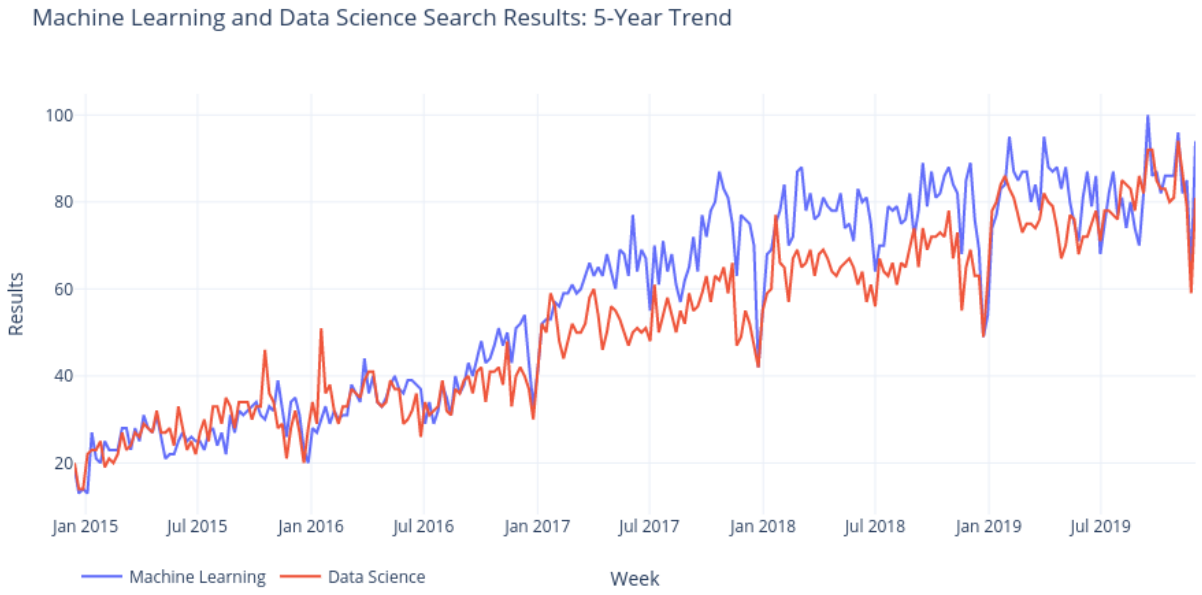
Getting Started with Data Analytics using Jupyter Notebooks, PySpark, and Docker

Gary A. Stafford
Dec 6, 2019 · 19 min read ★

Introduction

There is little question, big data analytics, data science, artificial intelligence (AI), and machine learning (ML), a subcategory of AI, have all experienced a tremendous surge in popularity over the last few years. Behind the marketing hype, these technologies are having a significant influence on many aspects of our modern lives. Due to their popularity and potential benefits, commercial enterprises, academic institutions, and the public sector are rushing to develop hardware and software solutions to lower the barriers to entry and increase the velocity of ML and Data Scientists and Engineers.



(courtesy Google Trends and Plotly)

Many open-source software projects are also lowering the barriers to entry into these technologies. An excellent example of one such open-source project working on this

challenge is Project Jupyter. Similar to Apache Zeppelin and the newly open-sourced Netflix's Polynote, Jupyter Notebooks enables data-driven, interactive, and collaborative data analytics.

This post will demonstrate the creation of a containerized data analytics environment using Jupyter Docker Stacks. The particular environment will be suited for learning and developing applications for Apache Spark using the Python, Scala, and R programming languages. We will focus on Python and Spark, using PySpark.

Featured Technologies

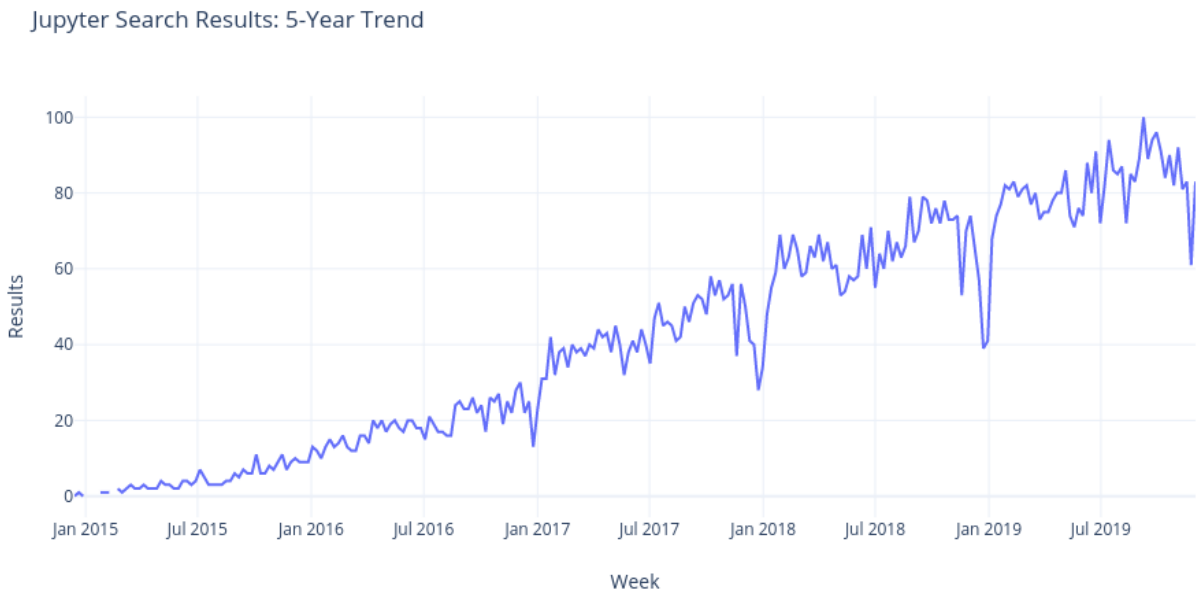
The following technologies are featured prominently in this post.

Jupyter Notebooks

According to Project Jupyter, the Jupyter Notebook, formerly known as the IPython Notebook, is an open-source web application that allows users to create and share documents that contain live code, equations, visualizations, and narrative text. Uses include data cleansing and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more. The word, Jupyter, is a loose acronym for **Ju***lia*, **Py***thon*, and **R**, but today, the Jupyter supports many programming languages.

Interest in Jupyter Notebooks has grown dramatically over the last 3–5 years, fueled in part by the major Cloud providers, AWS, Google Cloud, and Azure. Amazon Sagemaker, Amazon EMR (Elastic MapReduce), Google Cloud Dataproc, Google Colab

(Collaboratory), and Microsoft Azure Notebooks all have direct integrations with Jupyter notebooks for big data analytics and machine learning.



(courtesy Google Trends and Plotly)

Jupyter Docker Stacks

To enable quick and easy access to Jupyter Notebooks, Project Jupyter has created Jupyter Docker Stacks. The stacks are ready-to-run Docker images containing Jupyter applications, along with accompanying technologies. Currently, the Jupyter Docker Stacks focus on a variety of specializations, including the r-notebook, scipy-notebook, tensorflow-notebook, datascience-notebook, pyspark-notebook, and the subject of this post, the all-spark-notebook. The stacks include a wide variety of well-known packages to extend their functionality, such as scikit-learn, pandas, Matplotlib, Bokeh, NumPy, and Facets.

Apache Spark

According to Apache, Spark is a unified analytics engine for large-scale data processing. Starting as a research project at the UC Berkeley AMPLab in 2009, Spark was open-sourced in early 2010 and moved to the Apache Software Foundation in 2013. Reviewing the postings on any major career site will confirm that Spark is widely used by well-known modern enterprises, such as Netflix, Adobe, Capital One, Lockheed Martin, JetBlue Airways, Visa, and Databricks. At the time of this post, LinkedIn, alone, had approximately 3.5k listings for jobs that reference the use of Apache Spark, just in the United States.

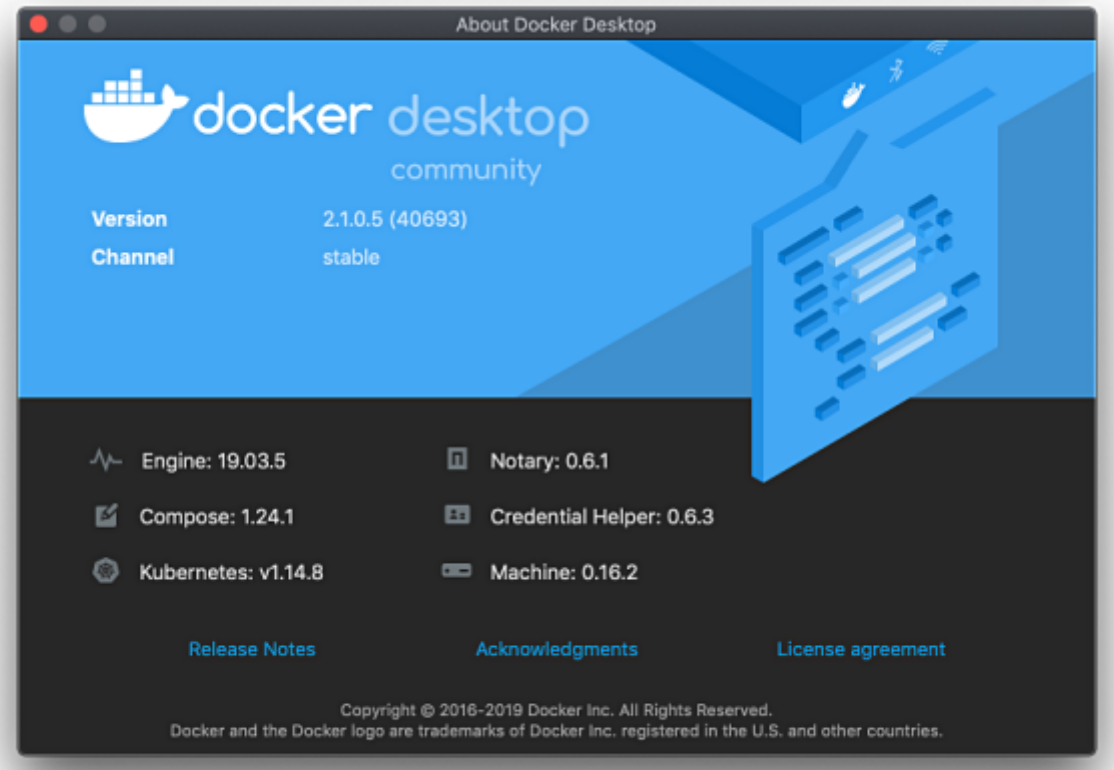
With speeds up to 100 times faster than Hadoop, Apache Spark achieves high performance for static, batch, and streaming data, using a state-of-the-art DAG (Directed Acyclic Graph) scheduler, a query optimizer, and a physical execution engine. Spark's polyglot programming model allows users to write applications quickly in Scala, Java, Python, R, and SQL. Spark includes libraries for Spark SQL (DataFrames and Datasets), MLlib (Machine Learning), GraphX (Graph Processing), and DStreams (Spark Streaming). You can run Spark using its standalone cluster mode, Apache Hadoop YARN, Mesos, or Kubernetes.

PySpark

The Spark Python API, PySpark, exposes the Spark programming model to Python. PySpark is built on top of Spark's Java API and uses Py4J. According to Apache, Py4J, a bridge between Python and Java, enables Python programs running in a Python interpreter to dynamically access Java objects in a Java Virtual Machine (JVM). Data is processed in Python and cached and shuffled in the JVM.

Docker

According to Docker, their technology gives developers and IT the freedom to build, manage, and secure business-critical applications without the fear of technology or infrastructure lock-in. For this post, I am using the current stable version of Docker Desktop Community version for macOS.



Docker Swarm

Current versions of Docker include both a Kubernetes and Swarm orchestrator for deploying and managing containers. We will choose Swarm for this demonstration. According to Docker, the cluster management and orchestration features embedded in the Docker Engine are built using swarmkit. Swarmkit is a separate project which implements Docker's orchestration layer and is used directly within Docker.

