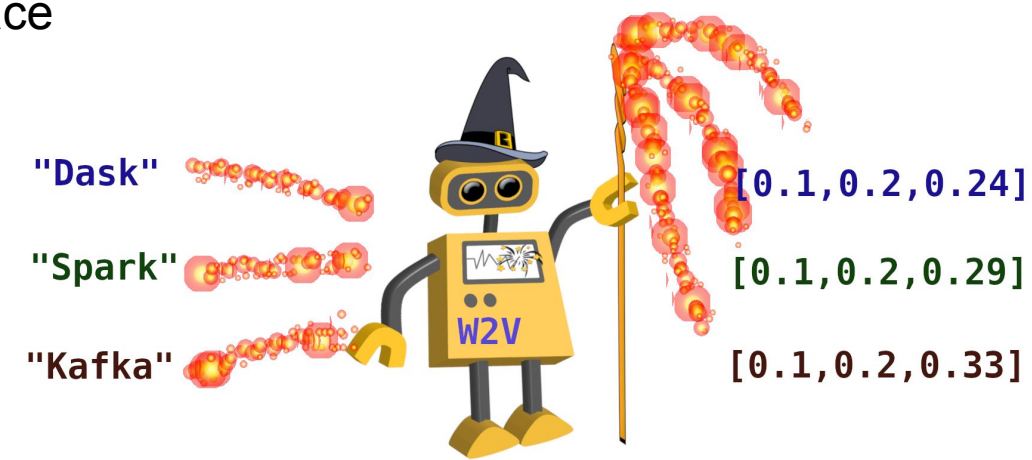
Task 3:
Obtaining embedding for paper titles

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

NLP & word2vec

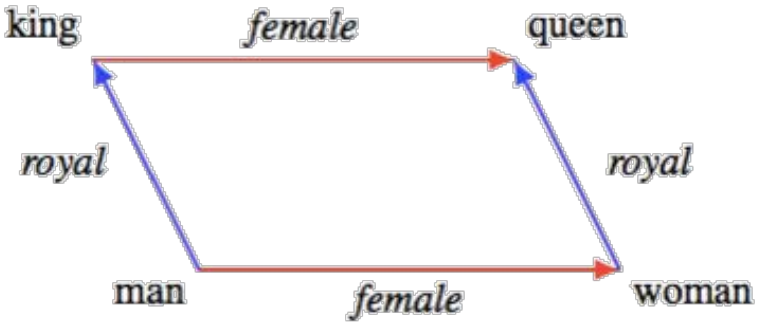
NLP is an area of AI focused on enabling machines to understand and process human language.

Word2Vec revolutionized the field by demonstrating that complex semantic relationships could be learned automatically and represented as mathematical directions in space

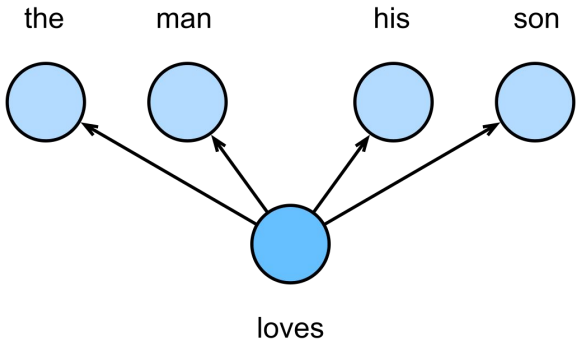


Word Embeddings

- They encode a word's **meaning** into a set of **coordinates**, placing it into a *meaning-space.*
- -> measure relationship between words in a mathematical way.
- King-Man+Woman = Queen



- Pre-trained fastText model
- Trained on Wikipedia using Skip-gram approach.
  - Represent each word with a vector -> to predict the vectors of its neighbors.



Approach

1. We used `metadata.csv` file to retrieve titles and paper-id (dask.DataFrame).
2. Tokenize all titles to retrieve the list of **unique words** (dd.DataFrame)
3. Load the most complete **FastText** english **model** (wiki.en.vec, 6.1GB) (dask.Bag -> dask.DataFrame)
4. **Inner joined** the list with the words of the model (dask.Dataframe)
5. **Mapped** vectors to title's words by merging dataframes. (dask.Dataframe)
6. **Grouped** words **by** paper id to retrieve list of vectors. (dask.Dataframe)

We triggered computation just in the last step and let the dataframe **persist** in the cluster since it was too big to be handled by the scheduler.

