ETL Report

Daniel Rose, Finn McSweeney, Rachel Walter, Isabelle Hyppolite

1/30/23

Table of contents

CSV ETL

Our team used multiple datasets in order to answer our initial questions and this included a global emissions dataset, world population, GDP, industry and air quality data. In order to get an overview of emissions relating to GDP and population the datasets were merged using inner joins and filtered to only include data from 2002-2021. These datasets were merged on the country code columns and further cleaning was not necessary. We also used data from the CDC relating to different industries and this was from 2011-2022. In order to use this data effectively, data was collected and cleaned from the environmental protection agency on particulate matter in the air and nitrogen dioxide levels. Each of these datasets had daily information for numerous cities across the country and the mean was taken for each city over the year. Due to each dataset being only one year, the same cleaning steps had to be applied to each file and eventually all datasets were appended from 2017-2021. Once both of these datasets had been cleaned and appended, we were able to merge them with the industry data collected from the CDC and were able to filter on different pollutants or industries.

Kafka Producer

Weather API

We used Open Weather's Air Pollution API for multiple portions of this project. For the Kafka portion, we used current data because we wanted to get live streaming data onto our dashboard. To access the API, we requested a key and used Virtual Studio Code to make our requests. For current data, all you need to specify is your key, and the latitude and longitude of the city you want air quality data on. Before creating our producer, we tested how we were going to get the data from the requests into a dataframe. First, we put the response into JSON with ".json()". Next, we took the "list" portion of the response and used ".json_normalize()" to get the majority of the data from the request into a dataframe. We then added a column for the city's latitude, longitude, and name, and for the date. For the date, the response comes in as Unix, so we converted the date's datatype from Unix to datetime with ".to_datetime()". We then used ".rename()" to change the column names, since ".json_normalize()" automatically assigned the headers for the columns as two dictionary key names combined together since the dictionaries were nested.

We also used the historic data from the API for the machine learning portion of our project. For historic data, you again need your key and the city's latitude and longitude, but this time you also need to specify the start and end date of the data you want to pull. This has to be in Unix, so we used a Unix converter to get our date into Unix to make the request. We wanted as much data as possible for our machine learning portion, but the data only went back to November 27th 2020 so we requested data from December 1st 2020 to December 31st 2022. We also wanted to use our data to predict AQI or PM2.5 values, and different cities may have different patterns so we focused on data specifically from New York City. We again put the response in JSON format with ".json". For the historic data, ".json_normalize" would not work since the request is made up of many dictionaries since the data from each hour is in its own dictionary. Instead, we imported "jmespath" to parse through the JSON. We used "jmespath.compile()" on three portions of the response to get three separate datasets. The datafames with AQI and the rest of the components both looked good, but for the date dataframe the column was named "0" so we again used ".rename()" to change the column's name to "date." We also used ".to_datetime()" again to convert the date from Unix to datetime. Lastly, we combined the three dataframes together with ".concat()", we also used ".reset_index()" to drop the index so we would not have multiple indexes in the final dataframe.

Admin Client

To produce data we first need an admin client to create a new topic for us to use. Our admin client takes in a server url, API key, and secret key that was given to us by our instructor in order to connect to the server. We also set the sasl mechanism to "PLAIN", security protocol settings to "SASL_SSL", and add a unique group id. Lastly we use the create_topics method to create our new topic called "group-3". We will use this group-3 when we both produce and consume data.

Producing

The first step in producing data to kafka is to create the data we want to produce. This is done above using our weather API data. We then convert this data into json format so that It can be sent in kafka. Next we want to create the actual producer itself. Like the admin client this requires our server url, API key, and secret key. We also have to set the security protocol and mechanism which are also the same as in our admin client. Lastly we configure our group id to its own unique id. Then to send the data we first use a try catch statement in case any errors occur. Inside of that is a json.dumps command to dump our json to a new variable. Next, we use the .produce method with our “group-3” topic and our data variable to send the data and a .flush method to make sure the message gets sent. If there are no errors our programs prints that it was successfully sent otherwise it will print out its error. We can now go and read the data in our Kafka consumer.

Kafka Consumer

Consuming

To be able to consume Kafka data the first thing we needed to do was to set all of the server variables. This includes the Kafka username, password, security protocol, mechanisms, and url which validates that we have access to the kafka server. We will then also declare our session timeout in case an error occurs while connecting to the server and our group id that should be different from our admin client or producer's group id. Lastly, we will declare a kafka consumer with our variables and subscribe to our “group-3” topic.

To read in the data on our server we first start with a try catch statement to cover any errors that might occur. We then use the consumer.poll command to read in the data from kafka but before we process the data we have to make sure it is not none. If the data is not None, we can continue and decode the value of the data. We then set this data to a json with all of our table rows. The last step is to convert this json to a pandas dataframe.

Data Lake

In order to connect to our Data Lake we first have to set its variables. The most important variable is the SAS Token which was manually created with specific permissions that allows us to both read and write from our data lake. We then specify our container name, storage account name and the root path that our files will be stored.

