Repeatable ETL Report

Christian, Addison, Aaron, Lucas

Introduction

Our group was given the topic of healthcare. Our goal was to determine the correlation between the percentage of underweight adults, access to safe drinking water, rates for malaria and tuberculosis, and adult mortality rate in countries worldwide, then use machine learning to predict mortality rates. To do so, we gathered data from the World Health Organization's API, simulated real-time data collection with Kafka, and cleaned & transformed data with Python in Azure Databricks. We also plan to forecast future statistics using current statistics.

To include the Census data, a state (or region) of the US with a given population value will be contrasted with a foreign country approximately sharing that population. This will allow us to compare how different circumstances (water availability, for instance) affect the mortality rate, giving us the ability to infer some potential cause-and-effect relationships between these variables.

Data Sources

StatisticsTimes.com. (2021). *List of Countries by Continent*. StatisticsTimes.com. Retrieved October 29, 2021, from https://statisticstimes.com/geography/countries-by-continents.php.

US Census Bureau. (2021, October 21). *National Population Totals and Components of Change: 2010-2019*. Census.gov. Retrieved October 29, 2021, from https://www.census.gov/data/tables/time-series/demo/popest/2010s-national-total.html.

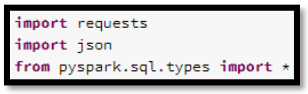
World Health Organization. (2021). *GHO ODATA API*. The Global Health Observatory. Retrieved October 29, 2021, from https://www.who.int/data/gho/info/gho-odata-api.

Extraction

WORLD HEALTH ORGANIZATION

Using the sources listed above, we were able to use API calls in Python to extract data. All work was done within the Azure Data Bricks environment, and each URL was given its own notebook. Starting with the WHO source, data can be extracted with the following steps:

1. Import applicable libraries, such as *requests*, *json*, and *pyspark*



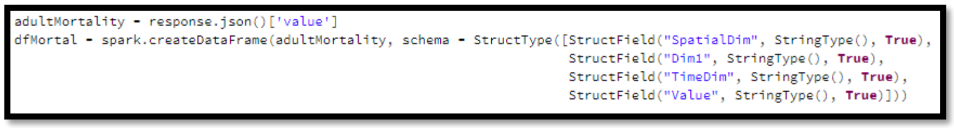
```
import requests
import json
from pyspark.sql.types import *
```

2. Use the URL corresponding to the set of data you'd like to read (this particular example is reading data related to Adult Mortality Rates) in a GET request.

   response = requests.get("https://ghoapi.azureedge.net/api/WHOSIS_000004?$filter=SpatialDimType%20eq%20%27Country%27%20and%20Dim1%20eq%20%27BTSX%27")

   a. This URL can be changed to reflect different data sets.
      i. Adult Mortality: https://ghoapi.azureedge.net/api/WHOSIS_000004?$filter=SpatialDimType%20eq%20%27Country%27%20and%20Dim1%20eq%20%27BTSX%27
      ii. Underweight Adults: https://ghoapi.azureedge.net/api/NCD_BMI_18A?$filter=SpatialDimType%20eq%20%27Country%27%20and%20Dim1%20eq%20%27BTSX%27
      iii. Drinking Water: https://ghoapi.azureedge.net/api/WSH_WATER_BASIC?$filter=SpatialDimType%20eq%20%27Country%27%20and%20Dim1%20eq%20%27TOTL%27
      iv. Malaria: https://ghoapi.azureedge.net/api/SDGMALARIA?$filter=SpatialDimType%20eq%20%27Country%27
      v. Tuberculosis: https://ghoapi.azureedge.net/api/MDG_0000000020?$filter=SpatialDimType%20eq%20%27Country%27%20and%20TimeDim%20ge%202000%20and%20TimeDim%20le%202016

3. Read in the values and typecast them



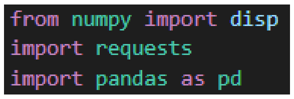
```
adultMortality = response.json()['value']
dfMortal = spark.createDataFrame(adultMortality, schema = StructType([StructField("SpatialDim", StringType(), True),
                                                                      StructField("Dim1", StringType(), True),
                                                                      StructField("TimeDim", StringType(), True),
                                                                      StructField("Value", StringType(), True)]))
```

At this stage, we have extracted the data from the API. Repeat this process for the other WHO data sets.

CENSUS

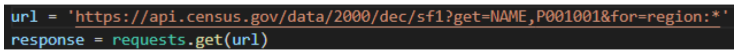
We needed to incorporate information from the Census. This information was only used for comparison, not for any heavy coding or analysis.

