1 Problem Formulation

1.1 Proposed problem(s) to be solved based on the selected dataset.

The COVID-19 pandemic spreads very rapidly and is lethal. It is necessary to keep track of number of infections, where they occur, and number of hospitalizations. The real-time data to accomplish the above, dataset prepared by the Our World in Data organization [1].

Table 1. Description of relevant columns in the COVID dataset

No	Name of Column	Type	Description
1	location	String	Country
2	continent	String	Continent of country
3	date	Date	Date of observation (yyyy-mm-dd)
4	median_age	Numeric	Median age of the population
5	total_cases	Numeric	Total confirmed cases
6	new_cases_per_million	Numeric	New confirmed cases per million
7	new_deaths_per_million	Numeric	New deaths per million
8	total_deaths_per_million	Numeric	Total deaths per million
9	new_vaccinations	Numeric	New vaccination doses administered
10	hosp_patients_per_million	Numeric	Number of COVID patients in hospital
11	gdp_per_capita	Numeric	Gross Domestic Product per Capita

1.2 Justification for Big Data analytics

Big data processes huge *volumes* of data. The COVID pandemic involves thousands of new infections, hundreds of hospitalizations, across 223 countries. To study which country did COVID came from, effectiveness of vaccination, it is necessary to analyse many rows of data.

Big data processes large data quickly (*velocity)* important since the COVID pandemic is dynamic. New variants crop up, upping infection cases from previous lows, necessitating new lockdowns. Data analysts need to quickly spot the uptick in infection, or effectiveness of vaccines, and inform relevant agencies.

Finally, big data needs to facilitate data cleaning (*veracity*). Big data need to provide high-level tools which facilitates quick and accurate data cleanup. Clean data is important for accurate analysis.

2. Conventional technique

Pandas is the most popular library for data analysis in Python [2]. Pandas is robust for text (CSV and delimited), Microsoft Excel, JSON, and HDF5 [3]. Pandas offers high-level functionality for crucial functions like joining, merging, indexing, creating derived variables, summary statistics, and time-series analysis. Last, Pandas is very fast in processing large datasets, because its back-end operation is written in C or Cython [2].

The *import* statement was used to search for *pandas* library and associate it with name *pd* [5]. The same is done for *matplotlib.pyplot* interface, and *warnings* library (**Figure 1)**. Next, the csv file is loaded as a Pandas DataFrame (**Figure 2**) using *pandas*, *read_csv* function.

```
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
import matplotlib.pyplot as plt
```

Figure 1. Importing pandas, ignore warnings, and importing matplotlib (visualization).

```
file = 'owid-covid-data.csv'
df = pd.read_csv(file)
```

Figure 2. Loading dataset with pandas

In **Figure 3**, the type of relevant columns is revealed using the *pandas DataFrame, dtype* method. It checks if the column type is amenable for further operations. The *continent*, *location*, and *date* columns are *strings*, while others are *numeric* – good for statistical analysis. Then, in **Figure 4**, the date column is converted from *string* to *date* type, using the*, to_datetime* method, for time-series analysis.

```
df[['location',
    'continent',
    'date',
    'total_cases',
    'new_cases_per_million',
    'new_deaths_per_million',
    'total_deaths_per_million',
    'new_vaccinations',
    'hosp_patients_per_million',
    'people_vaccinated_per_hundred',
    'gdp_per_capita']].dtypes
```

```
location                          object
continent                         object
date                              object
total_cases                      float64
new_cases_per_million            float64
new_deaths_per_million           float64
total_deaths_per_million         float64
new_vaccinations                 float64
hosp_patients_per_million        float64
people_vaccinated_per_hundred    float64
gdp_per_capita                   float64
dtype: object
```

Figure 3. Revealing type of each relevant column using *dtypes*/method.

```
df['date'] = pd.to_datetime(df['date'],format='%Y-%m-%d')
```

Figure 4. Converting *date* column to *date* format using *to_datetime* method.

