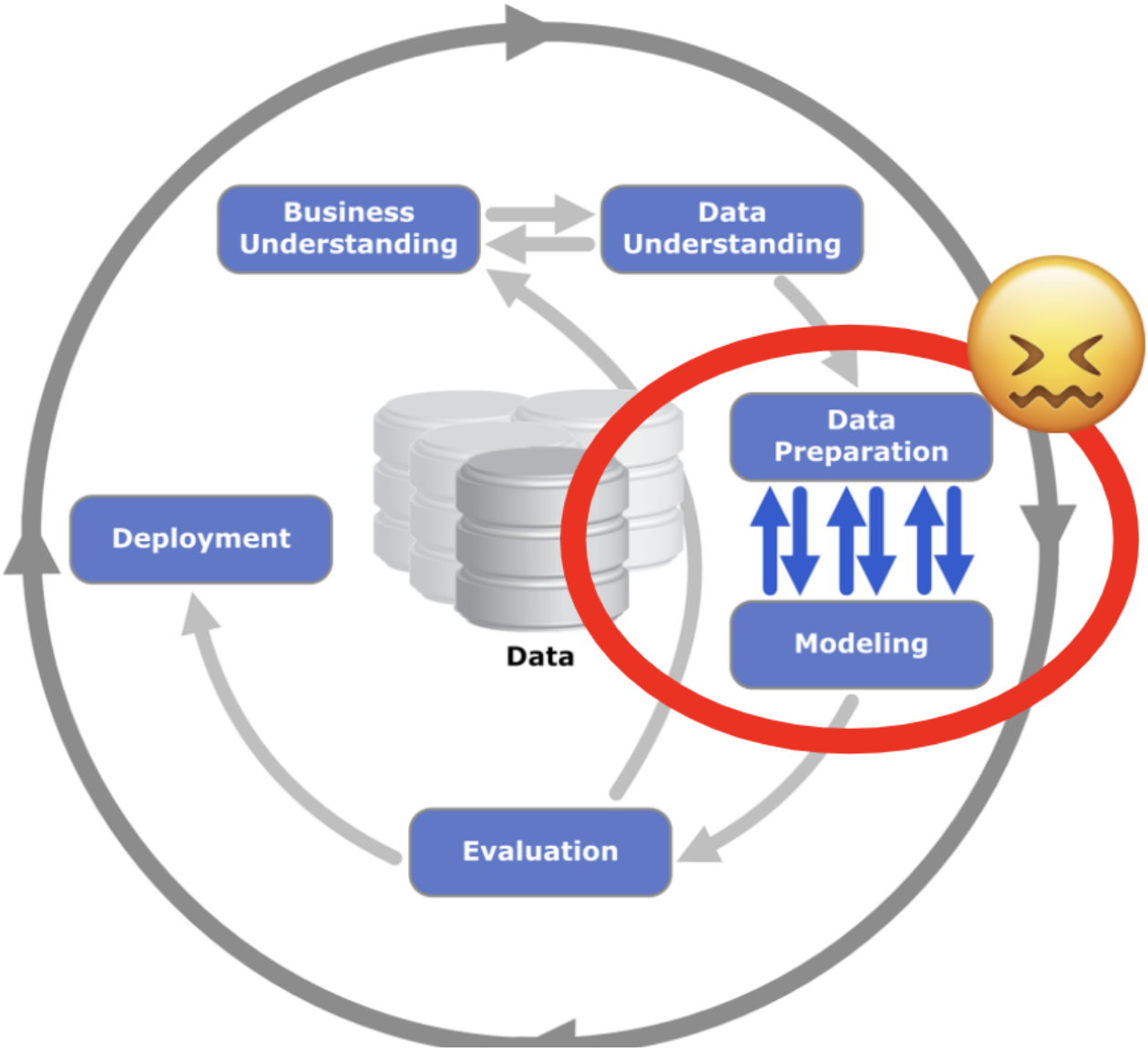
Every Data Scientist needs some SparkMagic

How to improve your data exploration and advanced analytics with the help of Spark Magic

Jan Teichmann Follow
May 21 · 7 min read ★



While data science is touted as the sexiest job of the 21st century, it is not spared of the infamous **Pareto principle** or **80/20 rule**. 80% of a commercial data scientist's time is spent on finding, cleansing, and preparing data. It is the least productive as well as the most dreaded part of a data scientist's job.