PostgreSQL

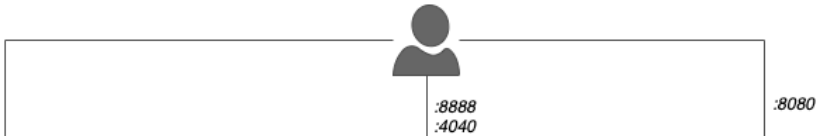
PostgreSQL is a powerful, open-source, object-relational database system. According to their website, PostgreSQL comes with many features aimed to help developers build applications, administrators to protect data integrity and build fault-tolerant environments, and help manage data no matter how big or small the dataset.

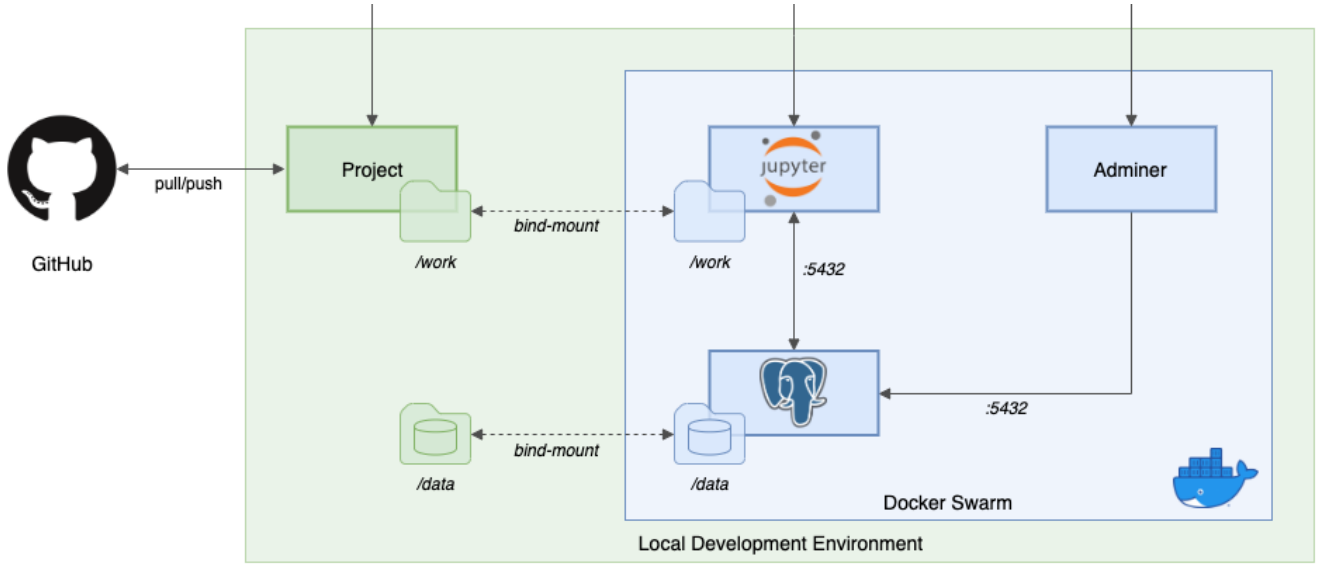
Demonstration

In this demonstration, we will explore the capabilities of the Spark Jupyter Docker Stack to provide an effective data analytics development environment. We will explore a few everyday uses, including executing Python scripts, submitting PySpark jobs, and working with Jupyter Notebooks, and reading and writing data to and from different file formats and a database. We will be using the latest `jupyter/all-spark-notebook` Docker Image. This image includes Python, R, and Scala support for Apache Spark, using Apache Toree.

Architecture

As shown below, we will deploy a Docker stack to a single-node Docker swarm. The stack consists of a Jupyter All-Spark-Notebook, PostgreSQL (Alpine Linux version 12), and Adminer container. The Docker stack will have two local directories bind-mounted into the containers. Files from our GitHub project will be shared with the Jupyter application container through a bind-mounted directory. Our PostgreSQL data will also be persisted through a bind-mounted directory. This allows us to persist data external to the ephemeral containers. If the containers are restarted or recreated, the data is preserved locally.





Source Code

All source code for this post can be found on GitHub. Use the following command to clone the project. Note this post uses the `v2` branch.

```
git clone \
  --branch v2 --single-branch --depth 1 --no-tags \
  https://github.com/garystafford/pyspark-setup-demo.git
```

Source code samples are displayed as GitHub Gists, which may not display correctly on some mobile and social media browsers.

Deploy Docker Stack

To start, create the `$HOME/data/postgres` directory to store PostgreSQL data files.

```
mkdir -p ~/data/postgres
```

This directory will be bind-mounted into the PostgreSQL container on line 41 of the stack.yml file, `$HOME/data/postgres:/var/lib/postgresql/data`. The `HOME` environment variable assumes you are working on Linux or macOS and is equivalent to `HOMEPATH` on Windows.

The Jupyter container's working directory is set on line 15 of the stack.yml file, `working_dir: /home/$USER/work`. The local bind-mounted working directory is `$PWD/work`. This path is bind-mounted to the working directory in the Jupyter

container, on line 29 of the stack.yml file, `$PWD/work:/home/$USER/work`. The `PWD` environment variable assumes you are working on Linux or macOS (`CD` on Windows).

By default, the user within the Jupyter container is `jovyan`. We will override that user with our own local host's user account, as shown on line 21 of the Docker stack file, `NB_USER: $USER`. We will use the host's `USER` environment variable value (equivalent to `USERNAME` on Windows). There are additional options for configuring the Jupyter container. Several of those options are used on lines 17–22 of the Docker stack file (*gist*).

```
1  # docker stack deploy -c stack.yml jupyter
2
3  version: "3.7"
4  networks:
5    demo-net:
6
7  services:
8    spark:
9      image: jupyter/all-spark-notebook:latest
10     ports:
11     - "8888:8888/tcp"
12     - "4040:4040/tcp"
13     networks:
14     - demo-net
15     working_dir: /home/$USER/work
16     environment:
17       CHOWN_HOME: "yes"
18       GRANT_SUDO: "yes"
19       NB_UID: 1000
20       NB_GID: 100
21       NB_USER: $USER
22       NB_GROUP: staff
23     user: root
24     deploy:
25      replicas: 1
26      restart_policy:
27        condition: on-failure
28     volumes:
29     - $PWD/work:/home/$USER/work
30   postgres:
31     image: postgres:12-alpine
32     environment:
33       POSTGRES_USERNAME: postgres
34       POSTGRES_PASSWORD: postgres1234
35       POSTGRES_DB: bakery
```

```
36      ports:
37      - "5432:5432/tcp"
38      networks:
39      - demo-net
40      volumes:
41      - $HOME/data/postgres:/var/lib/postgresql/data
42      deploy:
43       restart_policy:
44         condition: on-failure
45    adminer:
46      image: adminer:latest
47      ports:
48      - "8080:8080/tcp"
49      networks:
50      - demo-net
51      deploy:
52       restart_policy:
53         condition: on-failure
```

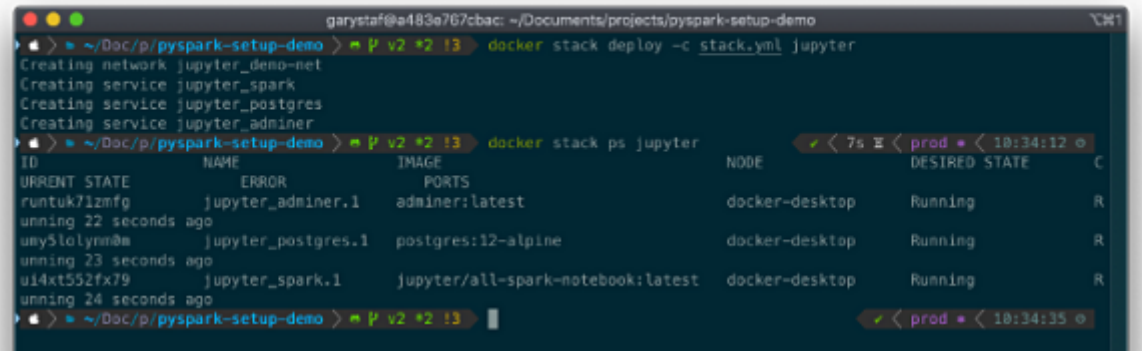
stack.yml hosted with ♡ by GitHub view raw

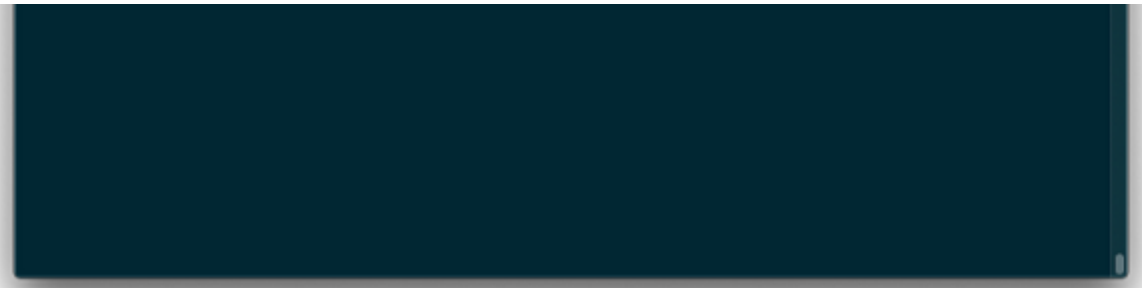
Assuming you have a recent version of Docker installed on your local development machine and running in swarm mode, standing up the stack is as easy as running the following docker command from the root directory of the project.

```
docker stack deploy -c stack.yml jupyter
```

The Docker stack consists of a new overlay network, `jupyter_demo-net`, and three containers. To confirm the stack deployed successfully, run the following docker command.

```
docker stack ps jupyter --no-trunc
```





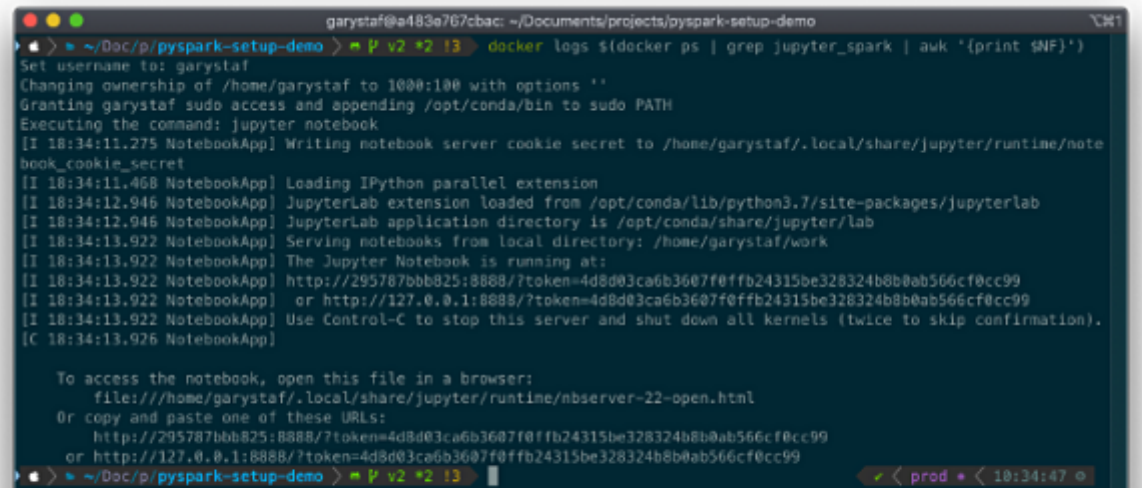
The `jupyter/all-spark-notebook` Docker image is large, approximately 5 GB. Depending on your Internet connection, if this is the first time you have pulled this image, the stack may take several minutes to enter a running state. Although not required, I usually pull Docker images in advance.

```
docker pull jupyter/all-spark-notebook:latest
docker pull postgres:12-alpine
docker pull adminer:latest
```

