You need to create a service account on google cloud with Big Query service enabled.

Next, we connect the client to the database. To do this, you will need to download a JSON file that contains the BigQuery service account credentials. If you don’t have a service account, follow this guide to create one, and then proceed to download the JSON file to your local machine.

Now that we have the BigQuery client set up and ready to use, we can execute queries on the BigQuery dataset.

For this, we use the query method, which inserts a query job into the BigQuery queue. These queries are then executed asynchronously – in the sense that we do not specify any timeout, and the client waits for the job to complete. As soon as the job is complete, the method returns a Query\_Job instance containing the results.

In this post, you’ll learn how to connect to a Google BigQuery warehouse with Python. There’s a few different ways to do this, but we’ll use the official Google Cloud Python Client (google-cloud-bigquery). We’ll walk through:

1. Installing the Google Cloud Bigquery Python Client (google-cloud-bigquery) 2. Authenticating and connecting to your BigQuery data warehouse 3. Running a query! At this point, you’ve successfully connected to and queried BigQuery from Python, and you can write any query you want. Job done? Not quite. We’ll also cover: 4. Reading data from a BigQuery query into a pandas DataFrame Bonus: Writing a DataFrame back into a BigQuery table.

If you’re running through this live, it should only take you around 10 minutes to go from zero to successful query. Let's get into it!

🐍 Installing the Google Cloud Python Client

There’s nothing particularly special about the environment needed for this, so if you already know you have a working Python 3.6+ installation, you can skip to the next section, Installing the google-cloud-python package.

Prepping your Python environment

There are two likely scenarios here for how you're accessing Python:

a. You’re using a Jupyter notebook or a Jupyter notebook alternative:

You should be pretty much good to go in this case. If you want to make sure, you can run this Python command in a cell and look for a response that's >= 3.6:

from platform import python\_version

print(python\_version())

Checking the Python version of a notebook

Checking the Python version of a notebook

b. You’re using the terminal / command line / some other Python IDE directly:

Open up your terminal (Terminal app on Mac, command prompt or Powershell on windows, etc.)

Run python --version to check what Python version is installed. Mine prints out Python 3.9.12, but as long as yours is 3.6 or greater, you’re perfect.

If you get a command not found: python error, or your output is Python 2.x, try running python3 --version. If that works, then you have separate installations of Python 3 and Python 2, and for the rest of this tutorial you’ll need to substitute python3 for python and pip3 for pip in the example commands.

If python3 --version also does not work, then you don’t have Python installed. The easiest way to get this working is to download the official installer for your machine.

PS: We won’t go deep into the setup of virtual environments here, but if you’re doing a lot of Python work directly at the command line, you’ll want to read up on them. Virtualenv works well, and lots of people also like conda.

Installing the google-cloud-bigquery package

This is probably the easiest step of the whole tutorial! You just have to run one command to install the official BigQuery Python Connector:

pip install google-cloud-bigquery

If you’re using a Jupyter notebook, add a ! before pip to let this command run.

!pip install google-cloud-bigquery

And if you had to run python3 -- version earlier to get a working output, you’ll run pip3 instead of pip to install the package:

pip3 install google-cloud-bigquery

This pip install command will spin through a bunch of messages and progress bars, and maybe a few warnings about your “pip version” or various package deprecations. This is OK, and unless anything actually says “ERROR” in red, you can probably ignore it.

Now we’re ready to connect to BigQuery.

Authenticating and connecting to BigQuery

In this next part, we’ll be working with sensitive information: your BigQuery service account credentials. You shouldn’t ever store these directly in code or check them into git. You never know what you might unthinkingly do with that code— send it to a coworker to help, post it on Stack Overflow with a question, check it into a public git repo...

Storing BigQuery credentials

BigQuery service account credentials are provided as a JSON file, so we don’t need to stress about using environment variables or anything. Just make sure you keep this file somewhere safe, and don’t check it into git!

If you don’t already have a service account with BigQuery scopes, this is a nice resource that walks you through the steps to generate one.

Opening a connection to BigQuery

Now let’s start working in Python. Open a new Python session (either in the terminal by running python or python3, or by opening your choice of Jupyter notebook or Jupyter notebook alternative.

We start by importing the BigQuery connector library that we just installed:

from google.cloud import bigquery

from google.oauth2 import service\_account

Next, we’ll open a connection to BigQuery. With this package, this means creating a client object that holds an authenticated session you can use to query data.

credentials = service\_account.Credentials.from\_service\_account\_file('path/to/your/service-account-file.json')

project\_id = 'your-bigquery-project-id-here'

client = bigquery.Client(credentials= credentials,project=project\_id)

If this all runs successfully, then congratulations— you’ve connected to BigQuery and are ready to run queries!

Running a query

BigQuery has some great public datasets that we can use to test our connection, so you can copy/paste this, but it’s pretty simple. You create a client.query() object with the SQL text you want to execute, and then fetch the result() from that query to run it.

query = client.query("""

SELECT \* FROM `bigquery-public-data.fda\_food.food\_events` LIMIT 1000

""")

results = query.result()

for row in results:

print(row)

You can replace that fda\_food query with any SQL query you want to run.