1. Imported libraries



```
from numpy import disp
import requests
import pandas as pd
```

2. Used an API call and GET request to load data relating to US regions



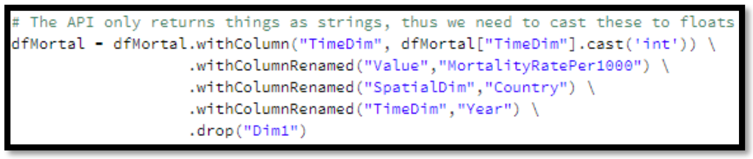
```
url = 'https://api.census.gov/data/2000/dec/sf1?get=NAME,P001001&for=region:*'
response = requests.get(url)
```

We have extracted data from the API.

Transformation

WORLD HEALTH ORGANIZATION

1. Replace column names with more suitable names.

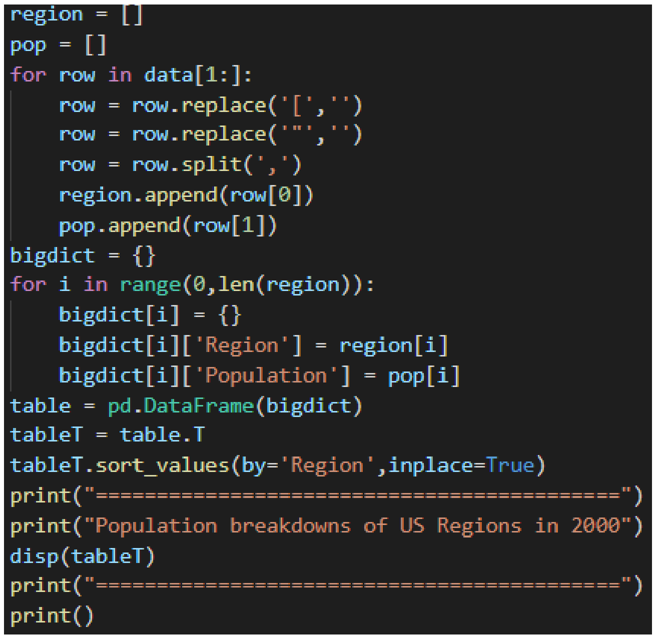


```
# The API only returns things as strings, thus we need to cast these to floats
dfMortal = dfMortal.withColumn("TimeDim", dfMortal["TimeDim"].cast('int')) \
                .withColumnRenamed("Value","MortalityRatePer1000") \
                .withColumnRenamed("SpatialDim","Country") \
                .withColumnRenamed("TimeDim","Year") \
                .drop("Dim1")
```

Repeat this process for the other WHO data sets.

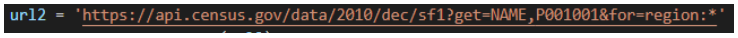
CENSUS

1. Turned the census data into a dictionary, then a pandas dataframe, then transposed, sorted and printed the data.



```
region = []
pop = []
for row in data[1:]:
    row = row.replace('[','')
    row = row.replace('"','')
    row = row.split(',')
    region.append(row[0])
    pop.append(row[1])
bigdict = {}
for i in range(0,len(region)):
    bigdict[i] = {}
    bigdict[i]['Region'] = region[i]
    bigdict[i]['Population'] = pop[i]
table = pd.DataFrame(bigdict)
tableT = table.T
tableT.sort_values(by='Region',inplace=True)
print("==========================================")
print("Population breakdowns of US Regions in 2000")
disp(tableT)
print("==========================================")
print()
```

This process can also be repeated for 2010 data using the following url:



```
url2 = 'https://api.census.gov/data/2010/dec/sf1?get=NAME,P001001&for=region:*'
```

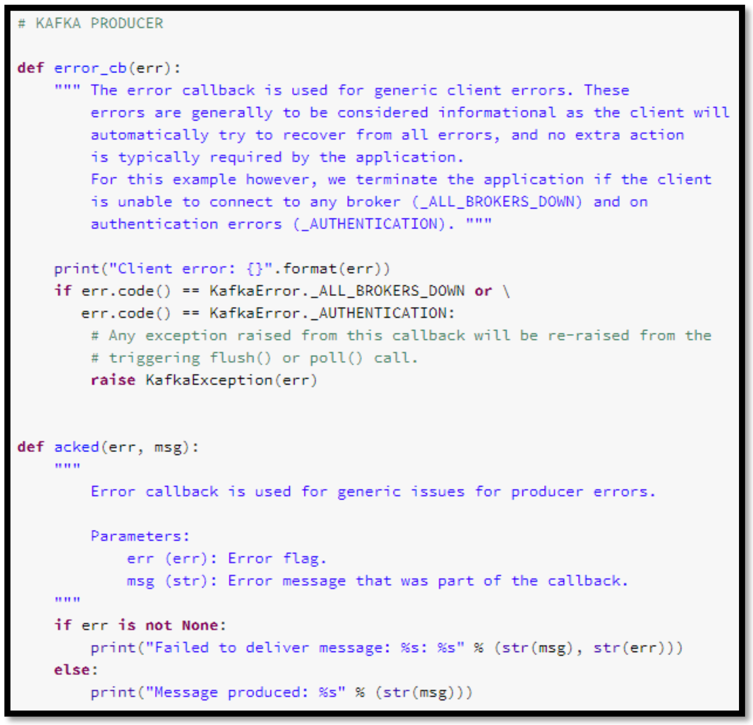
This allows us to see populations of the US regions in 2000 and 2010 that we can compare to other countries.