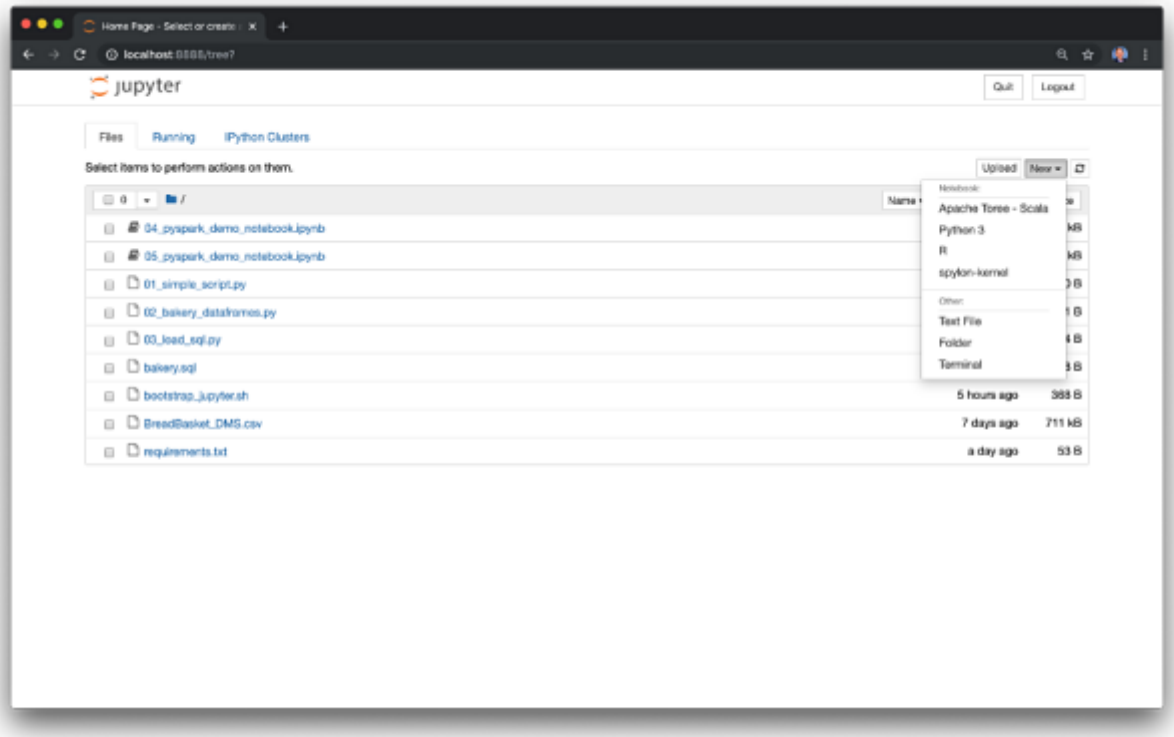
To access the Jupyter Notebook application, you need to obtain the Jupyter URL and access token. The Jupyter URL and the access token are output to the Jupyter container log, which can be accessed with the following command.

```
docker logs $(docker ps | grep jupyter_spark | awk '{print $NF}')
```

You should observe log output similar to the following. Retrieve the complete URL, for example, `http://127.0.0.1:8888/?token=f78cbe...`, to access the Jupyter web-based user interface.



From the Jupyter dashboard landing page, you should see all the files in the project's `work/` directory. Note the types of files you can create from the dashboard, including Python 3, R, and Scala (using Apache Toree or spylon-kernal) notebooks, and text. You can also open a Jupyter terminal or create a new Folder from the drop-down menu. At the time of this post, the latest `jupyter/all-spark-notebook` Docker Image runs Spark 2.4.4, Scala 2.11.12, Python 3.7.3, and OpenJDK 64-Bit Server VM, Java 1.8.0 Update 222.



Bootstrap the Environment

Included in the project is a bootstrap script, bootstrap_jupyter.sh. The script will install the required Python packages using pip, the Python package installer, and a requirement.txt file. The bootstrap script also installs the latest PostgreSQL driver JAR, configures Apache Log4j to reduce log verbosity when submitting Spark jobs, and installs htop. Although these tasks could also be done from a Jupyter terminal or from within a Jupyter notebook, using a bootstrap script ensures you will have a consistent

work environment every time you spin up the Jupyter Docker stack. Add or remove items from the bootstrap script as necessary (*gist*).

```
#!/bin/bash

set -ex

# install required python packages
python3 -m pip install --user --upgrade pip
python3 -m pip install -r requirements.txt --upgrade

# download latest postgres driver jar
POSTGRES_JAR="postgresql-42.2.8.jar"
if [ -f "$POSTGRES_JAR" ]; then
    echo "$POSTGRES_JAR exist"
else
    wget -nv "https://jdbc.postgresql.org/download/${POSTGRES_JAR}"
fi

# spark-submit logging level from INFO to WARN
sudo cp log4j.properties /usr/local/spark/conf/log4j.properties

# update/upgrade and install htop
sudo apt-get update -y && sudo apt-get upgrade -y
sudo apt-get install htop
```

bootstrap_jupyter.sh hosted with ♡ by GitHub view raw

That's it, our new Jupyter environment is ready to start exploring.

Running Python Scripts

One of the simplest tasks we could perform in our new Jupyter environment is running Python scripts. Instead of worrying about installing and maintaining the correct versions of Python and multiple Python packages on your own development machine, we can run Python scripts from within the Jupyter container. At the time of this post update, the latest `jupyter/all-spark-notebook` Docker image runs Python 3.7.3 and Conda 4.7.12. Let's start with a simple Python script, 01_simple_script.py.

```
#!/usr/bin/python3

import random

technologies = [
    'PySpark', 'Python', 'Spark', 'Scala', 'Java', 'Project Jupyter', 'R'
```

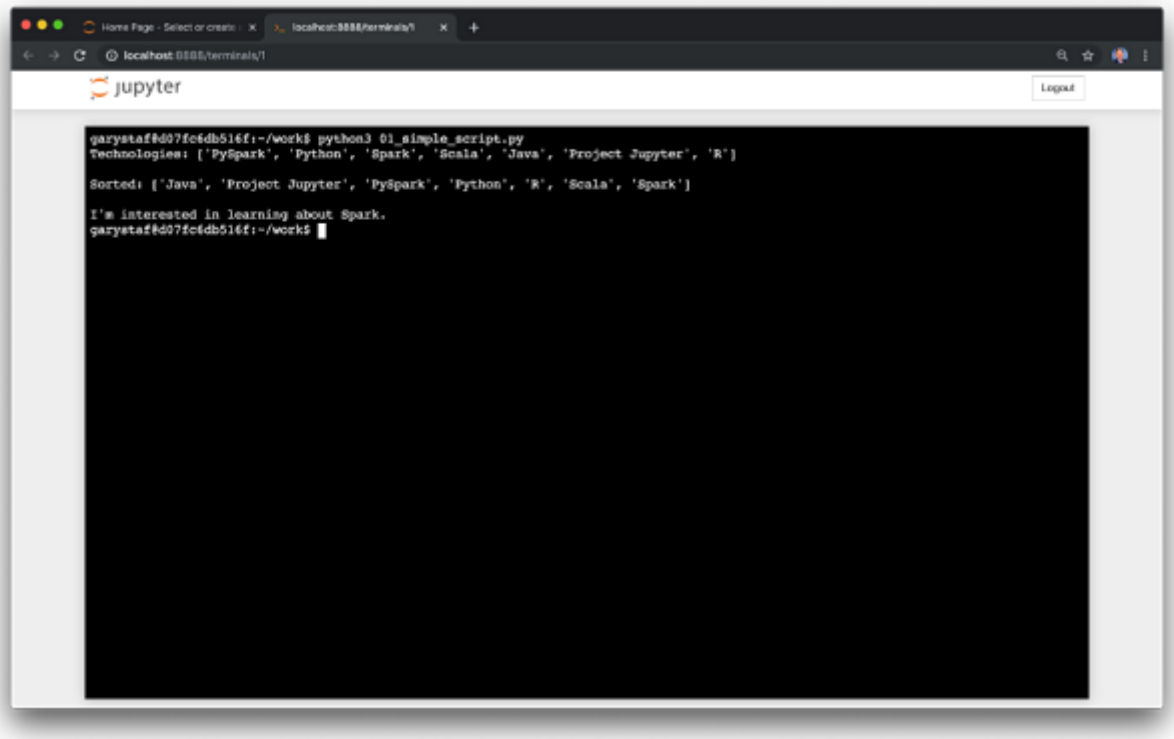
```
]

print("Technologies: %s\n" % technologies)

technologies.sort()
print("Sorted: %s\n" % technologies)

print("I'm interested in learning about %s." % random.choice(technologies))
```

01_simple_script.py hosted with ♡ by GitHub view raw

From a Jupyter terminal window, use the following command to run the script.

```
python3 01_simple_script.py
```

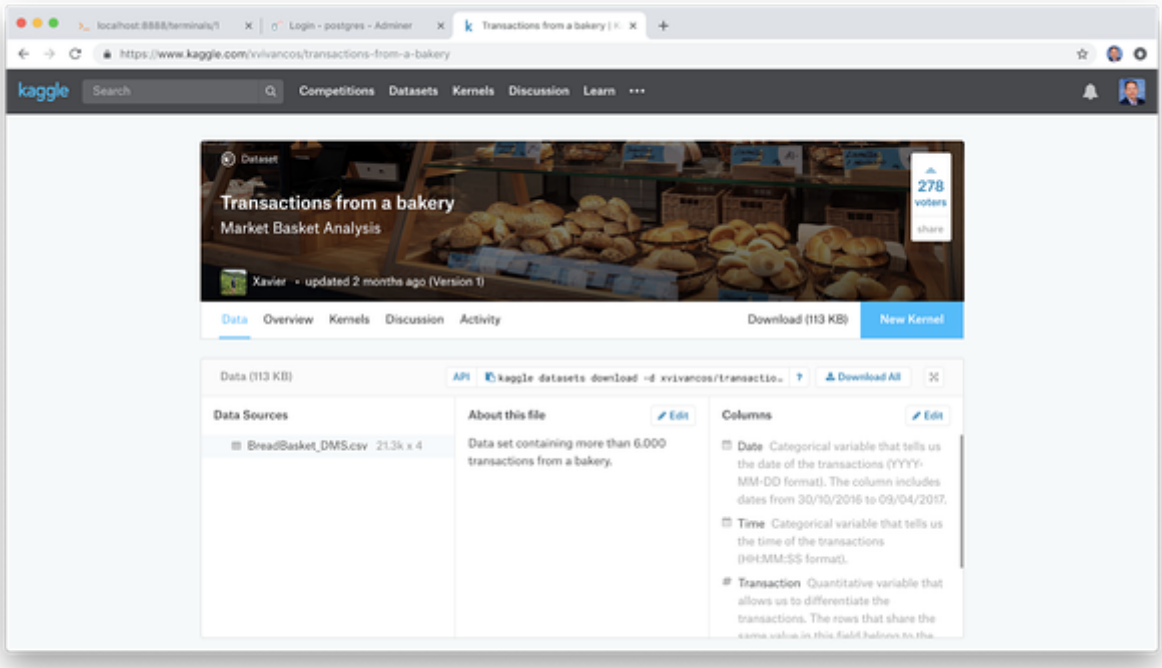
You should observe the following output.



Kaggle Datasets

To explore more features of the Jupyter and PySpark, we will use a publicly available dataset from Kaggle. Kaggle is an excellent open-source resource for datasets used for big-data and ML projects. Their tagline is '*Kaggle is the place to do data science projects*.' For this demonstration, we will use the Transactions from a bakery dataset from

Kaggle. The dataset is available as a single CSV-format file. A copy is also included in the project.