The internet offers endless opinions on how to break the 80/20 rule for data science but good advice is nevertheless hard to come by. A main source for the low productivity lies in the **duality of data preparation**:

- Access, join and aggregate big data stored in enterprise data lakes at speed
- Explore and visualise data and statistics in Notebooks with complex dependencies for Python packages

Big data is mostly unstructured and stored in **production environments** with enterprise governance and security restrictions in place. Accessing the data at speed requires costly distributed systems which are **centrally managed** by IT and have to be commonly shared with other data scientists and analysts.

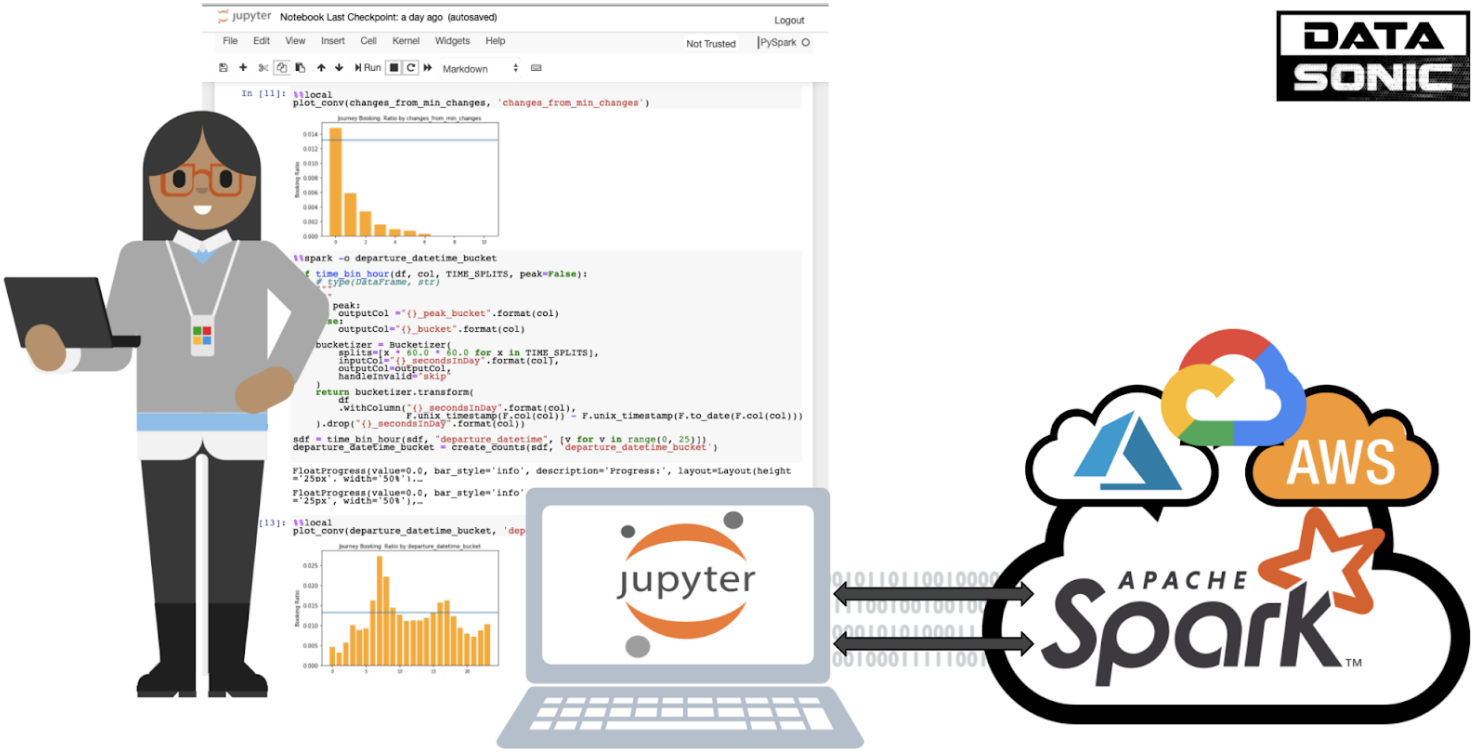
Spark is the data industries gold standard for working with data in distributed data lakes. But to work with Spark clusters cost-efficiently, and even allow multi-tenancy, it is difficult to accommodate individual requirements and dependencies. The industry trend for distributed data infrastructure is towards **ephemeral clusters** which makes it even harder for data scientists to deploy and manage their **Jupyter notebook environments**.