Load

After each WHO data set was extracted, we added a Kafka producer to each notebook. The code segments below were provided to us by instructors.

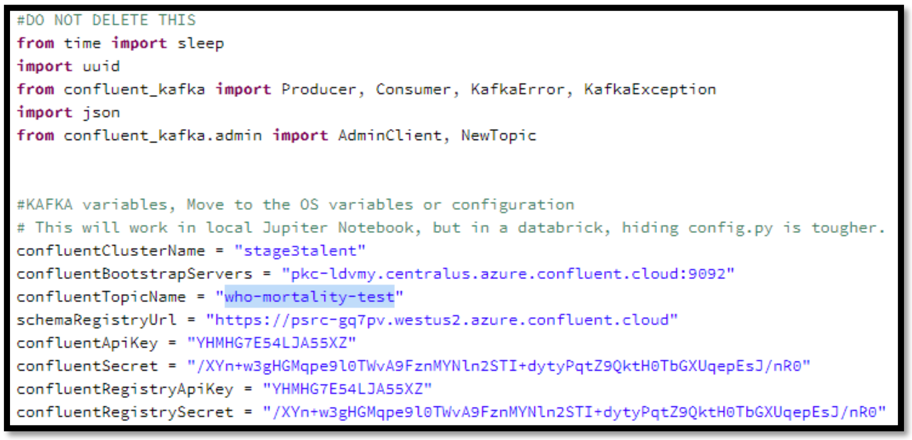
Commented [CB1]: Rewrite for clarity. Something like "After each data set was extracted, we added a Kafka producer to each Data Brick in the same notebook."

1)



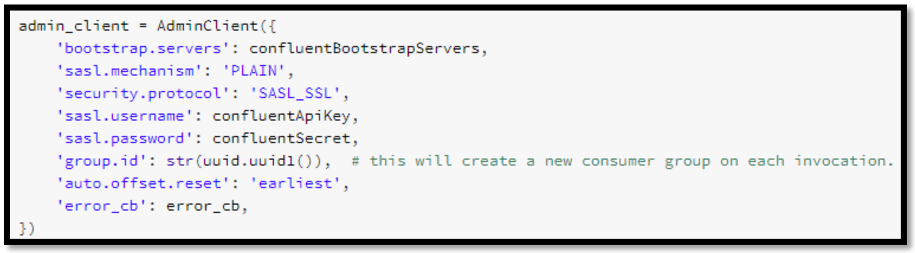
```python
# KAFKA PRODUCER

def error_cb(err):
    """ The error callback is used for generic client errors. These
        errors are generally to be considered informational as the client will
        automatically try to recover from all errors, and no extra action
        is typically required by the application.
        For this example however, we terminate the application if the client
        is unable to connect to any broker (_ALL_BROKERS_DOWN) and on
        authentication errors (_AUTHENTICATION). """

    print("Client error: {}".format(err))
    if err.code() == KafkaError._ALL_BROKERS_DOWN or \
       err.code() == KafkaError._AUTHENTICATION:
        # Any exception raised from this callback will be re-raised from the
        # triggering flush() or poll() call.
        raise KafkaException(err)


def acked(err, msg):
    """
        Error callback is used for generic issues for producer errors.

        Parameters:
            err (err): Error flag.
            msg (str): Error message that was part of the callback.
    """
    if err is not None:
        print("Failed to deliver message: %s: %s" % (str(msg), str(err)))
    else:
        print("Message produced: %s" % (str(msg)))
```

2) The next code segment set up the Kafka connection. The highlighted section below, *confluentTopicName*, must be changed for each producer. Good practice is to give it a sensible name, related to the data.