The 'Transactions from a bakery' dataset contains 21,294 rows with 4 columns of data. Although certainly not *big data*, the dataset is large enough to test out the Spark Jupyter Docker Stack functionality. The data consists of 9,531 customer transactions for 21,294 bakery items between 2016–10–30 and 2017–04–09 (*gist*).

Search this file…

Date	Time	Transaction	Item
2016-10-30	09:58:11	1	Bread
2016-10-30	10:05:34	2	Scandinavian
2016-10-30	10:05:34	2	Scandinavian
2016-10-30	10:07:57	3	Hot chocolate
2016-10-30	10:07:57	3	Jam
2016-10-30	10:07:57	3	Cookies
2016-10-30	10:08:41	4	Muffin
2016-10-30	10:13:03	5	Coffee
2016-10-30	10:13:03	5	Pastry
2016-10-30	10:13:03	5	Bread

bakery_data.csv hosted with ♡ by GitHub view raw

Submitting Spark Jobs

We are not limited to Jupyter notebooks to interact with Spark. We can also submit scripts directly to Spark from the Jupyter terminal. This is typically how Spark is used in a Production for performing analysis on large datasets, often on a regular schedule, using tools such as Apache Airflow. With Spark, you are load data from one or more data sources. After performing operations and transformations on the data, the data is persisted to a datastore, such as a file or a database, or conveyed to another system for further processing.

The project includes a simple Python PySpark ETL script, 02_pyspark_job.py. The ETL script loads the original Kaggle Bakery dataset from the CSV file into memory, into a Spark DataFrame. The script then performs a simple Spark SQL query, calculating the total quantity of each type of bakery item sold, sorted in descending order. Finally, the script writes the results of the query to a new CSV file, `output/items-sold.csv`.

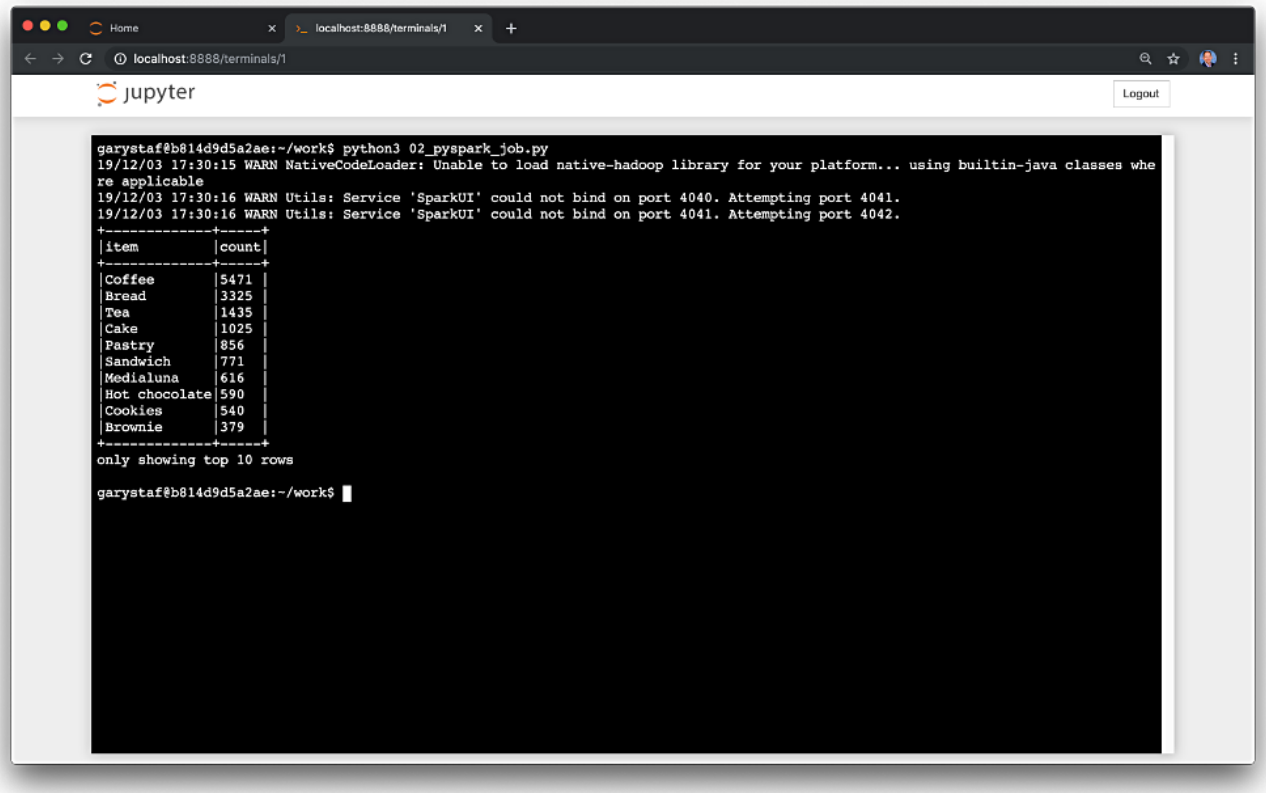
```
#!/usr/bin/python3

from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName('spark-demo') \
    .getOrCreate()

df_bakery = spark.read \
    .format('csv') \
    .option('header', 'true') \
    .option('delimiter', ',') \
    .option('inferSchema', 'true') \
    .load('BreadBasket_DMS.csv')

df_sorted = df_bakery.cube('item').count() \
    .filter('item NOT LIKE \'NONE\'') \
    .filter('item NOT LIKE \'Adjustment\'') \
    .orderBy(['count', 'item'], ascending=[False, True])

df_sorted.show(10, False)

df_sorted.coalesce(1) \
    .write.format('csv') \
    .option('header', 'true') \
    .save('output/items-sold.csv', mode='overwrite')
```

02_bakery_dataframes.py hosted with ♡ by GitHub view raw

Run the script directly from a Jupyter terminal using the following command.

```
python3 02_pyspark_job.py
```

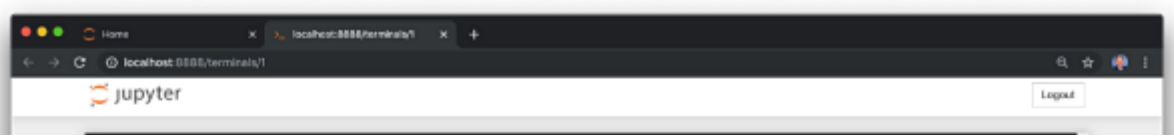
An example of the output of the Spark job is shown below.

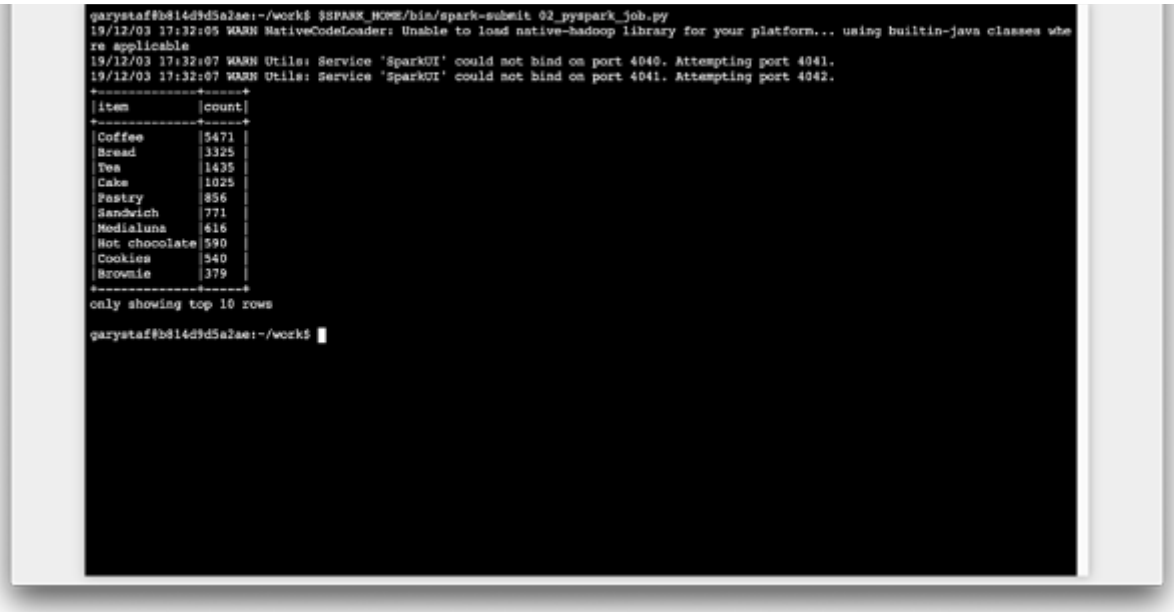


Typically, you would submit the Spark job using the `spark-submit` command. Use a Jupyter terminal to run the following command.

```
$SPARK_HOME/bin/spark-submit 02_pyspark_job.py
```

Below, we see the output from the `spark-submit` command. Printing the results in the output is merely for the purposes of the demo. Typically, Spark jobs are submitted non-interactively, and the results are persisted directly to a datastore or conveyed to another system.

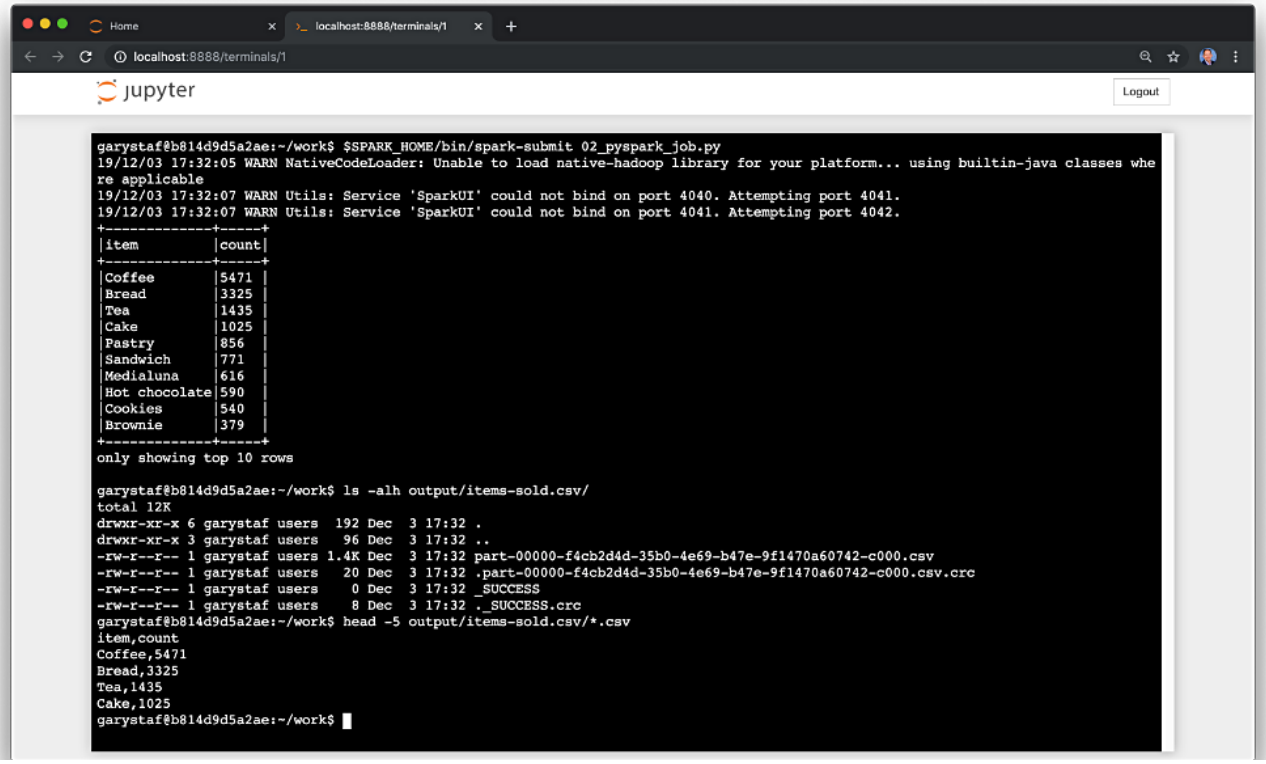




Using the following commands, we can view the resulting CVS file, created by the spark job.

```
ls -alh output/items-sold.csv/
head -5 output/items-sold.csv/*.csv
```

An example of the files created by the spark job is shown below. We should have discovered that coffee is the most commonly sold bakery item with 5,471 sales, followed by bread with 3,325 sales.