It's no surprise that many data scientists **work locally** on high-spec laptops where they can install and persist their Jupyter notebook environments more easily. So far so understandable. How do many data scientists then connect their local development environment with the data in the production data lake? They materialise csv files with Spark and download them from the cloud storage console.

Manually downloading csv files from cloud storage consoles is neither productive nor is it particularly robust. Wouldn't it be so much better to seamlessly connect a local Jupyter

Notebook with a remote cluster in an end-user friendly and transparent way? Meet SparkMagic!

SparkMagic for Jupyter Notebooks



Fair usage and public domain icons and svg by Sandro Pereira, MIT licensed

Sparkmagic is a project to interactively work with remote Spark clusters in Jupyter notebooks through the Livy REST API. It provides a set of Jupyter Notebook cell magics and kernels to turn Jupyter into an integrated Spark environment for remote clusters.

SparkMagic allows us to

- Run Spark code in **multiple languages** and
- Automatically creates a SparkSession with SparkContext and HiveContext against any **remote Spark cluster**
- Provides automatic **visualisation** of SQL queries
- Easily access Spark application logs and information

- Capture the output of Spark queries as a **local Pandas data frame** to interact easily with other Python libraries (e.g. matplotlib)
- **Send local files** or Pandas data frames to a remote cluster (e.g. sending pre-trained local ML model straight to the Spark cluster)

You can use the following Dockerfile to build a Jupyter Notebook with SparkMagic support:

```
FROM jupyter/all-spark-notebook:7a0c7325e470

USER $NB_USER
RUN pip install --upgrade pip
RUN pip install --upgrade --ignore-installed setuptools
RUN pip install pandas --upgrade
RUN pip install sparkmagic
RUN mkdir /home/$NB_USER/.sparkmagic
RUN wget https://raw.githubusercontent.com/jupyter-
incubator/sparkmagic/master/sparkmagic/example_config.json
RUN mv example_config.json /home/$NB_USER/.sparkmagic/config.json
RUN sed -i 's/localhost:8998/host.docker.internal:9999/g'
/home/$NB_USER/.sparkmagic/config.json
RUN jupyter nbextension enable --py --sys-prefix widgetsnbextension
RUN jupyter-kernelspec install --user --name SparkMagic $(pip show
sparkmagic | grep Location | cut -d" " -
f2)/sparkmagic/kernels/sparkkernel
RUN jupyter-kernelspec install --user --name PySparkMagic $(pip show
sparkmagic | grep Location | cut -d" " -
f2)/sparkmagic/kernels/pysparkkernel
RUN jupyter serverextension enable --py sparkmagic
USER root
RUN chown $NB_USER /home/$NB_USER/.sparkmagic/config.json
CMD ["start-notebook.sh", "--
NotebookApp.iopub_data_rate_limit=1000000000"]
USER $NB_USER
```