But this isn’t the best way to work with lots of data— you don’t just want to look at the results, you want to do things with them! You could replace the print(row) command in that for loop to do something else... but there's easier ways to work with data in Python!

Reading data into a pandas DataFrame

Since most data analytics and data science projects use pandas to crunch data, what we really want is to get results from a BigQuery query into a pandas DataFrame. Luckily, the library we’re using makes that really easy to do.

All you need to do to fetch the query results as a DataFrame is add to\_dataframe() instead of results(). You may need to also install the db-dtypes python package by running pip install db-dtypes.

query = client.query("""

SELECT \* FROM `bigquery-public-data.fda\_food.food\_events` LIMIT 1000

""")

result\_df = query.to\_dataframe()

This will assign the results of your query to a pandas DataFrame called result\_df. You can print a sample of the results to make sure it worked:

result\_df.head()

Reading data from BigQuery into a pandas DataFrame

Reading data from BigQuery into a pandas DataFrame

And you’re off! You can use this method to execute any BigQuery query and read the results directly into a pandas DataFrame. Now you can use any pandas functions or libraries from the greater Python ecosystem on your data, jumping into a complex statistical analysis, machine learning, geospatial analysis, or even modifying and writing data back to your data warehouse.

Writing data back to BigQuery

Sometimes, an analysis isn’t one way, and you want to put the results of your work back into BigQuery. If you’ve already configured the google-cloud-bigquery package as described above, it’s not too much extra work to pipe data back into the warehouse.

To do this, you’ll need to install one more package: pyarrow. The BigQuery connector uses this to compress a DataFrame prior to writeback.

pip install pyarrow

Let’s say we start with the results of that previous query, and just want to write them to another table in our own project.

from google.cloud import bigquery

from google.oauth2 import service\_account

credentials = service\_account.Credentials.from\_service\_account\_file('path/to/your/service-account-file.json')

project\_id = 'your-bigquery-project-id-here'client = bigquery.Client(credentials= credentials,project=project\_id)

query = client.query("""

SELECT \* FROM `bigquery-public-data.fda\_food.food\_events` LIMIT 1000

""")

result\_df = query.to\_dataframe()

## you could do any manipulation to the dataframe you want here

table\_id = 'your\_dataset\_name.new\_table\_name'

## optional: you can pass in a job\_config with a schema if you want to define

## a specific schema

# job\_config = bigquery.LoadJobConfig(schema=[

# bigquery.SchemaField("field\_name", "STRING"),

# ])

load\_job = client.load\_table\_from\_dataframe(

df, table\_id,

# job\_config=job\_config

) # this will execute the load job

result = load\_job.result()

result.done() # prints True if done

A successful DataFrame writeback to BigQuery

A successful DataFrame writeback to BigQuery

It’s easy to verify the success of this writeback job by running a quick query against the newly created table. Here’s everything all together:

Putting it all together

Putting it all together

This simple template can be repurposed to read or write any data on a Google BigQuery connection. For more (albeit a bit dense) information, you can check out the complete Google Documentation on using the BigQuery API.

Use BigQuery to write directly to a user-generated CSV without cloud storage

If you already have a target CSV that you’d like your transformed data to live in, the steps above can feel a bit out of the way. Thankfully, you can have BigQuery write your data directly to your CSV file, without using cloud storage.

To do so, you’ll need the following:

pyarrow: The python sdk for arrow library so we can process data faster

pandas: A popular library you’ve probably already used for data manipulation and analysis

Google Cloud credentials: The same credentials we used for our service account above.

For this method, we’ll first want to create our CSV file (skips this if you already have a CSV you’d like to write your data to). For our example, I’ve named this “extracted\_data.csv.”

Then, create a function to extract data from BigQuery to CSV with the following code:

[

from google.cloud import bigquery

import os

SERVICE\_ACCOUNT\_JSON = os.environ[‘GOOGLE\_APPLICATION\_CREDENTIALS’]

client = bigquery.Client.from\_sevice\_account.json(SERVICE\_ACCOUNT\_JSON)

def big\_query\_to\_csv():

query = “””

SELECT \* FROM bigquery-public-data.hacker\_news.stories LIMIT 100;

“””

df = client.query(query).to\_dataframe()

df.to\_csv(‘extracted\_data.csv, index=False,header=True)

print(‘csv file generated’)

big\_query\_to\_csv()

]

This method is much faster than our first workflow, but here’s a summary if you missed the steps (I know, they went by pretty quick):

Create or identify the target CSV we want to write to

Create a function to extract data from BigQuery to CSV

Alternative #2: Use a reverse ETL tool to quickly move your data from BigQuery to your CSV (and beyond)

Both of the methods above work well if you just need to quickly pull a couple of datasets for analysis or reference. However, if you have a lot of data, either workflow may take hours, depending on how much data you need to extract (and how often you need to regularly export it).