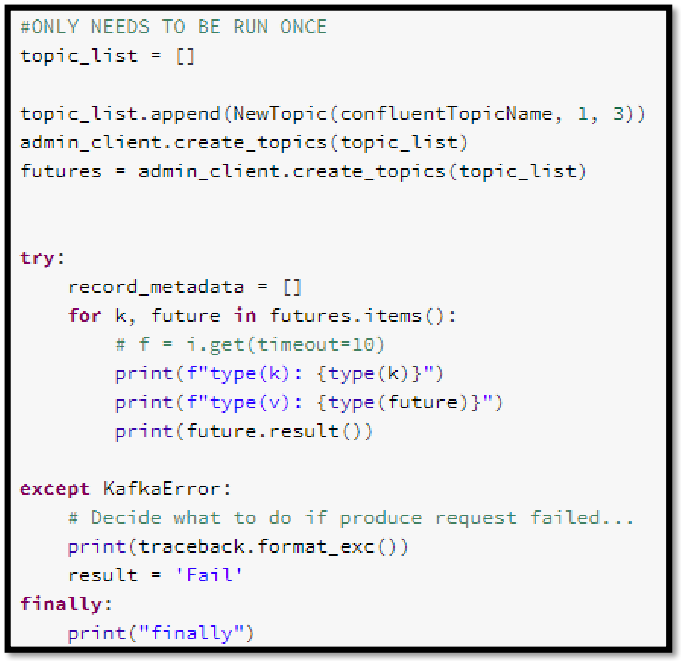
```
#DO NOT DELETE THIS
from time import sleep
import uuid
from confluent_kafka import Producer, Consumer, KafkaError, KafkaException
import json
from confluent_kafka.admin import AdminClient, NewTopic


#KAFKA variables, Move to the OS variables or configuration
# This will work in local Jupiter Notebook, but in a databrick, hiding config.py is tougher.
confluentClusterName = "stage3talent"
confluentBootstrapServers = "pkc-ldvmy.centralus.azure.confluent.cloud:9092"
confluentTopicName = "who-mortality-test"
schemaRegistryUrl = "https://psrc-gq7pv.westus2.azure.confluent.cloud"
confluentApiKey = "YHMHG7E54LJA55XZ"
confluentSecret = "/XYn+w3gHGMqpe9l0TWvA9FznMYNln2STI+dytyPqtZ9QktH0TbGXUqepEsJ/nR0"
confluentRegistryApiKey = "YHMHG7E54LJA55XZ"
confluentRegistrySecret = "/XYn+w3gHGMqpe9l0TWvA9FznMYNln2STI+dytyPqtZ9QktH0TbGXUqepEsJ/nR0"
```

3) The next code segment was also given by instructors. It accounts for security coverage.



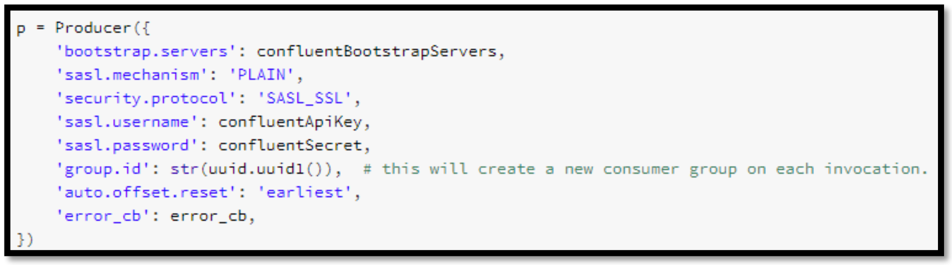
```
admin_client = AdminClient({
    'bootstrap.servers': confluentBootstrapServers,
    'sasl.mechanism': 'PLAIN',
    'security.protocol': 'SASL_SSL',
    'sasl.username': confluentApiKey,
    'sasl.password': confluentSecret,
    'group.id': str(uuid.uuid1()),  # this will create a new consumer group on each invocation.
    'auto.offset.reset': 'earliest',
    'error_cb': error_cb,
})
```

4) The next code segment creates the topic. It only needs to be run once.



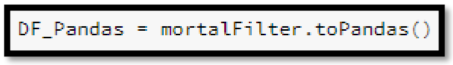
```
#ONLY NEEDS TO BE RUN ONCE
topic_list = []

topic_list.append(NewTopic(confluentTopicName, 1, 3))
admin_client.create_topics(topic_list)
futures = admin_client.create_topics(topic_list)


try:
    record_metadata = []
    for k, future in futures.items():
        # f = i.get(timeout=10)
        print(f"type(k): {type(k)}")
        print(f"type(v): {type(future)}")
        print(future.result())

except KafkaError:
    # Decide what to do if produce request failed...
    print(traceback.format_exc())
    result = 'Fail'
finally:
    print("finally")
```

5) The next segment completes the process of making the producer.