Code

Words from titles linked to paper ID

Filtered fastText model

```
the_word_and_friends = title_pid.merge(filtered_wiki, how='left', left_on='title',right_on='word') # merging
the_word_and_the_vec = the_word_and_friends[['pid','vec']].copy() # some cleaning/refining
the_word_and_the_vec['vec'].apply(list_or_nothing, meta=('vec', 'object'))
del the_word_and_friends # memory management
client.run(gc.collect)
```

Memory management

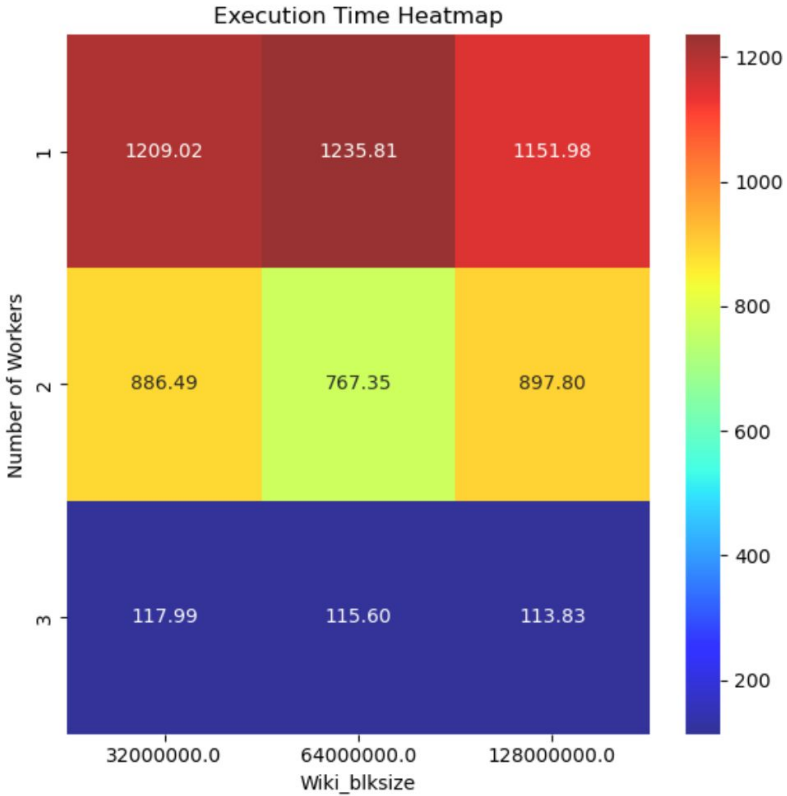
Helper functions

Benchmark

We ran a **benchmark** over the number of workers and Wiki_blksize parameter, which regulates the size of the chunks that we use to analyze the model

We **kept fixed** the partitions of titles and the partitions of the filtered wiki, since we thought they were the **least influential**

Clearly, the **number of workers** is the parameter that is most influential in terms of runtime