If your company has thousands of lines of data and hundreds of datasets, this time will add up quickly.

Thankfully, there’s a better (and faster and easier) way: Reverse ETL. Reverse ETL acts as the bridge between your warehouse (BigQuery) and operational destinations, including (but not limited to) CSVs. Reverse ETL tools like Census take out the API wrestling work and let you easily move data from A to B with the click of a sync button.

Speaking specifically to Census, as the pioneer of reverse ETL, we have a ton of experience saving anyone working with data sanity and time. Our tool works with more than 40 integrations (and counting), and our team includes some of the best data experts in the industry.

If you’re curious and want to try out reverse ETL, you can grab a demo here. Or, if you’d rather stick with the manual method for a bit longer, you can check out our library of resources here for more tutorials like this one.

OpenAI

Using the codes above, we built a spark session and set a name for the application. Then, the data was cached in off-heap memory to avoid storing it directly on disk, and the amount of memory was manually specified.

Step 2: Creating the DataFrame

We can now read the dataset we just downloaded:

df = spark.read.csv('datacamp\_ecommerce.csv',header=True,escape="\"")

OpenAI

Note that we defined an escape character to avoid commas in the .csv file when parsing.

Let’s take a look at the head of the dataframe using the show() function:

df.show(5,0)

OpenAI

show() function

The dataframe consists of 8 variables:

InvoiceNo: The unique identifier of each customer invoice.

StockCode: The unique identifier of each item in stock.

Description: The item purchased by the customer.

Quantity: The number of each item purchased by a customer in a single invoice.

InvoiceDate: The purchase date.

UnitPrice: Price of one unit of each item.

CustomerID: Unique identifier assigned to each user.

Country: The country from where the purchase was made

Step 3: Exploratory Data Analysis

Now that we have seen the variables present in this dataset, let’s perform some exploratory data analysis to further understand these data points:

Let’s start by counting the number of rows in the dataframe:

df.count() # Answer: 2,500

OpenAI

How many unique customers are present in the dataframe?

df.select('CustomerID').distinct().count() # Answer: 95

OpenAI

What country do most purchases come from?

To find the country from which most purchases are made, we need to use the groupBy() clause in PySpark:

from pyspark.sql.functions import \*

from pyspark.sql.types import \*

df.groupBy('Country').agg(countDistinct('CustomerID').alias('country\_count')).show()

OpenAI

The following table will be rendered after running the codes above:

groupBy()

Almost all the purchases on the platform were made from the United Kingdom, and only a handful were made from countries like Germany, Australia, and France.

Notice that the data in the table above isn’t presented in the order of purchases. To sort this table, we can include the orderBy() clause:

df.groupBy('Country').agg(countDistinct('CustomerID').alias('country\_count')).orderBy(desc('country\_count')).show()

OpenAI

The output displayed is now sorted in descending order:

descending order

When was the most recent purchase made by a customer on the e-commerce platform?

To find when the latest purchase was made on the platform, we need to convert the “InvoiceDate” column into a timestamp format and use the max() function in Pyspark:

spark.sql("set spark.sql.legacy.timeParserPolicy=LEGACY")

df = df.withColumn('date',to\_timestamp("InvoiceDate", 'yy/MM/dd HH:mm'))

df.select(max("date")).show()

OpenAI

You should see the following table appear after running the code above:

max()

When was the earliest purchase made by a customer on the e-commerce platform?

Similar to what we did above, the min() function can be used to find the earliest purchase date and time:

df.select(min("date")).show()

OpenAI

min()

Notice that the most recent and earliest purchases were made on the same day just a few hours apart. This means that the dataset we downloaded contains information of only purchases made on a single day.

Step 4: Data Pre-processing

Now that we have analyzed the dataset and have a better understanding of each data point, we need to prepare the data to feed into the machine learning algorithm.

Let’s take a look at the head of the dataframe once again to understand how the pre-processing will be done:

df.show(5,0)

OpenAI

pre-processing

From the dataset above, we need to create multiple customer segments based on each user’s purchase behavior.

The variables in this dataset are in a format that cannot be easily ingested into the customer segmentation model. These features individually do not tell us much about customer purchase behavior.

Due to this, we will use the existing variables to derive three new informative features - recency, frequency, and monetary value (RFM).

RFM is commonly used in marketing to evaluate a client’s value based on their:

Recency: How recently has each customer made a purchase?

Frequency: How often have they bought something?

Monetary Value: How much money do they spend on average when making purchases?

We will now preprocess the dataframe to create the above variables.