Build the image and tag it with:

```
docker build -t sparkmagic
```

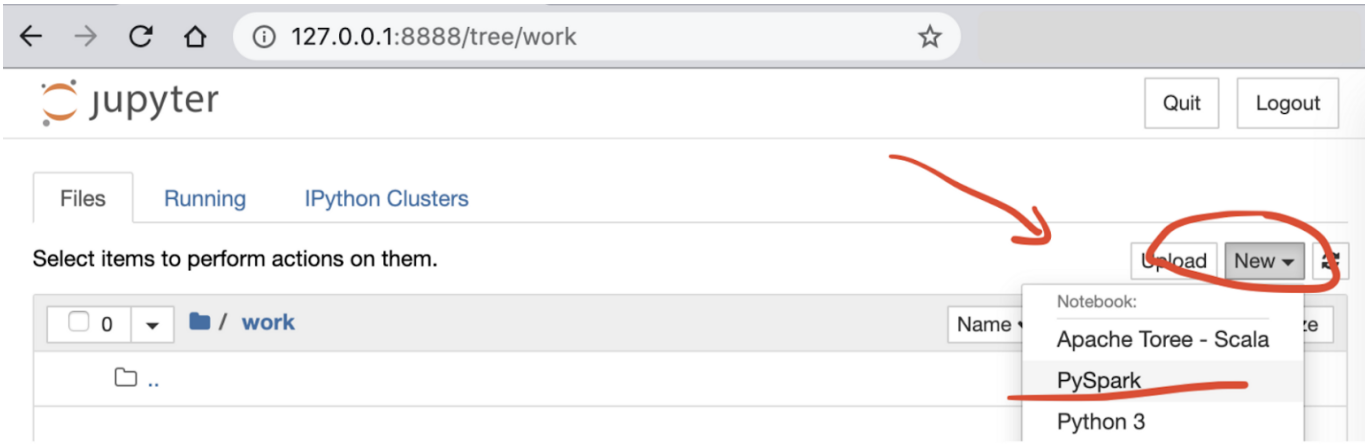
And start a local Jupyter container with Spark Magic support mounting your current working directory:

```
docker run -ti --name \"${PWD##*/}-pyspark\" -p 8888:8888 --rm -m
4GB --mount type=bind,source=\"${PWD}\",target=/home/jovyan/work
sparkmagic
```

To be able to connect to the Livy REST API on your remote Spark cluster you have to use ssh port forwarding on your local computer. Get the IP address of your remote cluster and run:

```
ssh -L 0.0.0.0:9999:localhost:8998 REMOTE_CLUSTER_IP
```

First, create a new Notebook with the SparkMagic enabled PySpark kernel as follows:



[OC]

In the SparkMagic enabled Notebook, you have a series of cell magics available to work across the local notebook as well as your remote Spark cluster as an integrated environment. The **%%help** magic prints out all the available magic commands:

```
In [8]: %%help
```

Magic	Example	Explanation
info	%%info	Outputs session information for the current Livy endpoint.
cleanup	%%cleanup -f	Deletes all sessions for the current Livy endpoint, including this notebook's session. The force flag is mandatory.
delete	%%delete -f -s 0	Deletes a session by number for the current Livy endpoint. Cannot delete this kernel's session.
logs	%%logs	Outputs the current session's Livy logs.

configure	%%configure -f {"executorMemory": "1000M", "executorCores": 4}	Configure the session creation parameters. The force flag is mandatory if a session has already been created and the session will be dropped and recreated. Look at Livy's POST /sessions Request Body for a list of valid parameters. Parameters must be passed in as a JSON string.
spark	%%spark -o df df = spark.read.parquet('...	Executes spark commands. Parameters: • -o VAR_NAME: The Spark dataframe of name VAR_NAME will be available in the %%local Python context as a Pandas dataframe with the same name. • -m METHOD: Sample method, either `take` or `sample`. • -n MAXROWS: The maximum number of rows of a dataframe that will be pulled from Livy to Jupyter. If this number is negative, then the number of rows will be unlimited. • -r FRACTION: Fraction used for sampling.
sql	%%sql -o tables -q SHOW TABLES	Executes a SQL query against the variable sqlContext (Spark v1.x) or spark (Spark v2.x). Parameters: • -o VAR_NAME: The result of the SQL query will be available in the %%local Python context as a Pandas dataframe. • -q: The magic will return None instead of the dataframe (no visualization). • -m, -n, -r are the same as the %%spark parameters above.
local	%%local a = 1	All the code in subsequent lines will be executed locally. Code must be valid Python code.
send_to_spark	%%send_to_spark -o variable -t str -n var	Sends a variable from local output to spark cluster. Parameters: • -i VAR_NAME: Local Pandas DataFrame(or String) of name VAR_NAME will be available in the %%spark context as a Spark dataframe(or String) with the same name. • -t TYPE: Specifies the type of variable passed as -i. Available options are: `str` for string and `df` for Pandas DataFrame. Optional, defaults to `str`. • -n NAME: Custom name of variable passed as -i. Optional, defaults to -i variable name. • -m MAXROWS: Maximum amount of Pandas rows that will be sent to Spark. Defaults to 2500.