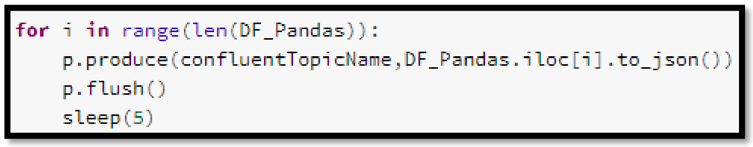
```
p = Producer({
    'bootstrap.servers': confluentBootstrapServers,
    'sasl.mechanism': 'PLAIN',
    'security.protocol': 'SASL_SSL',
    'sasl.username': confluentApiKey,
    'sasl.password': confluentSecret,
    'group.id': str(uuid.uuid1()),  # this will create a new consumer group on each invocation.
    'auto.offset.reset': 'earliest',
    'error_cb': error_cb,
})
```

6) We turned our data into a pandas dataframe:



```
DF_Pandas = mortalFilter.toPandas()
```

7) Lastly, to start the producer running, we run the following code:

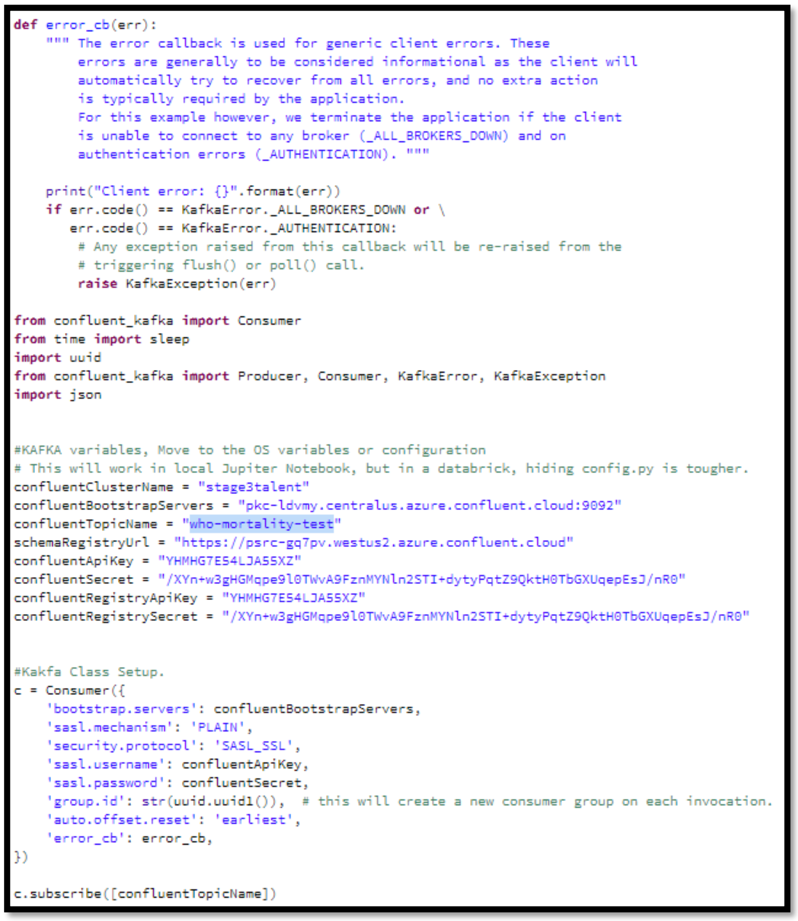


```
for i in range(len(DF_Pandas)):
    p.produce(confluentTopicName,DF_Pandas.iloc[i].to_json())
    p.flush()
    sleep(5)
```

At this point, when these steps have been applied to Adult Mortality, Water Accessibility, Underweight Adults, Tuberculosis, and Malaria, we have set up all necessary Producers.

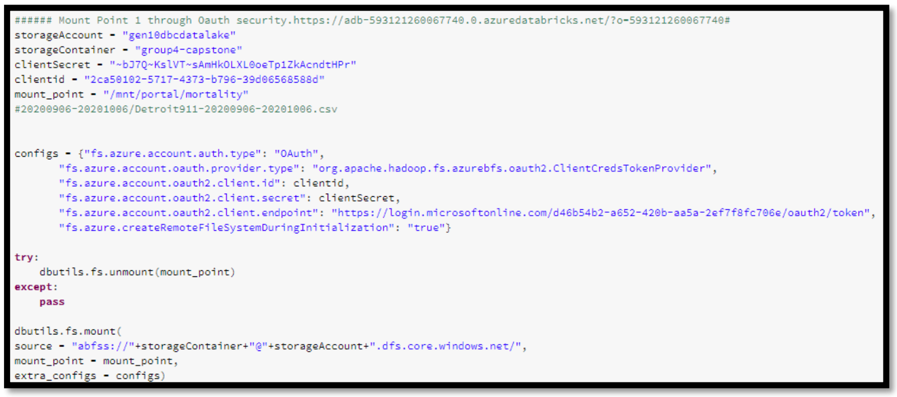
Next, we need to set up our Consumers. To set up a consumer for each of your WHO datasets, follow the steps below, using individual notebooks for each dataset.

1) Provided by instructors, this code segment connects to the producer. Note the highlighted section, which is the name of the Kafka topic used. This will need to be altered to reflect which producer you are connecting to.