Recency

First, let’s calculate the value of recency - the latest date and time a purchase was made on the platform. This can be achieved in two steps:

i) Assign a recency score to each customer

We will subtract every date in the dataframe from the earliest date. This will tell us how recently a customer was seen in the dataframe. A value of 0 indicates the lowest recency, as it will be assigned to the person who was seen making a purchase on the earliest date.

df = df.withColumn("from\_date", lit("12/1/10 08:26"))

df = df.withColumn('from\_date',to\_timestamp("from\_date", 'yy/MM/dd HH:mm'))

df2=df.withColumn('from\_date',to\_timestamp(col('from\_date'))).withColumn('recency',col("date").cast("long") - col('from\_date').cast("long"))

OpenAI

ii) Select the most recent purchase

One customer can make multiple purchases at different times. We need to select only the last time they were seen buying a product, as this is indicative of when the most recent purchase was made:

df2 = df2.join(df2.groupBy('CustomerID').agg(max('recency').alias('recency')),on='recency',how='leftsemi')

OpenAI

Let’s look at the head of the new dataframe. It now has a variable called “recency” appended to it:

df2.show(5,0)

OpenAI

recency

An easier way to view all the variables present in a PySpark dataframe is to use its printSchema() function. This is the equivalent of the info() function in Pandas:

df2.printSchema()

OpenAI

The output rendered should look like this:

output rendered

Frequency

Let’s now calculate the value of frequency - how often a customer bought something on the platform. To do this, we just need to group by each customer ID and count the number of items they purchased:

df\_freq = df2.groupBy('CustomerID').agg(count('InvoiceDate').alias('frequency'))

OpenAI

Look at the head of this new dataframe we just created:

df\_freq.show(5,0)

OpenAI

new dataframe

There is a frequency value appended to each customer in the dataframe. This new dataframe only has two columns, and we need to join it with the previous one:

df3 = df2.join(df\_freq,on='CustomerID',how='inner')

OpenAI

Let’s print the schema of this dataframe:

df3.printSchema()

OpenAI

schema of this dataframe

Monetary Value

Finally, let’s calculate monetary value - the total amount spent by each customer in the dataframe. There are two steps to achieving this:

i) Find the total amount spent in each purchase:

Each customerID comes with variables called “Quantity” and “UnitPrice” for a single purchase:

customerID

To get the total amount spent by each customer in one purchase, we need to multiply “Quantity” with “UnitPrice”:

m\_val = df3.withColumn('TotalAmount',col("Quantity") \* col("UnitPrice"))

OpenAI

ii) Find the total amount spent by each customer:

To find the total amount spent by each customer overall, we just need to group by the CustomerID column and sum the total amount spent:

m\_val = m\_val.groupBy('CustomerID').agg(sum('TotalAmount').alias('monetary\_value'))

OpenAI

Merge this dataframe with the all the other variables:

finaldf = m\_val.join(df3,on='CustomerID',how='inner')

OpenAI

Now that we have created all the necessary variables to build the model, run the following lines of code to select only the required columns and drop duplicate rows from the dataframe:

finaldf = finaldf.select(['recency','frequency','monetary\_value','CustomerID']).distinct()

OpenAI

Look at the head of the final dataframe to ensure that the pre-processing has been done accurately:

final dataframe

Standardization

Before building the customer segmentation model, let’s standardize the dataframe to ensure that all the variables are around the same scale:

from pyspark.ml.feature import VectorAssembler

from pyspark.ml.feature import StandardScaler

assemble=VectorAssembler(inputCols=[

'recency','frequency','monetary\_value'

], outputCol='features')

assembled\_data=assemble.transform(finaldf)

scale=StandardScaler(inputCol='features',outputCol='standardized')

data\_scale=scale.fit(assembled\_data)

data\_scale\_output=data\_scale.transform(assembled\_data)

OpenAI

Run the following lines of code to see what the standardized feature vector looks like:

data\_scale\_output.select('standardized').show(2,truncate=False)

OpenAI

standardized feature vector

These are the scaled features that will be fed into the clustering algorithm.

If you’d like to learn more about data preparation with PySpark, take this feature engineering course on Datacamp.

Step 5: Building the Machine Learning Model

Now that we have completed all the data analysis and preparation, let’s build the K-Means clustering model.

The algorithm will be created using PySpark’s machine learning API.

i) Finding the number of clusters to use

When building a K-Means clustering model, we first need to determine the number of clusters or groups we want the algorithm to return. If we decide on three clusters, for instance, then we will have three customer segments.

The most popular technique used to decide on how many clusters to use in K-Means is called the “elbow-method.”

This is done simply running the K-Means algorithm for a wide range of clusters and visualizing the model results for each cluster. The plot will have an inflection point that looks like an elbow, and we just pick the number of clusters at this point.

Read this Datacamp K-Means clustering tutorial to learn more about how the algorithm works.