[OC]

You can configure your remote Spark Application with the **%%configure** magic:

```
In [6]: %%configure -f
{
    "executorMemory": "2G",
    "executorCores": 1,
    "driverMemory": "5G",
    "driverCores": 2

}
```

```
Starting Spark application
```

ID	YARN Application ID	Kind	State	Spark UI	Driver log	Current session?
1	application_1589879398527_0002	pyspark	idle	Link	Link	✔

```
SparkSession available as 'spark'.
```

```
Current session configs: {'executorMemory': '2G', 'executorCores': 1, 'driverMemory': '5G', 'driverCores': 2, 'kind':
'pyspark'}
```

As you can see in the screenshot above the SparkMagic automatically started a remote PySpark Session and provides some useful links to connect to the Spark UI and logs.

The Notebook integrates 2 environments:

1. **%%local** which executes the cell locally on your laptop and the anaconda environment provided by the jupyter docker image
2. **%%spark** which executes the cell remotely via the PySpark REPL on your remote Spark cluster via the Livy REST API

The following code cell first imports the SparkSql types remotely. Secondly, it loads the Enigma-JHU Covid-19 data-set onto our remote Spark cluster using the remote SparkSession. We can see the output of the remote .show() command within our Notebook:

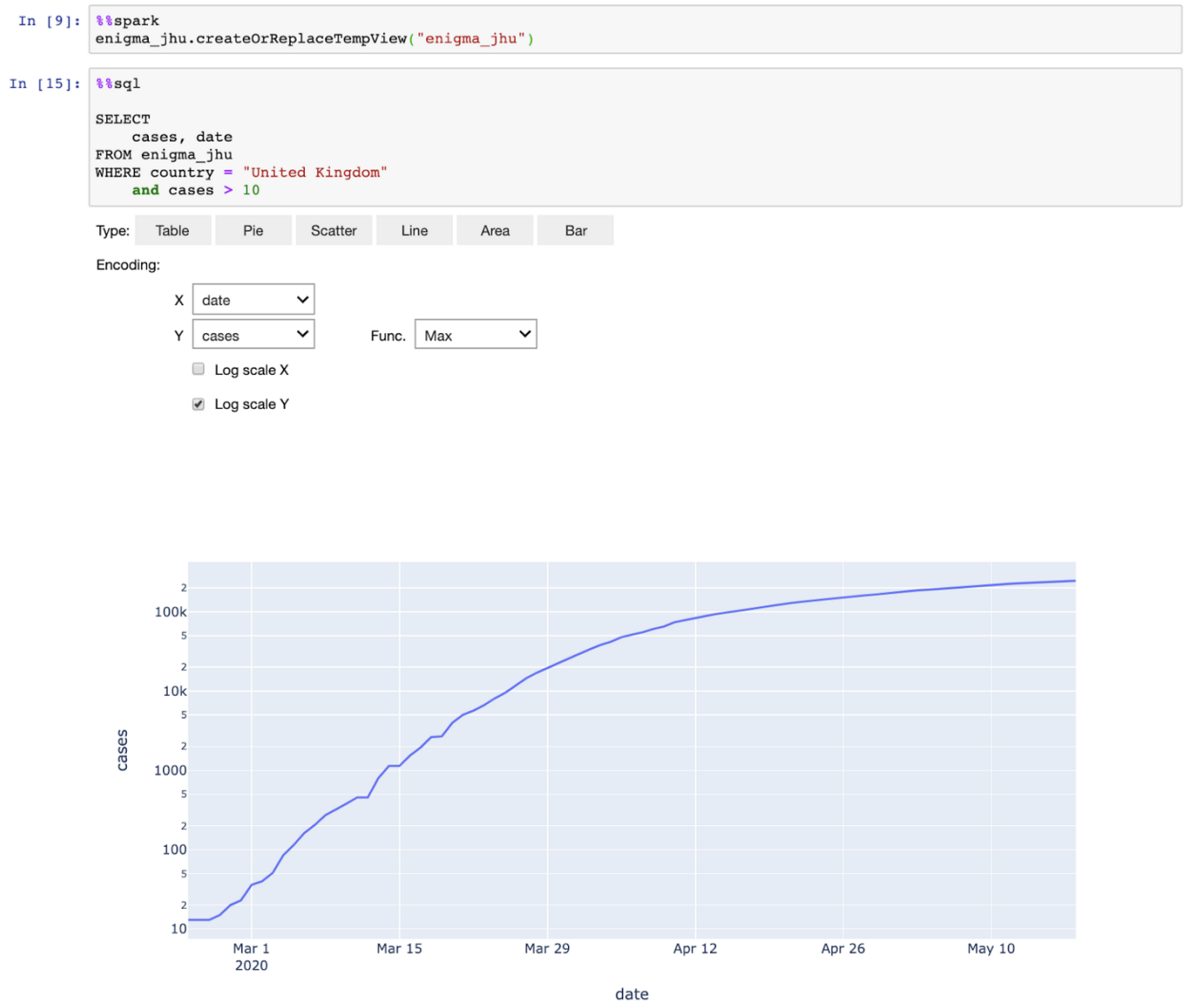
```
In [8]: %%spark
from pyspark.sql.types import *

enigma_jhu = (
    spark.read
    .options(header='true')
    .schema(
        StructType(
            [
                StructField("name", StringType(), True),
                StructField("level", StringType(), True),
                StructField("city", StringType(), True),
                StructField("county", StringType(), True),
                StructField("state", StringType(), True),
                StructField("country", StringType(), True),
                StructField("population", IntegerType(), True),
                StructField("lat", DoubleType(), True),
                StructField("long", DoubleType(), True),
                StructField("url", StringType(), True),
                StructField("aggregate", StringType(), True),
                StructField("tz", StringType(), True),
                StructField("cases", IntegerType(), True),