Computation results

We setup the cluster with the best parameters found, then we use it to **compute** ids and vectors relative to titles:

```
print(pid[0])
print(vectors[0])
```

```
00d1165856c978d9b09bee5e0a1fbca063df6c4a
[array([-3.1533e-02,  4.6278e-02, -1.2534e-01,  1.9165e-01, -1.2660e-01,
       -1.2853e-02,  1.0342e-01, -9.8085e-03,  1.5189e-01,  2.7582e-01,
        1.3695e-01,  8.8799e-03,  1.4132e-01, -1.2000e-01, -6.3439e-02,
       -1.5178e-01,  9.8090e-02, -1.2010e-01, -6.9086e-02,  1.4666e-02,
       -2.3041e-02,  3.0430e-02, -1.2664e-01, -6.3282e-02, -8.2246e-02,
        3.6718e-02,  2.2698e-01, -9.6025e-02, -1.1699e-02,  6.6158e-02,
       -1.8542e-01,  1.9223e-01, -6.1685e-02,  2.7049e-01,  7.5116e-02,
       -5.4928e-02, -8.6027e-02, -1.9387e-01,  1.4677e-01, -6.0130e-02,
        6.8269e-02,  7.1613e-02, -9.4414e-02,  3.6158e-02,  2.7820e-03,
       -8.1711e-02, -1.3369e-02, -5.3017e-02,  5.2227e-02, -7.9682e-02,
       -3.1768e-04,  3.0397e-02, -1.6847e-01,  2.1828e-02, -1.9577e-01,
       -5.0109e-02, -9.6879e-03,  8.5536e-02, -2.8135e-01,  1.7001e-01,
       -4.9194e-02, -1.6721e-01,  1.9018e-01, -4.7400e-02, -3.6412e-04,
        2.6316e-02, -2.2135e-01, -6.1583e-02, -2.1854e-01, -2.1669e-02,
       -2.9630e-01, -7.1949e-02,  1.0638e-02, -1.9055e-01, -1.1292e-01,
```

Then we matched every **pid** with the corresponding original title:

```
print('Obtained IDs:',len(pid))
print('Obtained titles:',len(titles))
print('Obtained Vector lists:',len(vectors))
```

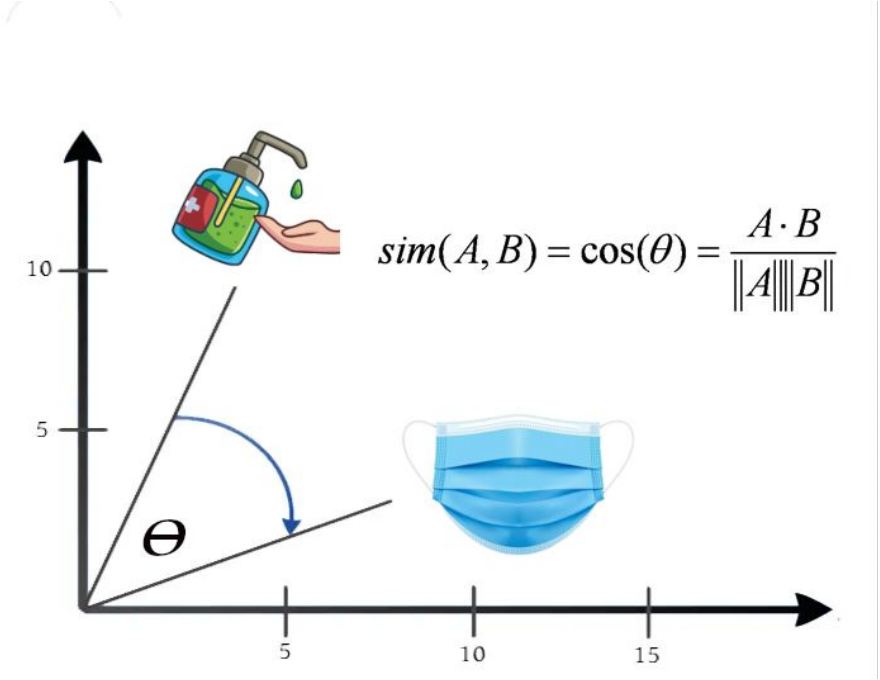
```
Obtained IDs: 1773
Obtained titles: 1773
Obtained Vector lists: 1773
```

Task 4:
Cosine similarity computation

What is Cosine Similarity

- It's a **measure** of similarity between two vectors
- Words with **similar context** occupy close spatial positions.
- The cosine of the **angle** between such vectors should be close to 1 -> angle ~ 0° (**perfect similarity**)
- If the value is -1 we have **perfect dissimilarity** instead

We can use this metric to retrieve the **most similar titles** in our dataset



Limitations

For this part we took only a **subset** of our original dataset. This is due to the fact that, to compute:

```
def cosine_similarity(a, b):
    num = np.dot(a.T, b)
    den = (np.dot(a.T, a) * np.dot(b.T,b))**(1.0/2)
    if den == 0:
        return 0.0
    return num / den
```

we are forced to early **compute** or **persist**, to get **numpy** objects. This leads us to an excessive use of memory that we cannot afford

```
sampled_raw = the_word_and_the_vec.sample(frac=0.002)
sampled_pids = set(sampled_raw['pid'].compute().tolist())
```