Interacting with Databases

To demonstrate the flexibility of Jupyter to work with databases, PostgreSQL is part of the Docker Stack. We can read and write data from the Jupyter container to the PostgreSQL instance, running in a separate container. To begin, we will run a SQL script, written in Python, to create our database schema and some test data in a new database table. To do so, we will use the psycopg2, the PostgreSQL database adapter package for the Python, we previously installed into our Jupyter container using the bootstrap script. The Python script, 03_load_sql.py, will execute a set of SQL statements contained in a SQL file, bakery.sql, against the PostgreSQL container instance.

```
#!/usr/bin/python3

import psycopg2

# connect to database
connect_str = 'host=postgres port=5432 dbname=bakery user=postgres password=postgres1234'
conn = psycopg2.connect(connect_str)
conn.autocommit = True
cursor = conn.cursor()

# execute sql script
sql_file = open('bakery.sql', 'r')
sqlFile = sql_file.read()
sql_file.close()
sqlCommands = sqlFile.split(';')
for command in sqlCommands:
    print(command)
    if command.strip() != '':
        cursor.execute(command)

# import data from csv file
with open('BreadBasket_DMS.csv', 'r') as f:
    next(f)  # Skip the header row.
    cursor.copy_from(
        f,
        'transactions',
        sep=',',
        columns=('date', 'time', 'transaction', 'item')
    )
    conn.commit()
```

```
31
32  # confirm by selecting record
33  command = 'SELECT COUNT(*) FROM public.transactions;'
34  cursor.execute(command)
35  recs = cursor.fetchall()
36  print('Row count: %d' % recs[0])
```

03 load sql.py hosted with ♡ by GitHub view raw

The SQL file, bakery.sql.

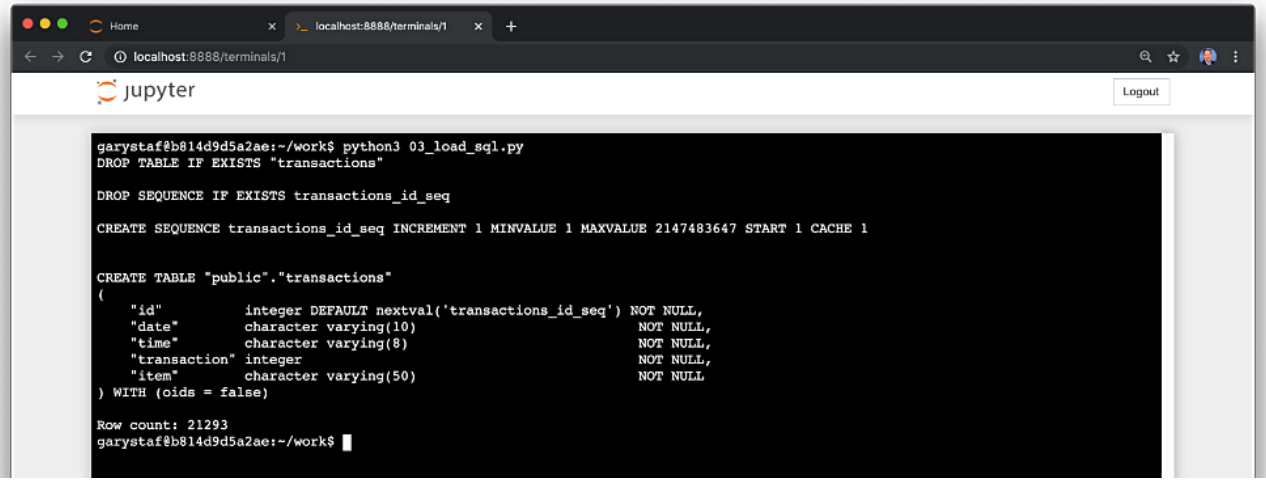
```
1   DROP TABLE IF EXISTS "transactions";
2   DROP SEQUENCE IF EXISTS transactions_id_seq;
3   CREATE SEQUENCE transactions_id_seq INCREMENT 1 MINVALUE 1 MAXVALUE 2147483647 START 1 CACHE 1;
4
5   CREATE TABLE "public"."transactions"
6   (
7       "id"          integer DEFAULT nextval('transactions_id_seq') NOT NULL,
8       "date"        character varying(10)                          NOT NULL,
9       "time"        character varying(8)                           NOT NULL,
10      "transaction" integer                                        NOT NULL,
11      "item"        character varying(50)                          NOT NULL
12  ) WITH (oids = false);
```

bakery.sql hosted with ♡ by GitHub view raw

To execute the script, run the following command.

```
python3 03_load_sql.py
```

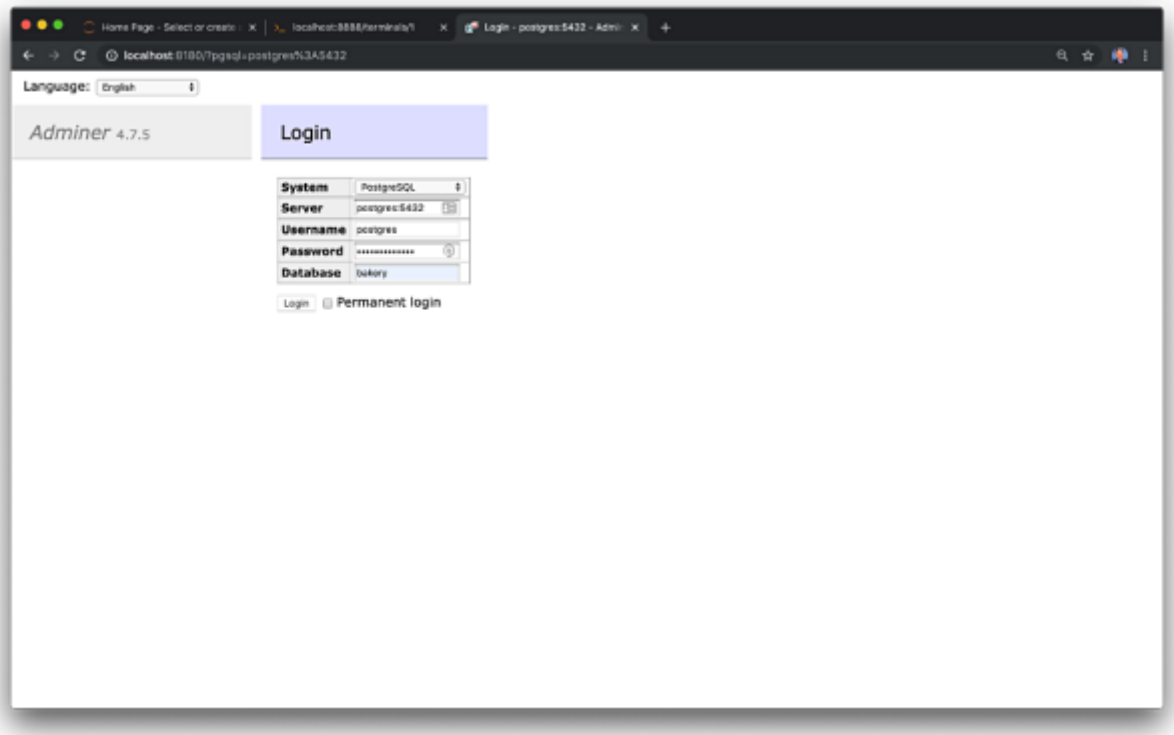
This should result in the following output, if successful.



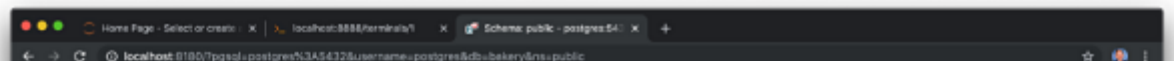
Adminer

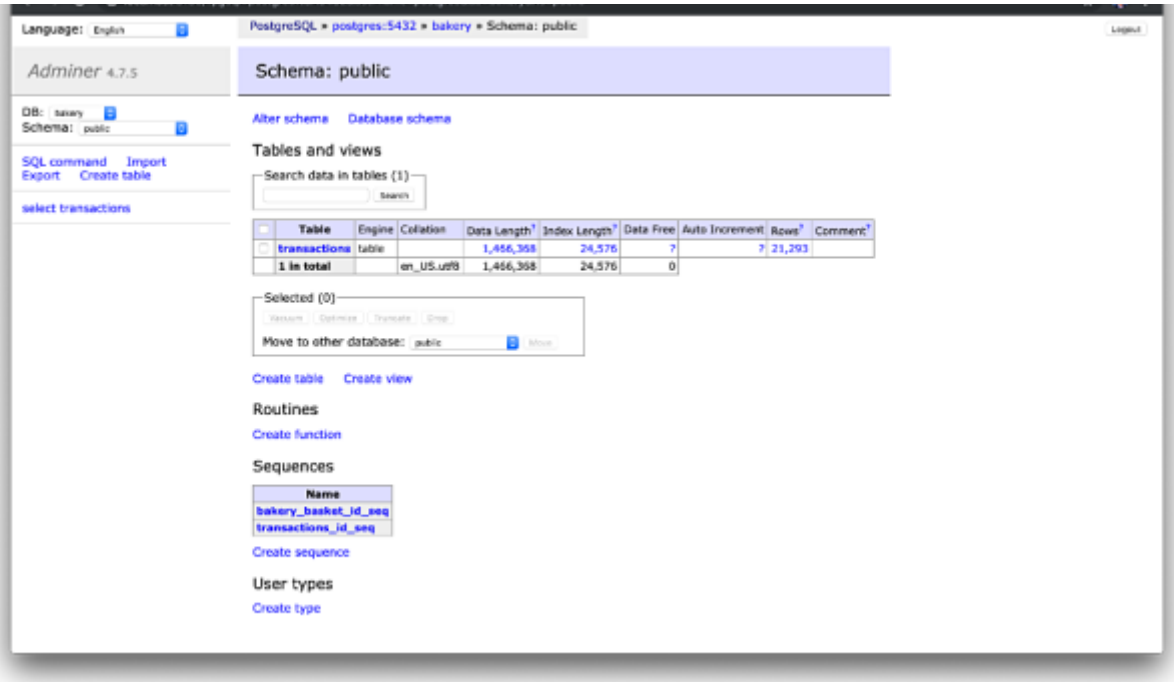
To confirm the SQL script's success, use Adminer. Adminer (*formerly phpMinAdmin*) is a full-featured database management tool written in PHP. Adminer natively recognizes PostgreSQL, MySQL, SQLite, and MongoDB, among other database engines.

Adminer should be available on localhost port 8080. The password credentials, shown below, are located in the stack.yml file. The server name, `postgres`, is the name of the PostgreSQL Docker container. This is the domain name the Jupyter container will use to communicate with the PostgreSQL container.