Pandas has robust sub-setting, sorting, and list-compatibility methods. To find, origin of COVID-19, we paired each country with the earliest date when total cases exceeded 1,000. First, the DataFrame is filtered by *total_cases* exceeding 1,000. Second, each country subset is sorted by date (*sort_values* method), and the earliest date per country is dumped into another DataFrame. Finally, the top 10 countries (earliest), and the bottom 10 (latest), are printed, using the *head* and *tail* method respectively. The code is depicted in **Figure 5**.

```
"""
1. Where did COVID-19 begin?
"""
df_short = df[ df['total_cases'] > 1000 ]
countries=df_short['location'].sort_values().unique()
countries=[countries[i] for i in range(len(countries))]
COUNTRY = []
DATE = []
for i, country in enumerate(countries):
    df_new = df_short[
             df_short['location'] == country
    ].sort_values('date').head(1)
    COUNTRY.append(country)
    DATE.append(df_new.iloc[0]['date'])
country_date = pd.DataFrame()
country_date['Country'] = COUNTRY
country_date['Date'] = DATE
print(country_date\
.sort_values('Date')\
.head(10)\
.reset_index(drop=True))
print(country_date\
.sort_values('Date')\
.tail(10)\
.reset_index(drop=True))
```

Figure 5. Display earliest, and latest countries with COVID-19 cases more than 1,000

Pandas can calculate the average, or sum, of variable, *to_aggregate*, based on grouping by another variable, *aggregate_by*. For example, the *new_cases* variable across each country, can be averaged or summed, by *date* variable. To do so, the *groupby* method is used, followed by the *mean,* or *sum* method. This is illustrated in the *aggregate* function. The input is a *DataFrame,* followed by variable used to group, *aggregate_by,* and the variable to be *sum* or *averaged*, named *to_aggregate*. **Figure 6.** shows the *aggregate* function. It returns the list of *aggregate_by and* corresponding *to_aggregate* values.

```
def aggregate ( dataframe,
                aggregate_by,
                to_aggregate,
                method ):

  if method == 'sum':

    df_short = dataframe\
             .sort_values(aggregate_by)\
             .groupby(aggregate_by)\
             [[to_aggregate]]\
             .sum()\
             .reset_index()

  elif method == 'avg':

    df_short = dataframe\
             .sort_values(aggregate_by)\
             .groupby(aggregate_by)\
             [[to_aggregate]]\
             .mean()\
             .reset_index()
  else:

    print('No valid method')

    return

  X = [df_short.iloc[i][aggregate_by]
       for i in range(len(df_short))]

  Y = [df_short.iloc[i][to_aggregate]
       for i in range(len(df_short))]

  return X, Y
```

Figure 6. Aggregation function (Pandas)

3. Big data technique (PySpark)

The PySpark library is an interface which allows Python users to access Apache Spark. The Apache Spark is an open-source engine for big-data workloads across multiple computers. The advantages of Apache Spark are:

i. Fast because processes happen in memory [6]
ii. Fault tolerance from Directed Acyclic Graph (DAG) architecture [7].

First, the *findspark* and *pyspark* libraries are imported (**Figure 7**).

```
import findspark

"""
The SparkSession class is the point-of-entry for all Apache Spark functionalities.
Import various Apache Spark built-in-functions like
  count - An aggregate function that returns the number of elements  in a group[4]
  sum   - An aggregate function that returns the sum of all elements in a group[5]
  avg   - An aggregate function that returns the average of all elements in a group[6]
"""

from pyspark.sql import SparkSession
from pyspark.sql.functions import count, when, sum, avg
```

Figure 7. Importing *findspark* and *pyspark* libraries.

Next, a Spark session is initialized (**Figure 8**). Subsequently, the *read.csv* method is used to load dataset (**Figure 9**). Next, types of selected columns are identified (**Figure 10**).

```
findspark.init()  # Allows Spark to be imported as a regular library [7]

# Get or create Spark session with name 'Covid Data Mining'
spark = SparkSession.builder.appName('Covid Data Mining').getOrCreate()
```

Figure 8. Initialize Spark session.

```
file = 'owid-covid-data.csv' # name of csv file with COVID data

df = spark.read.csv(file,header=True,inferSchema=True)
```

Figure 9. Load dataset.

```
df.select(['location',
           'continent',
           'date',
           'total_cases',
           'new_cases_per_million',
           'new_deaths_per_million',
           'total_deaths_per_million',
           'new_vaccinations',
           'hosp_patients_per_million',
           'people_vaccinated_per_hundred',
           'gdp_per_capita']).printSchema()
```

```
root
 |-- location: string (nullable = true)
 |-- continent: string (nullable = true)
 |-- date: date (nullable = true)
 |-- total_cases: double (nullable = true)
 |-- new_cases_per_million: double (nullable = true)
 |-- new_deaths_per_million: double (nullable = true)
 |-- total_deaths_per_million: double (nullable = true)
 |-- new_vaccinations: double (nullable = true)
 |-- hosp_patients_per_million: double (nullable = true)
 |-- people_vaccinated_per_hundred: double (nullable = true)
 |-- gdp_per_capita: double (nullable = true)
```

Figure 10. Type of selected columns using *printSchema* method for PySpark.