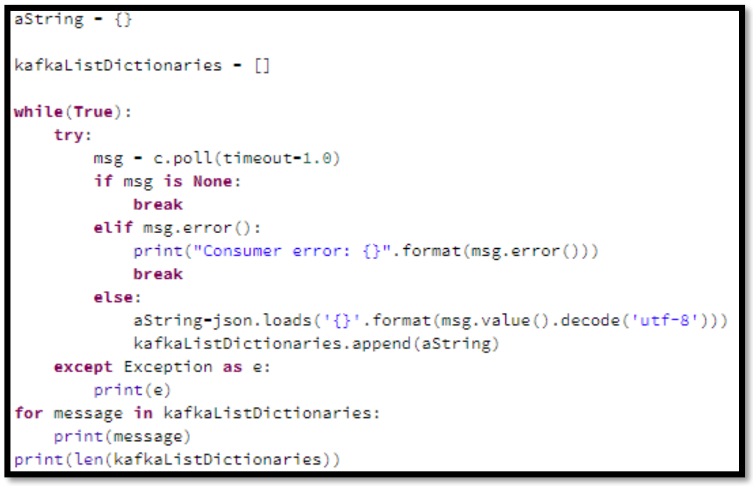
```
def error_cb(err):
    """ The error callback is used for generic client errors. These
        errors are generally to be considered informational as the client will
        automatically try to recover from all errors, and no extra action
        is typically required by the application.
        For this example however, we terminate the application if the client
        is unable to connect to any broker (_ALL_BROKERS_DOWN) and on
        authentication errors (_AUTHENTICATION). """

    print("Client error: {}".format(err))
    if err.code() == KafkaError._ALL_BROKERS_DOWN or \
       err.code() == KafkaError._AUTHENTICATION:
        # Any exception raised from this callback will be re-raised from the
        # triggering flush() or poll() call.
        raise KafkaException(err)

from confluent_kafka import Consumer
from time import sleep
import uuid
from confluent_kafka import Producer, Consumer, KafkaError, KafkaException
import json


#KAFKA variables, Move to the OS variables or configuration
# This will work in local Jupiter Notebook, but in a databrick, hiding config.py is tougher.
confluentClusterName = "stage3talent"
confluentBootstrapServers = "pkc-ldvmy.centralus.azure.confluent.cloud:9092"
confluentTopicName = "who-mortality-test"
schemaRegistryUrl = "https://psrc-gq7pv.westus2.azure.confluent.cloud"
confluentApiKey = "YHMHG7E54LJA55XZ"
confluentSecret = "/XYn+w3gHGMqpe9l0TWvA9FznMYNln2STI+dytyPqtZ9QktH0TbGXUqepEsJ/nR0"
confluentRegistryApiKey = "YHMHG7E54LJA55XZ"
confluentRegistrySecret = "/XYn+w3gHGMqpe9l0TWvA9FznMYNln2STI+dytyPqtZ9QktH0TbGXUqepEsJ/nR0"


#Kakfa Class Setup.
c = Consumer({
    'bootstrap.servers': confluentBootstrapServers,
    'sasl.mechanism': 'PLAIN',
    'security.protocol': 'SASL_SSL',
    'sasl.username': confluentApiKey,
    'sasl.password': confluentSecret,
    'group.id': str(uuid.uuid1()),  # this will create a new consumer group on each invocation.
    'auto.offset.reset': 'earliest',
    'error_cb': error_cb,
})

c.subscribe([confluentTopicName])
```

2) Connecting to the mount point uses another portion of instructor-provided code, seen below.



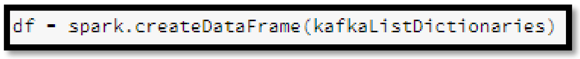
```python
###### Mount Point 1 through Oauth security.https://adb-593121260067740.0.azuredatabricks.net/?o=593121260067740#
storageAccount = "gen10dbcdatalake"
storageContainer = "group4-capstone"
clientSecret = "~bJ7Q~KslVT~sAmHkOLXL0oeTp1ZkAcndtHPr"
clientid = "2ca50102-5717-4373-b796-39d06568588d"
mount_point = "/mnt/portal/mortality"
#20200906-20201006/Detroit911-20200906-20201006.csv


configs = {"fs.azure.account.auth.type": "OAuth",
       "fs.azure.account.oauth.provider.type": "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider",
       "fs.azure.account.oauth2.client.id": clientid,
       "fs.azure.account.oauth2.client.secret": clientSecret,
       "fs.azure.account.oauth2.client.endpoint": "https://login.microsoftonline.com/d46b54b2-a652-420b-aa5a-2ef7f8fc706e/oauth2/token",
       "fs.azure.createRemoteFileSystemDuringInitialization": "true"}

try:
    dbutils.fs.unmount(mount_point)
except:
    pass

dbutils.fs.mount(
source = "abfss://"+storageContainer+"@"+storageAccount+".dfs.core.windows.net/",
mount_point = mount_point,
extra_configs = configs)
```

3) Build a list of dictionaries from your data.