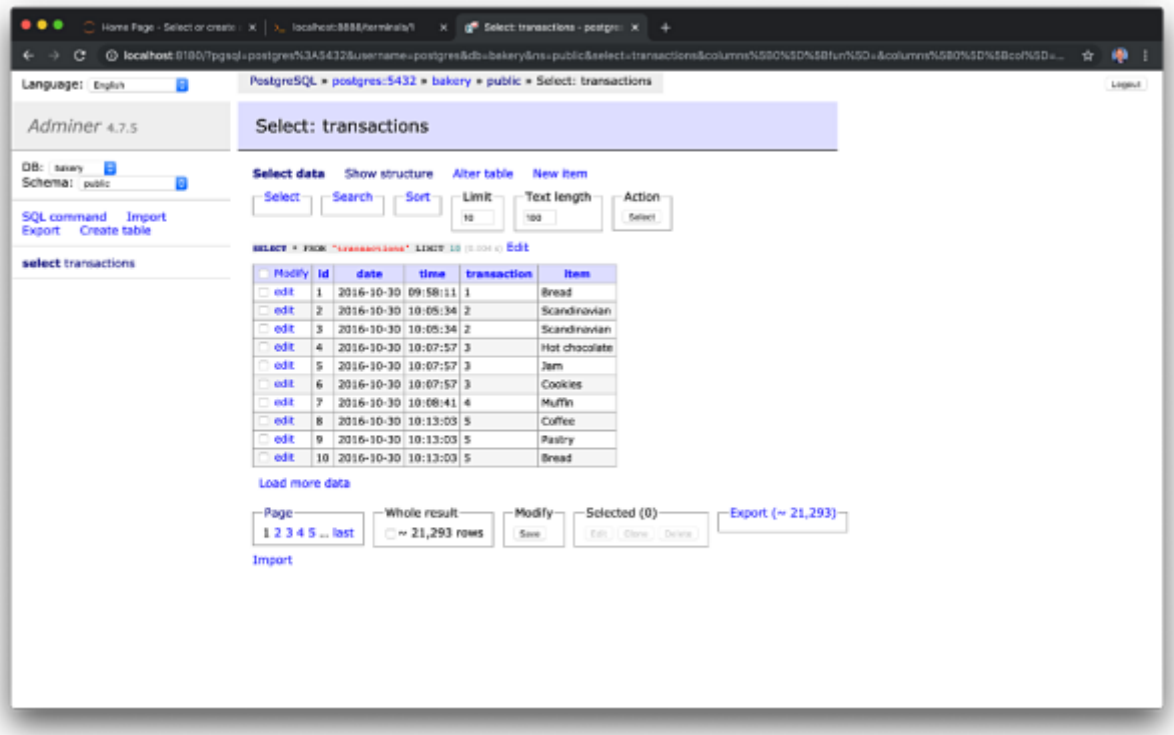


Connecting to the new `bakery` database with Adminer, we should see the `transactions` table.





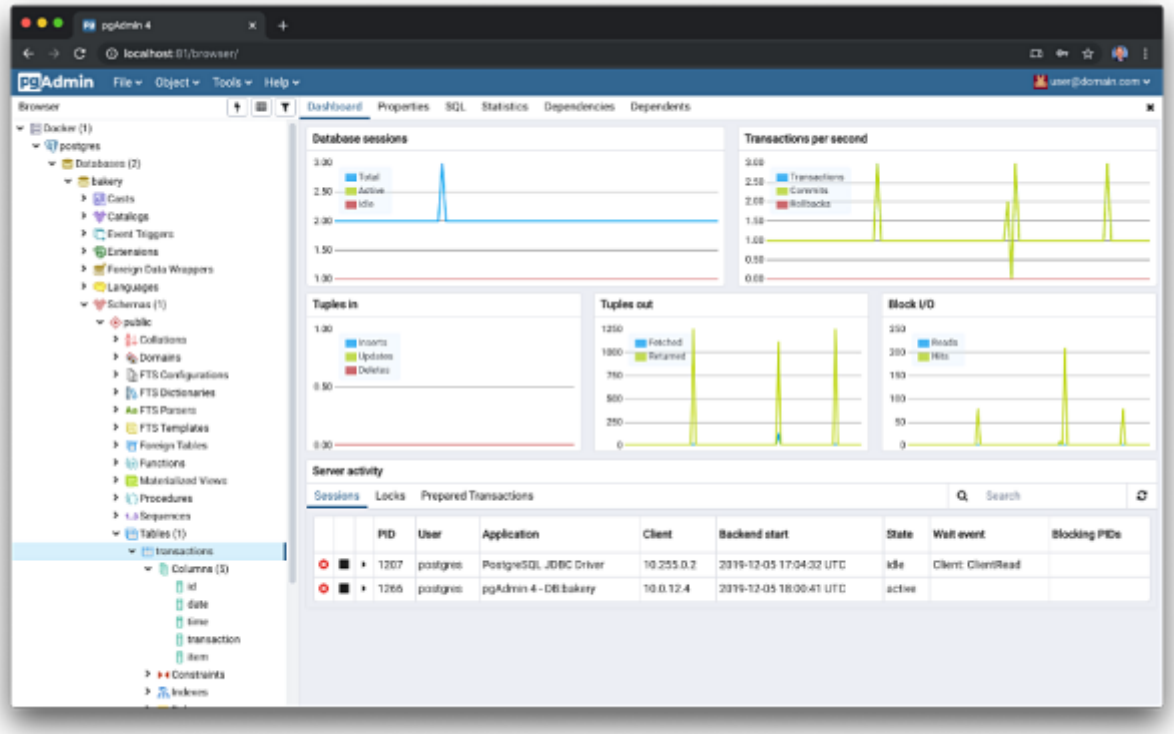
The table should contain 21,293 rows, each with 5 columns of data.

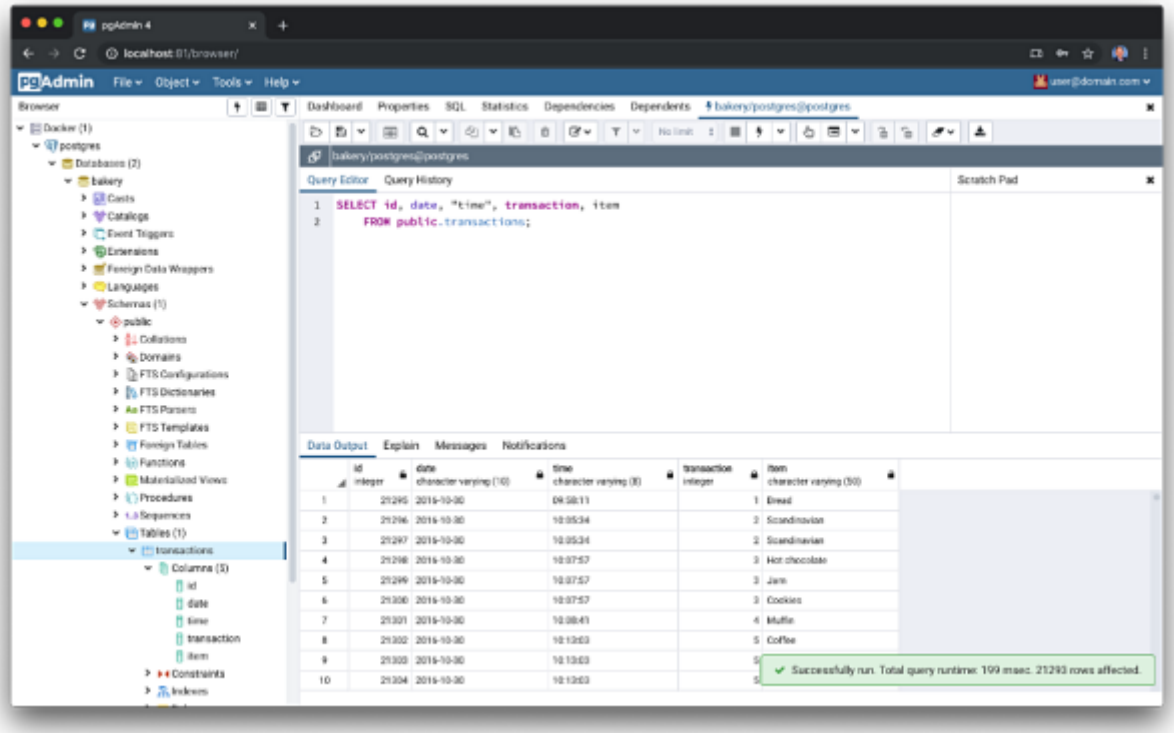


pgAdmin

Another excellent choice for interacting with PostgreSQL, in particular, is pgAdmin 4. It is my favorite tool for the administration of PostgreSQL. Although limited to PostgreSQL, the user interface and administrative capabilities of pgAdmin is superior to Adminer, in my opinion. For brevity, I chose not to include pgAdmin in this post. The Docker stack also contains a pgAdmin container, which has been commented out. To

use pgAdmin, just uncomment the container and re-run the Docker stack deploy command. pgAdmin should then be available on localhost port 81. The pgAdmin login credentials are in the Docker stack file.

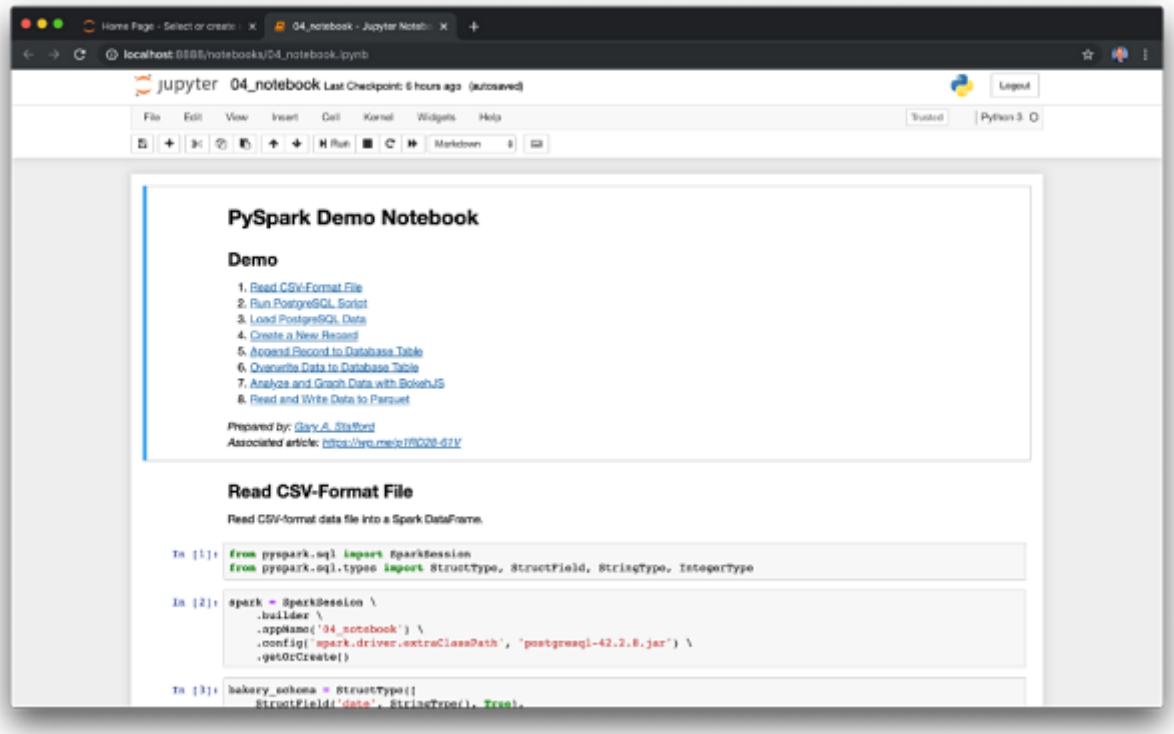




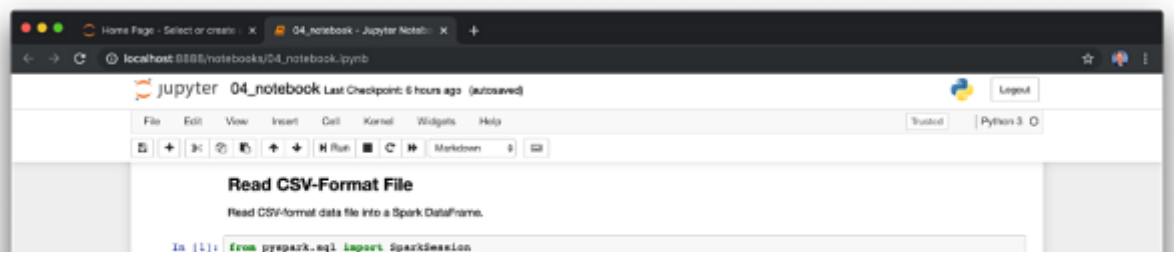
Developing Jupyter Notebooks

The real power of the Jupyter Docker Stacks containers is Jupyter Notebooks. According to the Jupyter Project, the notebook extends the console-based approach to interactive computing in a qualitatively new direction, providing a web-based application suitable for capturing the whole computation process, including developing, documenting, and executing code, as well as communicating the results. Notebook documents contain the inputs and outputs of an interactive session as well as additional text that accompanies the code but is not meant for execution.