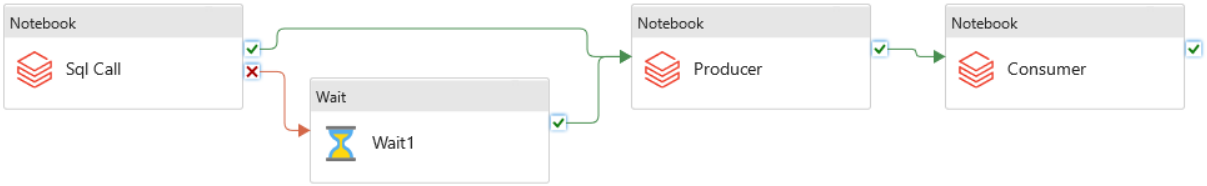
Next, we convert our python dataframe from our previous section into a spark dataframe. With both the spark dataframe and our specified variables we can set our spark configuration with the spark.conf.set command. Lastly, using the spark.write command, we save our spark dataframe into our specified path in our data lake.

Azure SQL

To connect to our SQL server, we first start by needing to set the configuration variables. This starts with declaring our server url, database name and table name. Next we declare our username and password that was manually created directly in our SQL server. Similarly to our Data Lake we use spark's spark.write command but with different options. The first option to set is the spark format which is set to jbdc which is an api that allows us to connect to external databases. The second option we set is the mode which is set to overwrite so we only have the current data we want to look at in our server. We then also have to set our SQL driver to a driver made by Microsoft for connecting to SQL servers. The rest of our options are our configuration variables. The command ends with a .save() which will finally save the data in the server.

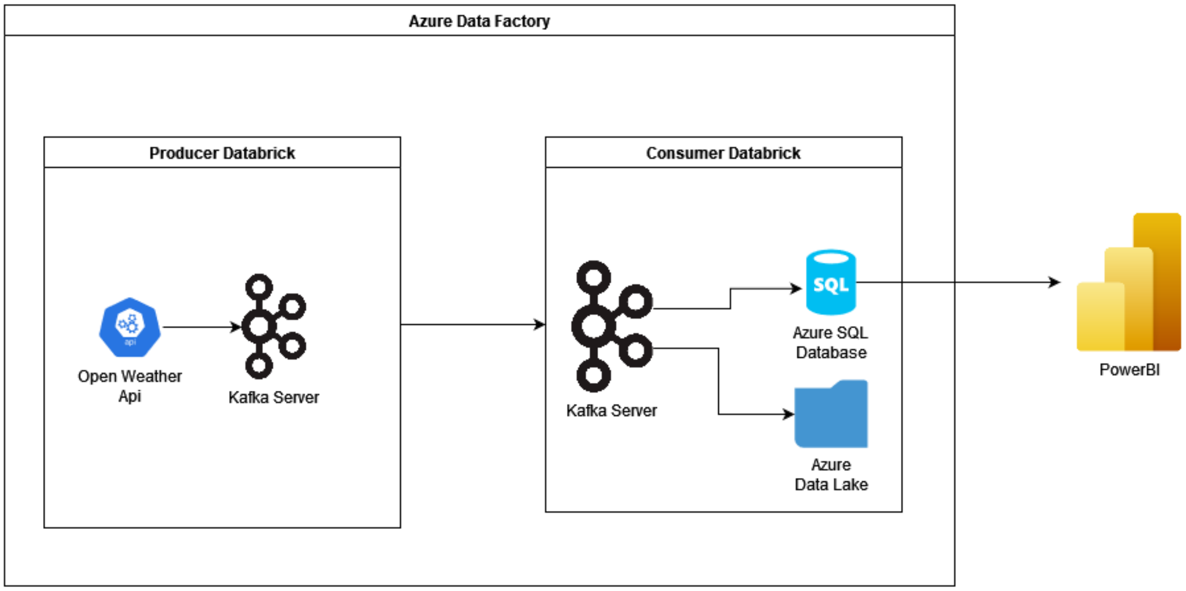
Data Factory

We use a data factory because it allows us to automatically run our Kafka producer and consumer files. To use our data factory we created a new pipeline that we called "EmissionsPipeline". This pipeline starts by calling our SQl server. We start with this because there is a chance our SQl server has been shut down due to inactivity and that it needs to start up. If the server call failed because the server was shutdown the data factory waites two minutes for the server to turn on before running our producer otherwise it will directly run our producer. After the producer runs then our consumer runs to update our data lake and SQL database with our new weather data. Below is a picture of our data factory pipeline.



Power BI

To read our SQL database in PowerBI and have our data automatically update we first create a datamart which is a newer feature in Power BI. Inside of our datamart we can directly connect to our SQL server using our server url, database name, table name, username, and password. Next we can point our Power BI dashboard to use the data stored in our datamart. Then to make the data automatically update we create a datamart schedule to refresh its sql data twice a day slightly after the data gets updated by our data factory. This allows us to have a stream of data all the way from our Weather API through kafka into an sql database and finally into our Power BI report. A picture of our full network diagram is shown below.


Azure Data Factory
Producer Databrick
Open Weather Api
Kafka Server
Consumer Databrick
Kafka Server
Azure SQL Database
Azure Data Lake
PowerBI