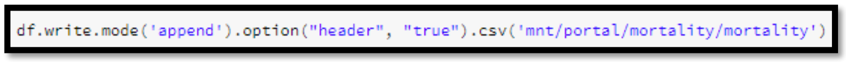
```python
aString = {}

kafkaListDictionaries = []

while(True):
    try:
        msg = c.poll(timeout=1.0)
        if msg is None:
            break
        elif msg.error():
            print("Consumer error: {}".format(msg.error()))
            break
        else:
            aString=json.loads('{}'.format(msg.value().decode('utf-8')))
            kafkaListDictionaries.append(aString)
    except Exception as e:
        print(e)
for message in kafkaListDictionaries:
    print(message)
print(len(kafkaListDictionaries))
```

4) Create a dataframe out of that list of dictionaries.



```python
df = spark.createDataFrame(kafkaListDictionaries)
```

5) Write this data to a csv and save it to the data lake.

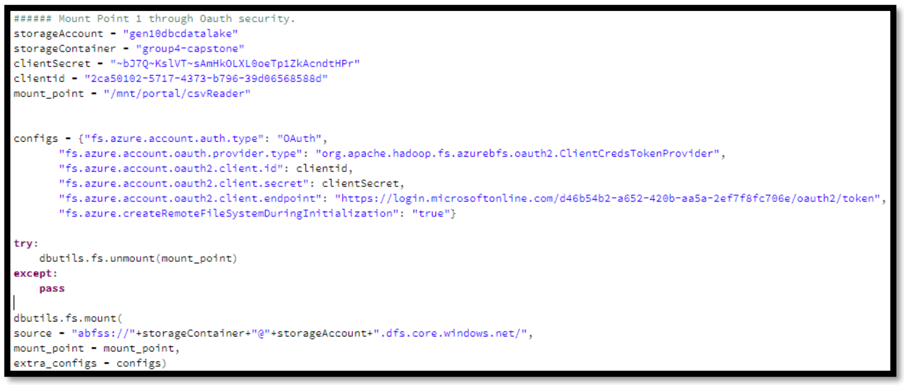


```python
df.write.mode('append').option("header", "true").csv('mnt/portal/mortality/mortality')
```

Data has now been produced, consumed, and most recently saved to the datalake in .csv format. Next, we need to load that data into a database.

We created another notebook in the databricks environment specifically to read these .csv files and push the data into a SQL database, as seen below.

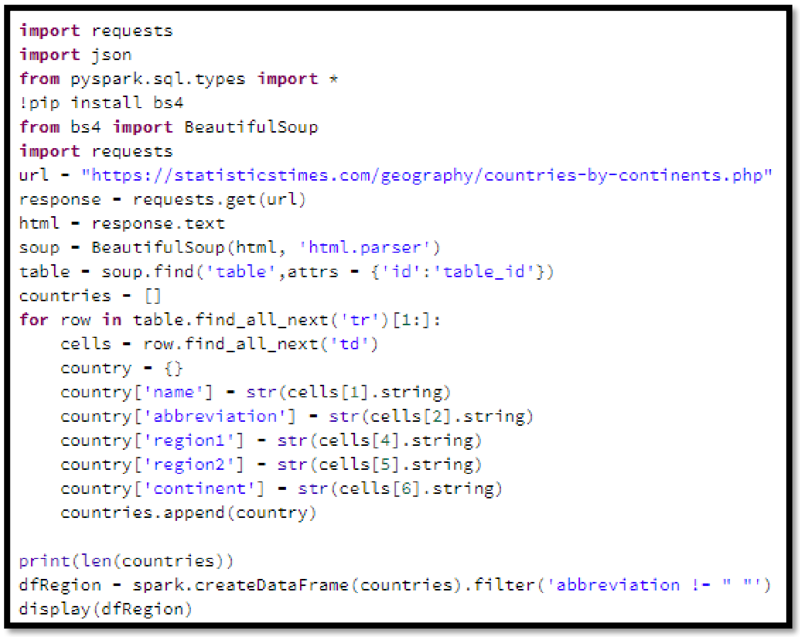
1) To start off with, attach to the mount point. This code was provided to us by instructors.



```
###### Mount Point 1 through Oauth security.
storageAccount = "gen10dbcdatalake"
storageContainer = "group4-capstone"
clientSecret = "~bJ7Q~KslVT~sAmHkOLXL0oeTp1ZkAcndtHPr"
clientid = "2ca50102-5717-4373-b796-39d06568588d"
mount_point = "/mnt/portal/csvReader"


configs = {"fs.azure.account.auth.type": "OAuth",
       "fs.azure.account.oauth.provider.type": "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider",
       "fs.azure.account.oauth2.client.id": clientid,
       "fs.azure.account.oauth2.client.secret": clientSecret,
       "fs.azure.account.oauth2.client.endpoint": "https://login.microsoftonline.com/d46b54b2-a652-420b-aa5a-2ef7f8fc706e/oauth2/token",
       "fs.azure.createRemoteFileSystemDuringInitialization": "true"}

try:
    dbutils.fs.unmount(mount_point)
except:
    pass

dbutils.fs.mount(
source = "abfss://"+storageContainer+"@"+storageAccount+".dfs.core.windows.net/",
mount_point = mount_point,
extra_configs = configs)
```