                StructField("deaths", IntegerType(), True),
                StructField("recovered", IntegerType(), True),
                StructField("active", IntegerType(), True),
                StructField("tested", IntegerType(), True),
                StructField("hospitalized", IntegerType(), True),
                StructField("discharged", IntegerType(), True),
                StructField("icu", IntegerType(), True),
                StructField("growthFactor", DoubleType(), True),
                StructField("date", DateType(), True),
            ]
        )
    )
    .csv("s3://covid19-lake/enigma-jhu/csv")
    .persist()
)
enigma_jhu.show(1)
```

```
+--------------------+------+----+-------+--------+-------+----------+-------+-----------------+--------------------+
---------+---------------+-----+------+---------+------+------+------------+----------+----+------------+----------+
|                name| level|city| county|   state|country|population|    lat|             long|                 url|
aggregate|             tz|cases|deaths|recovered|active|tested|hospitalized|discharged| icu|growthFactor|      date|
+--------------------+------+----+-------+--------+-------+----------+-------+-----------------+--------------------+
---------+---------------+-----+------+---------+------+------+------------+----------+----+------------+----------+
|Antwerp, Flanders...|county|null|Antwerp|Flanders|Belgium|   1847486|51.2485|4.717499999999999|https://epistat.w...|
null|Europe/Brussels|    4|  null|     null|  null|  null|        null|      null|null|        null|2020-01-22|
+--------------------+------+----+-------+--------+-------+----------+-------+-----------------+--------------------+
---------+---------------+-----+------+---------+------+------+------------+----------+----+------------+----------+
only showing top 1 row
```

[OC]

But that's just where the Magic begins. We can register the dataframe as a Hive table and use the **%%sql** magic to execute Hive queries against the data on our remote cluster and

create automated visualisations of the results in our local Notebook. While this isn't rocket science it is extremely convenient for data analysts and for quick data exploration in the early stages of a data science project.