Let’s run the following lines of code to build a K-Means clustering algorithm from 2 to 10 clusters:

from pyspark.ml.clustering import KMeans

from pyspark.ml.evaluation import ClusteringEvaluator

import numpy as np

cost = np.zeros(10)

evaluator = ClusteringEvaluator(predictionCol='prediction', featuresCol='standardized',metricName='silhouette', distanceMeasure='squaredEuclidean')

for i in range(2,10):

KMeans\_algo=KMeans(featuresCol='standardized', k=i)

KMeans\_fit=KMeans\_algo.fit(data\_scale\_output)

output=KMeans\_fit.transform(data\_scale\_output)

cost[i] = KMeans\_fit.summary.trainingCost

OpenAI

With the codes above, we have successfully built and evaluated a K-Means clustering model with 2 to 10 clusters. The results have been placed in an array, and can now be visualized in a line chart:

import pandas as pd

import pylab as pl

df\_cost = pd.DataFrame(cost[2:])

df\_cost.columns = ["cost"]

new\_col = range(2,10)

df\_cost.insert(0, 'cluster', new\_col)

pl.plot(df\_cost.cluster, df\_cost.cost)

pl.xlabel('Number of Clusters')

pl.ylabel('Score')

pl.title('Elbow Curve')

pl.show()

OpenAI

The codes above will render the following chart:

Elbow curve

ii) Building the K-Means Clustering Model

From the plot above, we can see that there is an inflection point that looks like an elbow at four. Due to this, we will proceed to build the K-Means algorithm with four clusters:

KMeans\_algo=KMeans(featuresCol='standardized', k=4)

KMeans\_fit=KMeans\_algo.fit(data\_scale\_output)

OpenAI

iii) Making Predictions

Let’s use the model we created to assign clusters to each customer in the dataset:

preds=KMeans\_fit.transform(data\_scale\_output)

preds.show(5,0)

OpenAI

Notice that there is a “prediction” column in this dataframe that tells us which cluster each CustomerID belongs to:

prediction

Step 6: Cluster Analysis

The final step in this entire tutorial is to analyze the customer segments we just built.

Run the following lines of code to visualize the recency, frequency, and monetary value of each customerID in the dataframe:

import matplotlib.pyplot as plt

import seaborn as sns

df\_viz = preds.select('recency','frequency','monetary\_value','prediction')

df\_viz = df\_viz.toPandas()

avg\_df = df\_viz.groupby(['prediction'], as\_index=False).mean()

list1 = ['recency','frequency','monetary\_value']

for i in list1:

sns.barplot(x='prediction',y=str(i),data=avg\_df)

plt.show()

OpenAI

The codes above will render the following plots:

chart 3

chart1

chart 2

Here is an overview of characteristics displayed by customers in each cluster:

Cluster 0: Customers in this segment display low recency, frequency, and monetary value. They rarely shop on the platform and are low potential customers who are likely to stop doing business with the ecommerce company.

Cluster 1: Users in this cluster display high recency but haven’t been seen spending much on the platform. They also don’t visit the site often. This indicates that they might be newer customers who have just started doing business with the company.

Cluster 2: Customers in this segment display medium recency and frequency and spend a lot of money on the platform. This indicates that they tend to buy high-value items or make bulk purchases.

Cluster 3: The final segment comprises users who display high recency and make frequent purchases on the platform. However, they don’t spend much on the platform, which might mean that they tend to select cheaper items in each purchase.

To go beyond the predictive modelingmodelling concepts covered in this course, you can take the Machine Learning with PySpark course on Datacamp.

Learning PySpark From Scratch - Next Steps:

If you managed to follow along with this entire PySpark tutorial, congratulations! You have now successfully installed PySpark onto your local device, analyzed an e-commerce dataset, and built a machine learning algorithm using the framework.

One caveat of the analysis above is that it was conducted with 2,500 rows of ecommerce data collected on a single day. The outcome of this analysis can be solidified if we had a larger amount of data to work with, as techniques like RFM modeling are usually applied onto months of historical data.

However, you can take the principles learned in this article and apply them to a wide variety of larger datasets in the unsupervised machine learning space.

Check out this cheat sheet by Datacamp to learn more about PySpark’s syntax and its modules.

Finally, if you’d like to go beyond the concepts covered in this tutorial and learn the fundamentals of programming with PySpark, you can take the Big Data with PySpark learning track on Datacamp. This track contains a series of courses that will teach you to do the following with PySpark:

Data Management, Analysis, and Pre-processing

Building and Tuning Machine Learning Pipelines

Big Data Analysis

Feature Engineering

Building Recommendation Engines

HDFS: Hadoop Distributed File System

HIVE: Data warehouse that helps in reading, writing, and managing large datasets

PIG: helps create applications that run on Hadoop, allowing to execute jobs in MapReduce

MapReduce: System used for processing large data sets

YARN: Yet Another Resource Negotiator

Spark: Popular analytics engine that works in-memory

Oozie: Open-source workflow scheduling program

Zookeeper: Centralized service for maintaining config info, naming, providing distributed synchronization, and more

Mahout: Helps create ML applications

Pydoop is a Python interface to Hadoop that allows you to write MapReduce applications in pure Python:

class Mapper(api.Mapper):

def map(self, context):

for w in context.value.split():

context.emit(w, 1)

class Reducer(api.Reducer):

def reduce(self, context):

context.emit(context.key, sum(context.values))

Feature highlights:

a rich HDFS API;

a MapReduce API that allows to write pure Python record readers / writers, partitioners and combiners;

transparent Avro (de)serialization.