This task scales badly with **n**

By selecting a sample of the dataset we are allowed to perform this task both in **parallelized** and **serialized** approach (numpy is able to handle a few files, but not all of our dataset). We will then **compare the** performances

Computation: serial

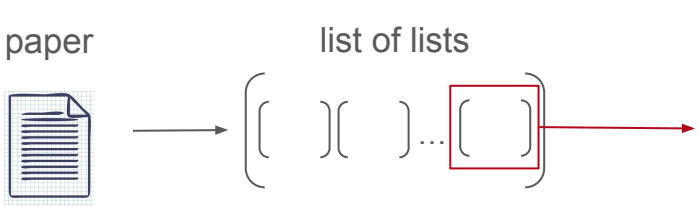
We first tried **serial**

We discarded all elements of len = 0 (missing vector)

```
for i in range (0, len(vectors)):
    val = vectors[i]
    # keeping not None vectors and with len > 0
    if len(val) > 0:
        kept.append(val)
        count = count + 1
```

```
Original length: 1773
Kept entries after skirming empty: 1773
```

Then we had to come up with a plan to compute the similarity. The structure of our data is the following:



vector

```
[array([-3.1533e-02,  4.6278e-02, -1.2534e-01,  1.9165e-01, -1.2660e-01,
        -1.2853e-02,  1.0342e-01, -9.8085e-03,  1.5189e-01,  2.7582e-01,
         1.3695e-01,  8.8799e-03,  1.4132e-01, -1.2000e-01, -6.3439e-02,
        -1.5178e-01,  9.8090e-02, -1.2010e-01, -6.9086e-02,  1.4666e-02,...
```

Computation: serial

If we want to compute similarity between two titles, we have to **iterate** over all couples of **vectors** in those titles

If we want to compute similarity between all titles, we have to **iterate** over all couples of **titles** in the selected dataset

We will use a **nested for loop** to handle the iterations and put the results in a matrix, averaging over full papers

Note that the resulting matrix is **symmetric** (similarity between papers *k* - *l* is same as *l* - *k*): we can preserve resources by computing only the **upper half**

```
for k in (range(n_entries)):
    for l in range(k):

        m_a = kept[k]
        m_b = kept[l]
        m_ab = np.zeros((len(m_a), len(m_b)))

        for i in range (0, len(m_a)):
            for j in range (0, len(m_b)):
                va = m_a[i]
                vb = m_b[j]

                # skip if none
                if va is None or vb is None:
                    continue
                va = np.array(va)
                vb = np.array(vb)
                m_ab[i][j] = cosine_similarity(va.T, vb.T)

        res_matrix[l][k] = m_ab.mean()
```

Computation: serial

We convert the matrix into a **list**, retrieving the coordinates of highest similarities (that will match the index of similar papers). We printed the top results:

```
Coordinates: (764, 1077) Value: 0.9723288
Coordinates: (369, 1323) Value: 0.96707714
Coordinates: (1270, 1463) Value: 0.95814687
Coordinates: (1418, 1463) Value: 0.95814687
Coordinates: (995, 1010) Value: 0.95786285
```

```
First:  A Human PrM Antibody That Recognizes a Novel Cryptic Epitope on Dengue E Glycoprotein
Second:  Sequence Comparison of Avian Infectious Bronchitis Virus S1 Glycoproteins of the Florida Sero
type and Five Variant Isolates from Georgia and California

First:  Patient characteristics and severity of human rhinovirus infections in children
Second:  Evidence of Recombination and Genetic Diversity in Human Rhinoviruses in Children with Acute
Respiratory Infection

First:  Immunization with Live Human Rhinovirus (HRV) 16 Induces Protection in Cotton Rats against HRV
14 Infection
Second:  Evidence of Recombination and Genetic Diversity in Human Rhinoviruses in Children with Acute
Respiratory Infection

First:  Protein-Protein Interactions of Viroporins in Coronaviruses and Paramyxoviruses: New Targets f
or Antivirals?
Second:  Characterization of monoclonal antibody against SARS coronavirus nucleocapsid antigen and dev
elopment of an antigen capture ELISA

First:  Protein-Protein Interactions of Viroporins in Coronaviruses and Paramyxoviruses: New Targets f
or Antivirals?
Second:  Preventive Behavioral Responses to the 2015 Middle East Respiratory Syndrome Coronavirus Outb
reak in Korea
```