PySpark has high-level tools as follows:

i. *select* method for choosing columns.

ii. *filter* method for sub-setting by condition.

iii. *sort* and *orderBy* method for sorting by variable.

iv. *groupBy* and *agg* method for aggregation.

v. *sum* and *avg* for sum and average operations respectively.

The usage of PySpark for analysing COVID data are in **Figures 11** and **12**.

```
"""
1. Where did COVID-19 begin?
"""
df_short = df\
.select('location','date','total_cases')\
.filter(df['total_cases']>1000)

countries = df_short.select('location').distinct().collect()
countries = [countries[i][0] for i in range(len(countries))]

COUNTRY = []
DATE = []
for i, country in enumerate(countries):
    # Filter by country, sort by date, take the earliest date
    df_new = df_short\
        .filter(df_short['location']==country)\
        .sort('date').head(1)
    # Save name of coutry, and earliest date where
    #   total_cases > 1000, to COUNTRY and DATE lists
    COUNTRY.append(country)
    DATE.append(df_new[0]['date'])
    print(f'{i},{len(countries)}')

spark_country_date = SparkSession\
    .builder.appName('CountryDate').getOrCreate()
columns = ['Country','Date']
country_date = spark_country_date\
    .createDataFrame(data=zip(COUNTRY,DATE),schema=columns)

print(country_date.orderBy('Date').show(10))

print(country_date.orderBy('Date',ascending=False).show(10))
```

Figure 11. Code to show where did COVID-19 begin. The top 10, earliest and latest country, where COVID-19 exceeding 1,000 cases, are printed with the code.

```
def aggregate ( dataframe,
                aggregate_by,
                to_aggregate,
                method ):

  if method == 'sum':

    df_short = dataframe\
             .orderBy(aggregate_by)\
             .groupBy(aggregate_by)\
             .agg(sum(to_aggregate))\
             .collect()

    new_to_aggregate = f'sum({to_aggregate})'

  elif method == 'avg':

    df_short = dataframe\
             .orderBy(aggregate_by)\
             .groupBy(aggregate_by)\
             .agg(avg(to_aggregate))\
             .collect()

    new_to_aggregate = f'avg({to_aggregate})'

  else:

    print('No valid method')
    return

  X = [df_short[i][aggregate_by]
       for i in range(len(df_short))]

  Y = [df_short[i][new_to_aggregate]
       for i in range(len(df_short))]

  return X, Y
```

Figure 12. Code to average or sum variable, *to_aggregate*, based on grouping by variable, *aggregate_by*. It returns a list of values, *aggregate_by*, and another list of values, *to_aggregate*.

4. Results

The following *matplotlib* code to plot *time-series*, *scatterplot,* and *double-y-axis*, is depicted in **Figure 13, 14,** and **15** respectively.

```
def time_series(TITLE,
                YLABEL,
                XLABEL,
                PNGFILE,
                X,
                Y):

  plt.figure(figsize=(25,15))
  plt.plot(X,Y,linestyle='-',color='b')
  plt.title(TITLE,fontsize=24,fontweight='bold')
  plt.xticks(fontsize=18,rotation=45)
  plt.yticks(fontsize=18)
  plt.ylabel(YLABEL,fontsize=24,fontweight='bold')
  plt.xlabel(XLABEL,fontsize=24,fontweight='bold')
  plt.savefig(PNGFILE,bbox_inches='tight')
```

Figure 13. Code to plot time-series.

```
def scatter(TITLE,
            YLABEL,
            XLABEL,
            PNGFILE,
            X,
            Y):

  plt.figure(figsize=(25,15))
  plt.scatter(X,Y)
  plt.xlabel(XLABEL,fontsize=24,fontweight='bold')
  plt.ylabel(YLABEL,fontsize=24,fontweight='bold')
  plt.title(TITLE,fontsize=24,fontweight='bold')
  plt.xticks(fontsize=24)
  plt.yticks(fontsize=24)
  plt.savefig(PNGFILE)
```