Pydoop enables MapReduce programming via a pure (except for a performance-critical serialization section) Python client for Hadoop Pipes, and HDFS access through an extension module based on libhdfs.

Pydoop Script makes it easy to solve simple problems. It makes it feasible to write simple (even throw-away) scripts to perform simple manipulations or analyses on your data, especially if it’s text-based. If you can specify your algorithm in two simple functions that have no state or have a simple state that can be stored in module variables, then you can consider using Pydoop Script. If, on the other hand, you need more sophisticated processing, consider using the full Pydoop API.

Let's get this out of the way. No, Hadoop isn't in itself a data storage system. Hadoop actually has several components, including MapReduce and the Hadoop Distributed File System (HDFS).

So Pydoop is on this list, but you'll need to pair Hadoop with other layers (such as Hive) to more easily wrangle data.

Pydoop is a Hadoop-Python interface that allows you to interact with the HDFS API and write MapReduce jobs using pure Python code.

This library allows the developer to access important MapReduce functions, such as RecordReader and Partitioner, without needing to know Java. For this last example, I think the people at Edureka do it better than I could. So here's a great quick intro.

Find The Intro Here

Pydoop itself might be a little too low-level for most data engineers. More than likely, most of you will be writing ETLs in Airflow that run on top of these systems. But it's still great to at least get a general understanding of what you are working with.

While Pydoop Script allows to solve many problems with minimal programming effort, some tasks require a broader set of features. If your data is not simple text with one record per line, for instance, you may need to write a record reader; if you need to change the way intermediate keys are assigned to reducers, you have to write your own partitioner. These components are accessible via the Pydoop MapReduce API.

The rest of this section serves as an introduction to MapReduce programming with Pydoop; the API reference has all the details.

Mappers and Reducers

The Pydoop API is object-oriented: the application developer writes a Mapper class, whose core job is performed by the map() method, and a Reducer class that processes data via the reduce() method. The following snippet shows how to write the mapper and reducer for wordcount, an application that counts the occurrence of each word in a text data set:

import pydoop.mapreduce.api as api

import pydoop.mapreduce.pipes as pipes

class Mapper(api.Mapper):

def map(self, context):

for w in context.value.split():

context.emit(w, 1)

class Reducer(api.Reducer):

def reduce(self, context):

context.emit(context.key, sum(context.values))

FACTORY = pipes.Factory(Mapper, reducer\_class=Reducer)

def main():

pipes.run\_task(FACTORY)

if \_\_name\_\_ == "\_\_main\_\_":

main()

The mapper is instantiated by the MapReduce framework that, for each input record, calls the map method passing a context object to it. The context serves as a communication interface between the framework and the application: in the map method, it is used to get the current key (not used in the above example) and value, and to emit (send back to the framework) intermediate key-value pairs. The reducer works in a similar way, the main difference being the fact that the reduce method gets a set of values for each key. The context has several other functions that we will explore later.

To run the above program, save it to a wc.py file and execute:

pydoop submit --upload-file-to-cache wc.py wc input output

Where input is the HDFS input directory.

See the section on running Pydoop programs for more details. Source code for the word count example is located under examples/pydoop\_submit/mr in the Pydoop distribution.

Counters and Status Updates

Hadoop features application-wide counters that can be set and incremented by developers. Status updates are arbitrary text messages sent to the framework: these are especially useful in cases where the computation associated with a single input record can take a considerable amount of time, since Hadoop kills tasks that read no input, write no output and do not update the status within a configurable amount of time (ten minutes by default).

The following snippet shows how to modify the above example to use counters and status updates:

class Mapper(api.Mapper):

def \_\_init\_\_(self, context):

super(Mapper, self).\_\_init\_\_(context)

context.set\_status("initializing mapper")

self.input\_words = context.get\_counter("WORDCOUNT", "INPUT\_WORDS")

def map(self, context):

words = context.value.split()

for w in words:

context.emit(w, 1)

context.increment\_counter(self.input\_words, len(words))

class Reducer(api.Reducer):

def \_\_init\_\_(self, context):

super(Reducer, self).\_\_init\_\_(context)

context.set\_status("initializing reducer")

self.output\_words = context.get\_counter("WORDCOUNT", "OUTPUT\_WORDS")

def reduce(self, context):

context.emit(context.key, sum(context.values))

context.increment\_counter(self.output\_words, 1)

Counter values and status updates show up in Hadoop’s web interface. In addition, the final values of all counters are listed in the command line output of the job (note that the list also includes Hadoop’s default counters).

Record Readers and Writers

By default, Hadoop assumes you want to process plain text and splits input data into text lines. If you need to process binary data, or your text data is structured into records that span multiple lines, you need to write your own RecordReader. The record reader operates at the HDFS file level: its job is to read data from the file and feed it as a stream of key-value pairs (records) to the mapper. To interact with HDFS files, we need to import the hdfs submodule:

import pydoop.hdfs as hdfs

The following example shows how to write a record reader that mimics Hadoop’s default LineRecordReader, where keys are byte offsets with respect to the whole file and values are text lines:

class Reader(api.RecordReader):

"""

Mimics Hadoop's default LineRecordReader (keys are byte offsets with

respect to the whole file; values are text lines).