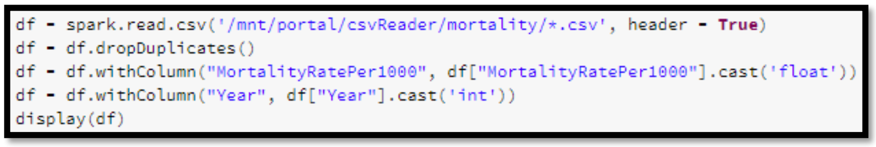
2) Create table that includes geographic regions

Commented [CB2]: Pls clarify - which previous steps?



```
import requests
import json
from pyspark.sql.types import *
!pip install bs4
from bs4 import BeautifulSoup
import requests
url = "https://statisticstimes.com/geography/countries-by-continents.php"
response = requests.get(url)
html = response.text
soup = BeautifulSoup(html, 'html.parser')
table = soup.find('table',attrs = {'id':'table_id'})
countries = []
for row in table.find_all_next('tr')[1:]:
    cells = row.find_all_next('td')
    country = {}
    country['name'] = str(cells[1].string)
    country['abbreviation'] = str(cells[2].string)
    country['region1'] = str(cells[4].string)
    country['region2'] = str(cells[5].string)
    country['continent'] = str(cells[6].string)
    countries.append(country)

print(len(countries))
dfRegion = spark.createDataFrame(countries).filter('abbreviation != " "')
display(dfRegion)
```

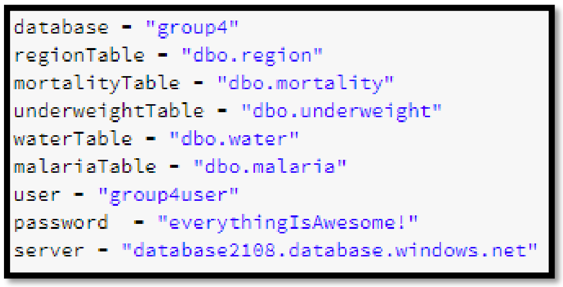
3) Read in the csv files, drop duplicates, recast datatypes appropriately for Year and Mortality Rate



```
df = spark.read.csv('/mnt/portal/csvReader/mortality/*.csv', header = True)
df = df.dropDuplicates()
df = df.withColumn("MortalityRatePer1000", df["MortalityRatePer1000"].cast('float'))
df = df.withColumn("Year", df["Year"].cast('int'))
display(df)
```

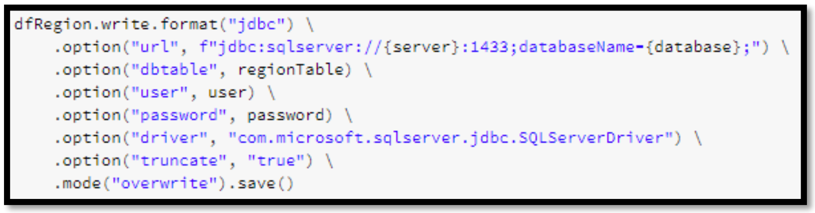
4) Repeat this previous step for all datasets (Mortality, Water, Underweight Adults, Malaria, Tuberculosis)

5) Push data to database by connecting to it first.



```
database = "group4"
regionTable = "dbo.region"
mortalityTable = "dbo.mortality"
underweightTable = "dbo.underweight"
waterTable = "dbo.water"
malariaTable = "dbo.malaria"
user = "group4user"
password  = "everythingIsAwesome!"
server = "database2108.database.windows.net"
```

6) Then write the data to the database using the following code section.



```
dfRegion.write.format("jdbc") \
    .option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
    .option("dbtable", regionTable) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .option("truncate", "true") \
    .mode("overwrite").save()
```

Repeating this previous step for all dataframes will complete the process of loading the SQL database with the data. We can now use Power BI to create visualizations and a dashboard for this project.

Conclusion

We have extracted data, transformed data, and loaded data – completing the ETL process. After finding the data, accessing them via API, transforming them, and finally loading them into a database, we can now address our original questions and create meaningful visualizations.