A truly useful feature of SparkMagic is to seamless pass data between the local Notebook and the remote cluster. The daily challenge of data scientists is to create and persist their Python environments while working with ephemeral clusters to interact with their company's data lake.

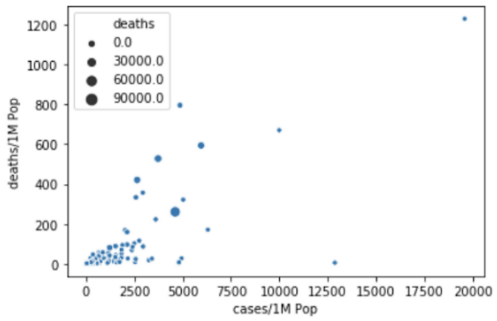
In the following example, you can see how we import seaborn as a local library and use it to plot covid_data pandas data frame. But where does that data come from? It's been created and send by the remote Spark cluster. The magic **%%spark -o** allows us to

define a remote variable to transfer to the local notebook context on cell execution. Our variable covid_data is a

- SparkSQL DataFrame on the remote cluster and a
- Pandas DataFrame in the local Jupyter Notebook

```
In [12]: %%spark -o covid_data
import pyspark.sql.functions as F

covid_data = (
    enigma_jhu
    .groupBy("country")
    .agg(
        F.max("cases").alias("cases"),
        F.max("deaths").alias("deaths"),
        F.max("population").alias("population"),
    )
    .withColumn(
        "cases/1M Pop",
        (F.col("cases") / (F.col("population"))) * F.lit(10.0**6)
    )
    .withColumn(
        "deaths/1M Pop",
        (F.col("deaths") / (F.col("population"))) * F.lit(10.0**6)
    )
)
```

```
In [15]: %%local
%matplotlib inline
import seaborn as sns

_ = sns.scatterplot(data=covid_data, x="cases/1M Pop", y="deaths/1M Pop", size="deaths")
```



The ability to aggregate big data in a remote cluster to work with locally in a Jupyter Notebook using Pandas is extremely helpful for data exploration. E.g. to use Spark to pre-aggregate data for a histogram into counts by bins to plot the histogram in Jupyter using the pre-aggregated counts and a simple bar plot. Another useful feature is the ability to sample a remote Spark DataFrame with the magic **%%spark -o covid_data -m sample -r 0.5**

The integrated environment allows you also to sent local data to the remote Spark cluster using the magic **%%send_to_spark**

The two data types supported are **Pandas DataFrames** and **strings**. To send anything more else or more complex, for example, a trained scikit model for scoring, to the

remote Spark cluster you can use serialisation to create a string representation for transfer:

```
import pickle
import gzip
import base64

serialised_model = base64.b64encode(
    gzip.compress(
        pickle.dumps(trained_scikit_model)
    )
).decode()
```

Deploying PySpark workloads to Production?

As you saw, there is **a big pain point** to this pattern of ephemeral PySpark clusters: **bootstrapping EMR clusters** with Python packages. This problem isn't going away when you deploy production workloads. You can read how to use PEX to speed up deployment of PySpark applications on ephemeral AWS EMR clusters in my previous blog post:

PEX — The secret sauce for the perfect PySpark deployment of AWS EMR workloads

How to use PEX to speed up deployment of PySpark applications on ephemeral AWS EMR clusters

towardsdatascience.com

Jan is a successful thought leader and consultant in the data transformation of companies and has a track record of bringing data science into commercial production usage at scale. He has recently been recognised by dataIQ as one of the 100 most influential data and analytics practitioners in the UK.

Connect on LinkedIn: https://www.linkedin.com/in/janteichmann/

Read other articles: https://medium.com/@jan.teichmann

Data Science Machine Learning Towards Data Science Spark Jupyter Notebook