"""

def \_\_init\_\_(self, context):

super(Reader, self).\_\_init\_\_(context)

self.logger = LOGGER.getChild("Reader")

self.logger.debug('started')

self.isplit = context.input\_split

for a in "filename", "offset", "length":

self.logger.debug(

"isplit.{} = {}".format(a, getattr(self.isplit, a))

)

self.file = hdfs.open(self.isplit.filename)

self.file.seek(self.isplit.offset)

self.bytes\_read = 0

if self.isplit.offset > 0:

discarded = self.file.readline()

self.bytes\_read += len(discarded)

def close(self):

self.logger.debug("closing open handles")

self.file.close()

self.file.fs.close()

def next(self):

if self.bytes\_read > self.isplit.length:

raise StopIteration

key = self.isplit.offset + self.bytes\_read

record = self.file.readline()

if not record: # end of file

raise StopIteration

self.bytes\_read += len(record)

return (key, record.decode("utf-8"))

def get\_progress(self):

return min(float(self.bytes\_read) / self.isplit.length, 1.0)

From the context, the record reader gets the following information on the byte chunk assigned to the current task, or input split:

the name of the file it belongs to;

its offset with respect to the beginning of the file;

its length.

This allows to open the file, seek to the correct offset and read until the end of the split is reached. The framework gets the record stream by means of repeated calls to the next() method. The get\_progress() method is called by the framework to get the fraction of the input split that’s already been processed. The close method (present in all components except for the partitioner) is called by the framework once it has finished retrieving the records: this is the right place to perform cleanup tasks such as closing open handles.

To use the reader, pass the class object to the factory with record\_reader\_class=Reader and, when running the program with pydoop submit, set the --do-not-use-java-record-reader flag.

The record writer writes key/value pairs to output files. The default behavior is to write one tab-separated key/value pair per line; if you want to do something different, you have to write a custom RecordWriter:

class Writer(api.RecordWriter):

def \_\_init\_\_(self, context):

super(Writer, self).\_\_init\_\_(context)

self.logger = LOGGER.getChild("Writer")

jc = context.job\_conf

outfn = context.get\_default\_work\_file()

self.logger.info("writing to %s", outfn)

hdfs\_user = jc.get("pydoop.hdfs.user", None)

self.file = hdfs.open(outfn, "wt", user=hdfs\_user)

self.sep = jc.get("mapreduce.output.textoutputformat.separator", "\t")

def close(self):

self.logger.debug("closing open handles")

self.file.close()

self.file.fs.close()

def emit(self, key, value):

self.file.write(key + self.sep + str(value) + "\n")

The above example, which simply reproduces the default behavior, also shows how to get job configuration parameters: the one starting with mapreduce is a standard Hadoop parameter, while pydoop.hdfs.user is a custom parameter defined by the application developer. Configuration properties are passed as -D <key>=<value> (e.g., -D mapreduce.output.textoutputformat.separator='|') to the submitter.

To use the writer, pass the class object to the factory with record\_writer\_class=Writer and, when running the program with pydoop submit, set the --do-not-use-java-record-writer flag.

Partitioners and Combiners

The Partitioner assigns intermediate keys to reducers. If you do not explicitly set a partitioner via the factory, partitioning will be done on the Java side. By default, Hadoop uses HashPartitioner, which selects the reducer on the basis of a hash function of the key.

To write a custom partitioner in Python, subclass Partitioner, overriding the partition() method. The framework will call this method with the current key and the total number of reducers N as the arguments, and expect the chosen reducer ID — in the [0, ..., N-1] range — as the return value.

The following examples shows how to write a partitioner that simply mimics the default HashPartitioner behavior:

from hashlib import md5

class Partitioner(api.Partitioner):

def \_\_init\_\_(self, context):

super(Partitioner, self).\_\_init\_\_(context)

self.logger = LOGGER.getChild("Partitioner")

def partition(self, key, n\_reduces):

reducer\_id = int(md5(key).hexdigest(), 16) % n\_reduces

self.logger.debug("reducer\_id: %r" % reducer\_id)

return reducer\_id

The combiner is functionally identical to a reducer, but it is run locally, on the key-value stream output by a single mapper. Although nothing prevents the combiner from processing values differently from the reducer, the former, provided that the reduce function is associative and idempotent, is typically configured to be the same as the latter, in order to perform local aggregation and thus help cut down network traffic.

Local aggregation is implemented by caching intermediate key/value pairs in a dictionary. Like in standard Java Hadoop, cache size is controlled by mapreduce.task.io.sort.mb and defaults to 100 MB. Pydoop uses sys.getsizeof() to determine key/value size, which takes into account Python object overhead. This can be quite substantial (e.g., sys.getsizeof(b"foo") == 36) and must be taken into account if fine tuning is desired.