Computation: parallel

We now want to compute the same task but in a **parallelized setting**

We cannot do much: we **assign the computation** of the function to the client and that's it

This leads the cluster to perform a **very high number** of single simple operations

For time reasons, we computed the similarities only for a **smaller subset** (1/100 of the previous subset)

```
Time needed for computation in parallel (dataset / 10): 78.25245428085327 s
Time needed for computation in serial: 17.647429943084717 s
```

For this task specifically, the **serial approach** is far better than the **parallelized approach**

Thank you!

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Backup: Secure SHell

What it is:

- Cryptographic network protocol for operating network services securely over an unsecured network.

Uses:

- Remote login, secure file transfer (scp), port forwarding.

Security Features:

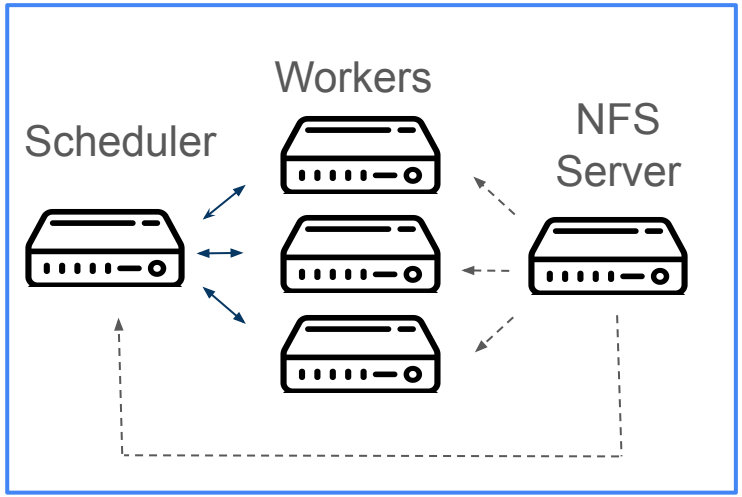
- Confidentiality, Integrity, Authentication.

```
#~/.ssh/config
Host cloudveneto
        HostName gate.cloudveneto.it
        User lbelli
        IdentityFile ~/.ssh/[redacted]

Host scheduler
        HostName 10.67.22.173
        User ubuntu
        IdentityFile ~/.ssh/[redacted].pem
        Proxyjump cloudveneto

Host sched-notebook
        HostName 10.67.22.173
        User ubuntu
        IdentityFile ~/.ssh/[redacted].pem
        Proxyjump cloudveneto
        LocalForward 8888 localhost:8888
```

Backup: NFS

Network File System

- DFS
- 1984, with updates
- Allows user on client access files over network like local storage.
- Our implementation was: transparent, not so scalable, not fault tolerant.



Backup: Task 4 with Dask vectors

What if **cosine similarity** was computed with **Dask vectors**?

```python
def to_stackable_array(df):
    arr = np.stack(df['vec'].values)
    return da.from_array(arr, chunks=(100, arr.shape[1]))
```

```python
norms = da.linalg.norm(vectors, axis=1, keepdims=True)
X_norm = vectors / norms
X_norm = X_norm.rechunk((1000, 300))
cos_sim_matrix = (X_norm @ X_norm.T).compute()
```

Dask implements data structures similar to Numpy arrays: we can exploit them to compute everything in cluster

This leads to **faster** computing times:

```
Total time needed for parallelized version: 238.369s
Array creation time: 238.221s
Matrix computation time: 0.148s
```

But this still scales very badly with **n**, not allowing us to perform any consideration on the results for larger datasets

In the end, using Dask parallelized vectors or Dataframes with serial computation lead to **similar results** both in terms of **time** and **scalability**