Figure 14. Code to plot scatter.

```
def double_y (TITLE,
              Y1LABEL,
              Y2LABEL,
              XLABEL,
              PNGFILE,
              X,
              Y1,
              Y2):

  fig,ax1 = plt.subplots(figsize=(25,15))
  ax1.plot(X,Y1,linestyle='-',color='r')
  ax1.set_ylabel(Y1LABEL,fontsize=24,
                 fontweight='bold',color='r')
  ax1.set_xlabel(XLABEL,fontsize=24,
                 fontweight='bold',color='k')
  ax1.tick_params(axis='y',color='r')
  ax1.set_title(TITLE,fontsize=24,fontweight='bold')
  ax1.tick_params(axis='x', labelsize=20)
  ax1.tick_params(axis='y', labelsize=20)

  ax2 = ax1.twinx()
  ax2.plot(X,Y2,linestyle='--',color='b')
  ax2.set_ylabel(Y2LABEL,fontsize=24,
                 fontweight='bold',color='b')
  ax2.tick_params(axis='y',color='b')
  ax2.tick_params(axis='x', labelsize=20)
  ax2.tick_params(axis='y', labelsize=20)

  ax2.spines['left'].set_color('r')
  ax2.spines['right'].set_color('b')

  plt.savefig(PNGFILE,bbox_inches='tight')
```

Figure 15. Double-y-axis code.

Based on the codes above, both PySpark and Pandas were separately used to derive the same results.

1. Where did COVID-19 begin?

Refer to code in **Figure 5** (Pandas) and **Figure 11** (PySpark). The results are in **Figure 16**.

```
+--------------+----------+
|       Country|      Date|
+--------------+----------+
|         China|2020-01-25|
|   South Korea|2020-02-26|
|         Italy|2020-03-01|
|          Iran|2020-03-03|
|       Germany|2020-03-08|
|         Spain|2020-03-08|
| United States|2020-03-11|
|   Switzerland|2020-03-12|
|        Norway|2020-03-13|
|United Kingdom|2020-03-13|
+--------------+----------+
only showing top 10 rows
```

Figure 16. Top ten countries earliest to exceed 1,000 COVID-19 cases.

The first country to attain 1,000 COVID-19 cases was China, followed by South Korea. One week later, Europe followed by the United States attained 1,000 COVID-19 cases.

2. How fast did COVID-19 spread ?

New COVID-19 cases of each country are summed up and grouped by each day (**Figure 17**). Time-series are plotted (**Figure 18**).

```
X, Y = aggregate(df,'date','new_cases_smoothed','sum')

time_series('Total New COVID Cases (Smoothed)vs. Date',
            'Total New Cases (Smoothed) / Millions',
            'Date (Daily)',
            'Total_New_Cases_vs_Date.png',
            X,
            Y)
```

Figure 17. Code for aggregating sum of new cases (Y) vs. date (X)