Important

Due to the caching, when using a combiner there are limitations on the types that can be used for intermediate keys and values. First of all, keys must be hashable. In addition, values belonging to a mutable type should not change after having been emitted by the mapper. For instance, the following (however contrived) example would not work as expected:

intermediate\_value = {}

class Mapper(api.Mapper):

def map(self, ctx):

intermediate\_value.clear()

intermediate\_value[ctx.key] = ctx.value

ctx.emit("foo", intermediate\_value)

For these reasons, it is recommended to use immutable types for both keys and values when the job includes a combiner.

Custom partitioner and combiner classes must be declared to the factory as done above for record readers and writers. To recap, if we need to use all of the above components, we need to instantiate the factory as:

FACTORY = pipes.Factory(

Mapper,

reducer\_class=Reducer,

record\_reader\_class=Reader,

record\_writer\_class=Writer,

partitioner\_class=Partitioner,

combiner\_class=Reducer

)

Then the web app will be made by using Flask and deployed locally or publicly as per the need of the user.

# Python program to create Blockchain

# For timestamp

import datetime

# Calculating the hash

# in order to add digital

# fingerprints to the blocks

import hashlib

# To store data

# in our blockchain

import json

# Flask is for creating the web

# app and jsonify is for

# displaying the blockchain

from flask import Flask, jsonify

class Blockchain:

# This function is created

# to create the very first

# block and set its hash to "0"

def \_\_init\_\_(self):

self.chain = []

self.create\_block(proof=1, previous\_hash='0')

# This function is created

# to add further blocks

# into the chain

def create\_block(self, proof, previous\_hash):

block = {'index': len(self.chain) + 1,

'timestamp': str(datetime.datetime.now()),

'proof': proof,

'previous\_hash': previous\_hash}

self.chain.append(block)

return block

# This function is created

# to display the previous block

def print\_previous\_block(self):

return self.chain[-1]

# This is the function for proof of work

# and used to successfully mine the block

def proof\_of\_work(self, previous\_proof):

new\_proof = 1

check\_proof = False

while check\_proof is False:

hash\_operation = hashlib.sha256(

str(new\_proof\*\*2 - previous\_proof\*\*2).encode()).hexdigest()

if hash\_operation[:5] == '00000':

check\_proof = True

else:

new\_proof += 1

return new\_proof

def hash(self, block):

encoded\_block = json.dumps(block, sort\_keys=True).encode()

return hashlib.sha256(encoded\_block).hexdigest()

def chain\_valid(self, chain):

previous\_block = chain[0]

block\_index = 1

while block\_index < len(chain):

block = chain[block\_index]

if block['previous\_hash'] != self.hash(previous\_block):

return False

previous\_proof = previous\_block['proof']

proof = block['proof']

hash\_operation = hashlib.sha256(

str(proof\*\*2 - previous\_proof\*\*2).encode()).hexdigest()

if hash\_operation[:5] != '00000':

return False

previous\_block = block

block\_index += 1

return True

# Creating the Web

# App using flask

app = Flask(\_\_name\_\_)

# Create the object

# of the class blockchain

blockchain = Blockchain()

# Mining a new block

@app.route('/mine\_block', methods=['GET'])

def mine\_block():

previous\_block = blockchain.print\_previous\_block()

previous\_proof = previous\_block['proof']

proof = blockchain.proof\_of\_work(previous\_proof)

previous\_hash = blockchain.hash(previous\_block)

block = blockchain.create\_block(proof, previous\_hash)

response = {'message': 'A block is MINED',

'index': block['index'],

'timestamp': block['timestamp'],

'proof': block['proof'],

'previous\_hash': block['previous\_hash']}

return jsonify(response), 200

# Display blockchain in json format

@app.route('/get\_chain', methods=['GET'])

def display\_chain():

response = {'chain': blockchain.chain,

'length': len(blockchain.chain)}

return jsonify(response), 200

# Check validity of blockchain

@app.route('/valid', methods=['GET'])

def valid():

valid = blockchain.chain\_valid(blockchain.chain)

if valid:

response = {'message': 'The Blockchain is valid.'}

else:

response = {'message': 'The Blockchain is not valid.'}

return jsonify(response), 200

# Run the flask server locally

A

Creating Blockchain using Python, mining new blocks, and displaying the whole blockchain:

The data will be stored in JSON format which is very easy to implement and easy to read. The data is stored in a block and the block contains multiple data. Each and every minute multiple blocks are added and to differentiate one from the other we will use fingerprinting.

The fingerprinting is done by using hash and to be particular we will use the SHA256 hashing algorithm. Every block will contain its own hash and also the hash of the previous function so that it cannot get tampered with.

This fingerprinting will be used to chain the blocks together. Every block will be attached to the previous block having its hash and to the next block by giving its hash.

The mining of the new block is done by giving successfully finding the answer to the proof of work. To make mining hard the proof of work must be hard enough to get exploited.

After mining the block successfully the block will then be added to the chain.

After mining several blocks the validity of the chain must be checked in order to prevent any kind of tampering with the blockchain.