To explore the capabilities of Jupyter notebooks, the project includes two simple Jupyter notebooks. The first notebooks, 04_notebook.ipynb, demonstrates typical PySpark functions, such as loading data from a CSV file and from the PostgreSQL database, performing basic data analysis with Spark SQL including the use of PySpark user-defined functions (UDF), graphing the data using BokehJS, and finally, saving data back to the database, as well as to the fast and efficient Apache Parquet file format. Below we see several notebook cells demonstrating these features.

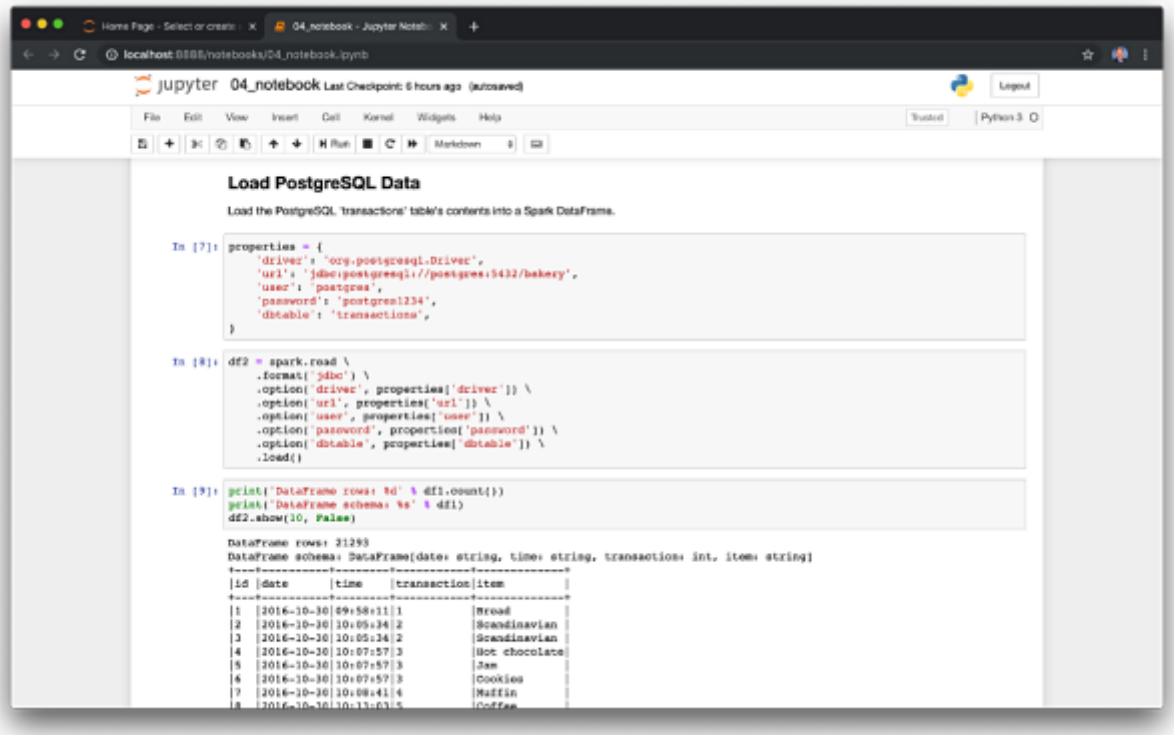


Markdown for Notebook Documentation



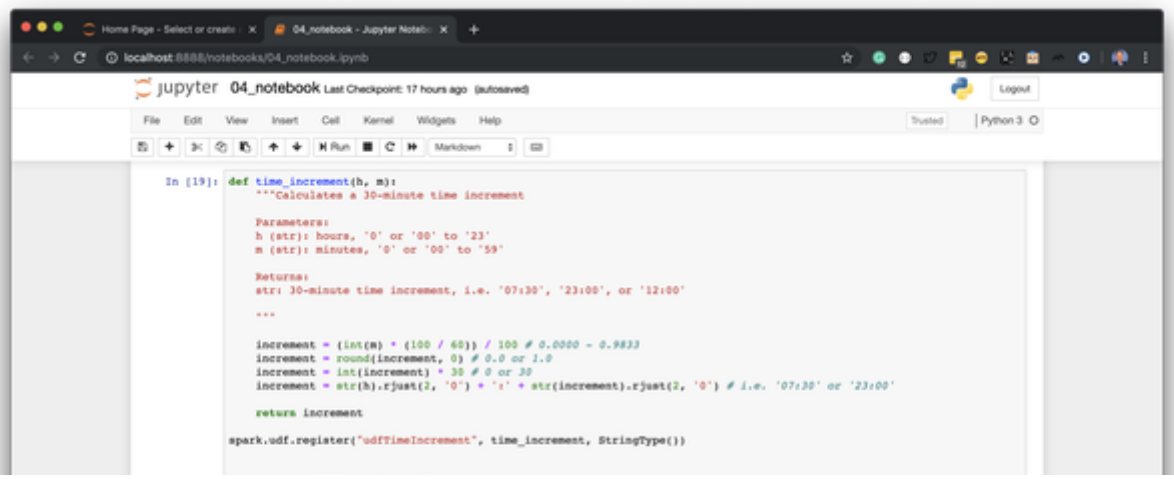
```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType

In [2]:
spark = SparkSession \
    .builder \
    .appName('04_notebook') \
    .config('spark.driver.extraClassPath', 'postgresql-42.2.8.jar') \
    .getOrCreate()

In [3]:
bakery_schema = StructType([
    StructField('date', StringType(), True),
    StructField('time', StringType(), True),
    StructField('transaction', IntegerType(), True),
    StructField('item', StringType(), True)
])

In [4]:
df1 = spark.read \
        .format('csv') \
        .option('header', 'true') \
        .load('BreadBasket_DMS.csv', schema=bakery_schema)

In [5]:
print('DataFrame rows: %d' % df1.count())
print('DataFrame schema: %s' % df1)
df1.show(10, False)

DataFrame rows: 21293
DataFrame schema: DataFrame[date: string, time: string, transaction: int, item: string]
+----------+--------+-----------+--------------+
|date      |time    |transaction|item          |
+----------+--------+-----------+--------------+
|2016-10-30|09:58:11|1          |Bread         |
|2016-10-30|10:05:34|2          |Scandinavian  |
|2016-10-30|10:05:34|2          |Scandinavian  |
```

Read CSV-Format Files into Spark DataFrame



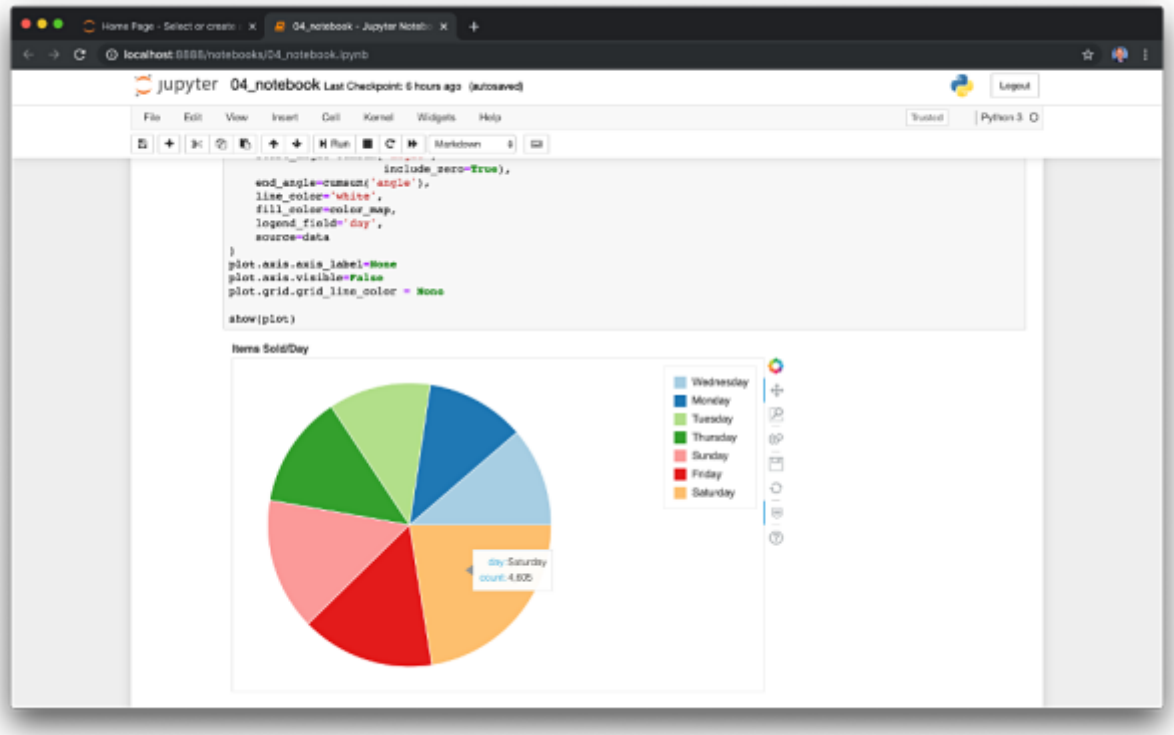
```
jupyter 04_notebook Last Checkpoint: 6 hours ago (autosaved)

Load PostgreSQL Data

Load the PostgreSQL 'transactions' table's contents into a Spark DataFrame.

In [7]:
properties = {
    'driver': 'org.postgresql.Driver',
    'url': 'jdbc:postgresql://postgres:5432/bakery',
    'user': 'postgres',
    'password': 'postgres1234',
    'dbtable': 'transactions',
}

In [8]:
df2 = spark.read \
    .format('jdbc') \
    .option('driver', properties['driver']) \
    .option('url', properties['url']) \
    .option('user', properties['user']) \
    .option('password', properties['password']) \
    .option('dbtable', properties['dbtable']) \
    .load()

In [9]:
print('DataFrame rows: %d' % df1.count())
print('DataFrame schema: %s' % df1)
df2.show(10, False)

DataFrame rows: 21293
DataFrame schema: DataFrame[date: string, time: string, transaction: int, item: string]
+---+----------+--------+-----------+--------------+
|id |date      |time    |transaction|item          |
+---+----------+--------+-----------+--------------+
|1  |2016-10-30|09:58:11|1          |Bread         |
|2  |2016-10-30|10:05:34|2          |Scandinavian  |
|3  |2016-10-30|10:05:34|2          |Scandinavian  |
|4  |2016-10-30|10:07:57|3          |Hot chocolate |
|5  |2016-10-30|10:07:57|3          |Jam           |
|6  |2016-10-30|10:07:57|3          |Cookies       |
|7  |2016-10-30|10:08:41|4          |Muffin        |
|8  |2016-10-30|10:13:03|5          |Coffee        |
```