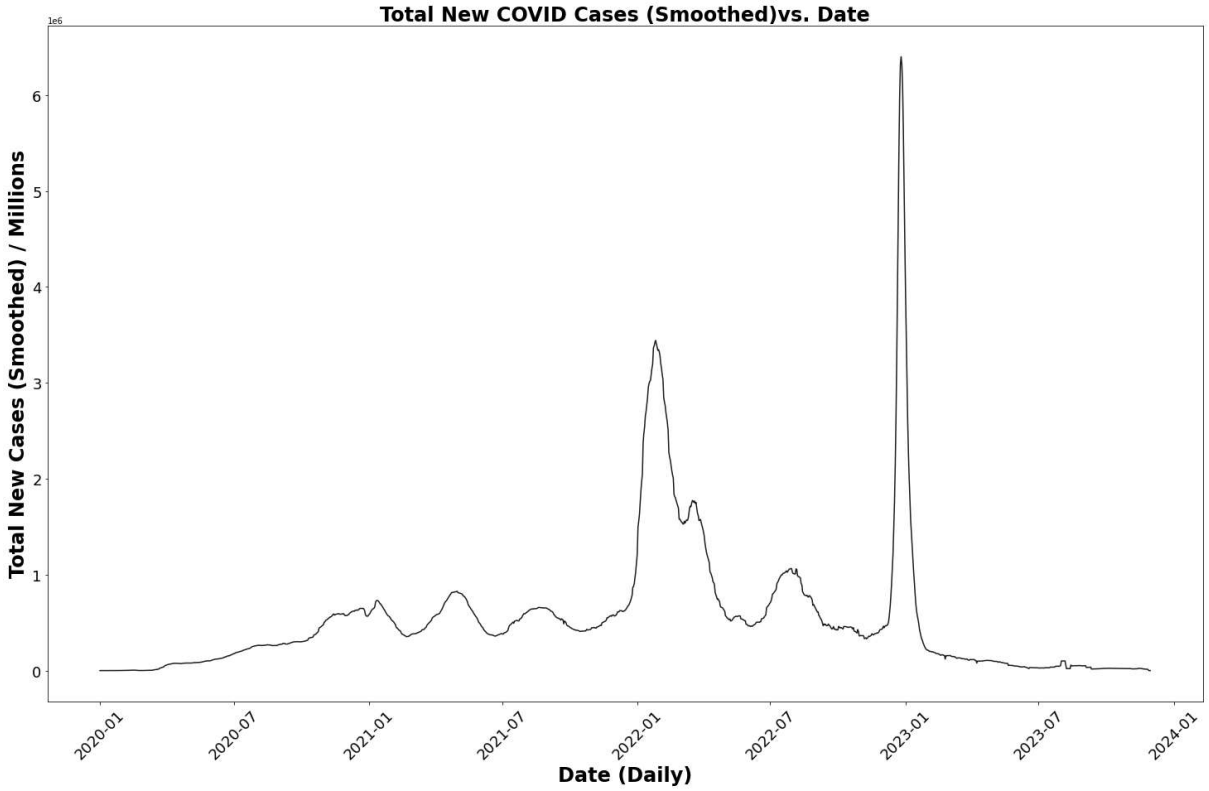

Figure 18. Time-series of total new cases vs. date (daily)

3. Impact of vaccination on COVID-19 total cases

Figure 19 depicted the code, while **Figure 20** is the plot, to compare number of vaccinations vs. new COVID-19 cases.

```
X, new_cases_smoothed = aggregate(
    df,
    'date',
    'new_cases_smoothed',
    'sum')

X, new_vaccinations_smoothed = aggregate(
    df,
    'date',
    'new_vaccinations_smoothed',
    'sum')

double_y('New Cases (Smoothed) vs. New Vaccinations (Smoothed)',
         'New Cases (Smoothed) / Millions',
         'New Vaccinations (Smoothed) / Millions',
         'Dates (Daily)',
         'New_Cases_New_Vaccinations.png',
         X,
         new_cases_smoothed,
         new_vaccinations_smoothed)
```

Figure 19. Code to plot, number of vaccinations compared with COVID-19 new cases.

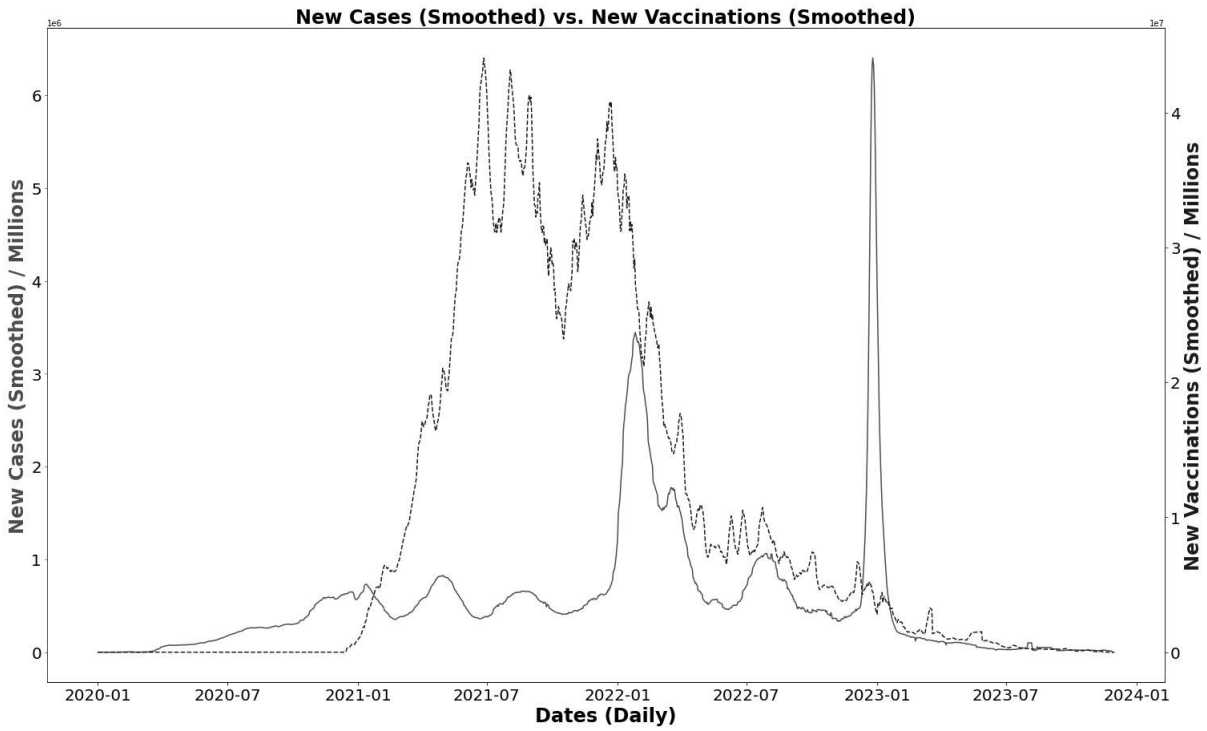

Figure 20. New cases (red) contrasted with new vaccinations (blue).

COVID-19 new cases (red) increased in 2020. After vaccination (blue) was introduced in 2021, the COVID-19 new cases stopped increasing. However, the surge in cases during the first quarter of 2022 was attributed to the highly transmissible Omicron variant [8]. Subsequently, the number of COVID new cases reduced rapidly after 2022. Omicron was milder [8], meaning more survivors. Therefore, more people have hybrid immunity, which is more effective than vaccine-only-immunity [9].

5. Advantage of PySpark (Big data) over Pandas (conventional).

The size of data that Pandas can process is limited to a single machine. On the other hand, in PySpark size of data that PySpark can process is limited only by the number of machines in our cluster [10].

Data scientists can expedite big data processing by conducting work over a cluster of machines. As PySpark is inter-compatible with Pandas, the data scientist can convert the smaller, processed data back into Pandas, for analysis in a local machine [11].

6. Reflection

Both PySpark and Pandas can process big COVID data quickly. Both have high-level tools to make data manipulation easy. Nevertheless, Pandas will run out of memory when the data size

is too large for the local machine. In this case, we should distribute the data processing job across a cluster of machines. PySpark is the most efficient library to do so.

- The codes and data are contained in the Google drive link: https://shorturl.at/aAX35
- 1,076 words until here.

References:

1. Mathieu, E., Ritchie, H., Rodes-Guirao, L., Appel, C., Giattino, C., Hassel, J., Macdonald, B., Dattani, S., Beltekian, D., Orti -Ospina, E. and Roser, M. (2020). Coronavirus Pandemic (COVID-19). Published online at OurWorldInData.org. Retrieved from https://github.com/owid/covid-19-data/tree/master/public/data
2. NVIDIA. (2023). Pandas. https://www.nvidia.com/en-us/glossary/data-science/pandas-python/
3. NumFocus Inc. (2023). Pandas. https://pandas.pydata.org/
4. Noble Desktop. (2023). Pandas Library Overview. https://www.nobledesktop.com/learn/python/pandas-overview
5. Hunter, J., Dale, D., Firing, E., Droettboom, M. and the Matplotlib development team. (2012). https://matplotlib.org/3.5.3/api/_as_gen/matplotlib.pyplot.html
6. Apache Spark. (2023). pyspark.sql.DataFrame.fillna. https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.fillna.html
7. AWS. (2023). What is Apache Spark? https://aws.amazon.com/what-is/apache-spark/
8. Team Verywell Health. (2023). A Timeline of COVID-19 Variants. https://www.verywellhealth.com/covid-variants-timeline-6741198.
9. Gier, B., Huiberts, A., Hoeve, C.E., Hartog, G., Werkhoven, H., Binnendijk, R., Hahne, S.J.M, Melker, H.E., Hof, S., and Knol, M.J. (2023). Effects of COVID-19 vaccinations and previous infection on Omicron SARS-CoV-2 infection and relation with serology. *Nature Communications,* 14, 4793 (2023). https://doi.org/10.1038/s41467-023-40195-z.
10. Vason, I. (2023). From Pandas to Pyspark. https://www.tothenew.com/blog/from-pandas-to-pyspark/
11. IBM. (2016). PySpark: High-performance data processing without learning Scala. https://www.ibm.com/downloads/cas/DVRQZOE.