Load Data from PostgreSQL into Spark DataFrame



```
jupyter 04_notebook Last Checkpoint: 17 hours ago (autosaved)

In [19]:
def time_increment(h, m):
    """Calculates a 30-minute time increment

    Parameters:
    h (str): hours, '0' or '00' to '23'
    m (str): minutes, '0' or '00' to '59'

    Returns:
    str: 30-minute time increment, i.e. '07:30', '23:00', or '12:00'

    """

    increment = (int(m) * (100 / 60)) / 100 # 0.0000 - 0.9833
    increment = round(increment, 0) # 0.0 or 1.0
    increment = int(increment) * 30 # 0 or 30
    increment = str(h).rjust(2, '0') + ':' + str(increment).rjust(2, '0') # i.e. '07:30' or '23:00'

    return increment

spark.udf.register("udfTimeIncrement", time_increment, StringType())
```

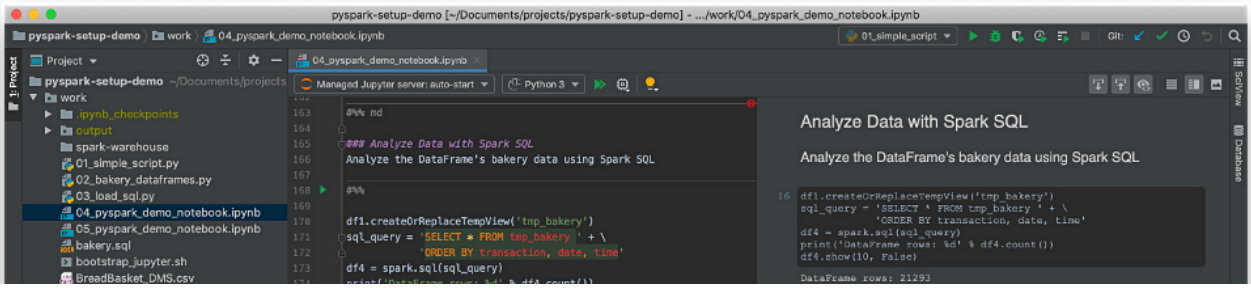
Perform Spark SQL Query including use of UDF

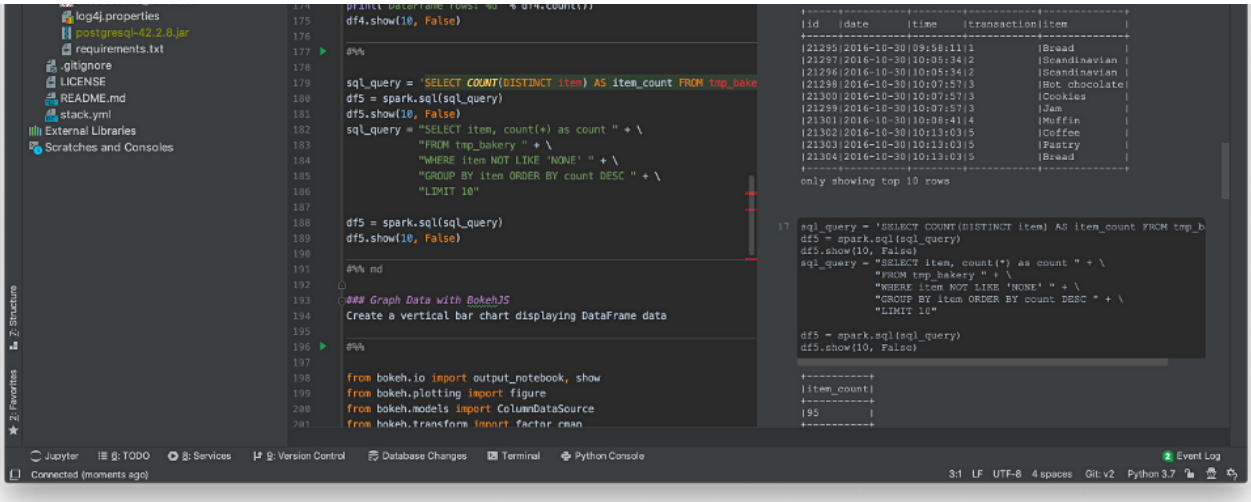


Plot Spark SQL Query Results using BokehJS

IDE Integration

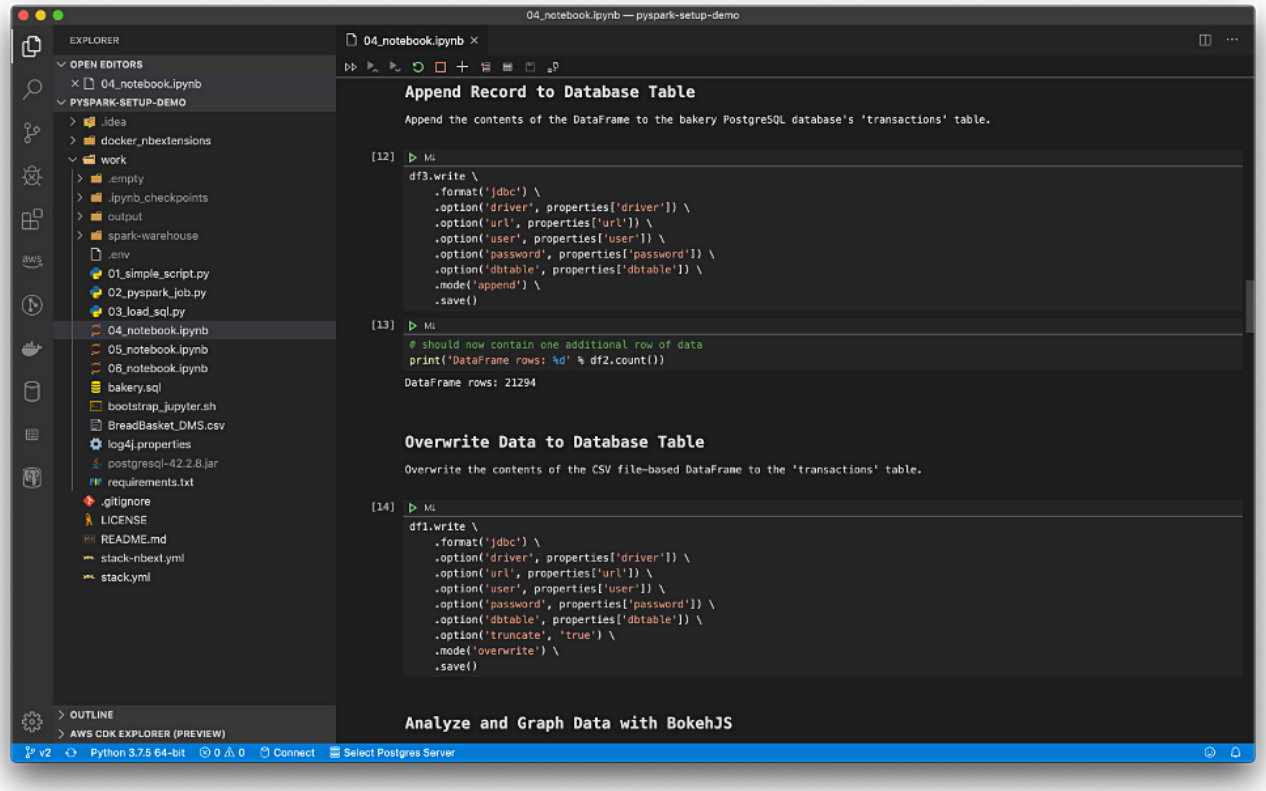
Recall, the working directory, containing the GitHub source code for the project, is bind-mounted to the Jupyter container. Therefore, you can also edit any of the files, including notebooks, in your favorite IDE, such as JetBrains PyCharm and Microsoft Visual Studio Code. PyCharm has built-in language support for Jupyter Notebooks, as shown below.





PyCharm 2019.2.5 (Professional Edition)

As does Visual Studio Code using the Python extension.



Visual Studio Code Version: 1.40.2

Using Additional Packages

As mentioned in the Introduction, the Jupyter Docker Stacks come ready-to-run, with a wide variety of Python packages to extend their functionality. To demonstrate the use of these packages, the project contains a second Jupyter notebook document, 05_notebook.ipynb. This notebook uses SciPy, the well-known Python package for

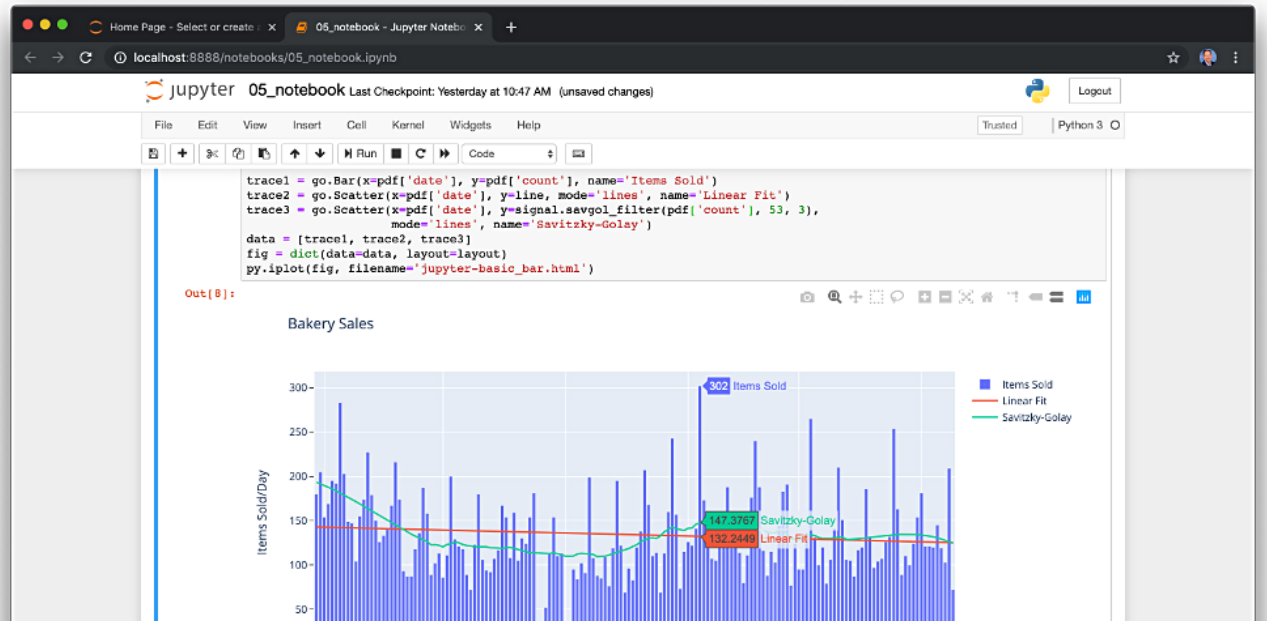
mathematics, science, and engineering, NumPy, the well-known Python package for scientific computing, and the Plotly Python Graphing Library. While NumPy and SciPy are included on the Jupyter Docker Image, the bootstrap script uses pip to install the required Plotly packages. Similar to Bokeh, shown in the previous notebook, we can use these libraries to create richly interactive data visualizations.

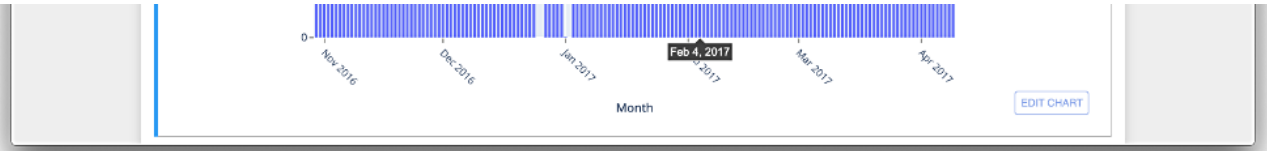
Plotly

To use Plotly from within the notebook, you will first need to sign up for a free account and obtain a username and API key. To ensure we do not accidentally save sensitive Plotly credentials in the notebook, we are using python-dotenv. This Python package reads key/value pairs from a `.env` file, making them available as environment variables to our notebook. Modify and run the following two commands from a Jupyter terminal to create the `.env` file and set you Plotly username and API key credentials. Note that the `.env` file is part of the `.gitignore` file and will not be committed to back to git, potentially compromising the credentials.

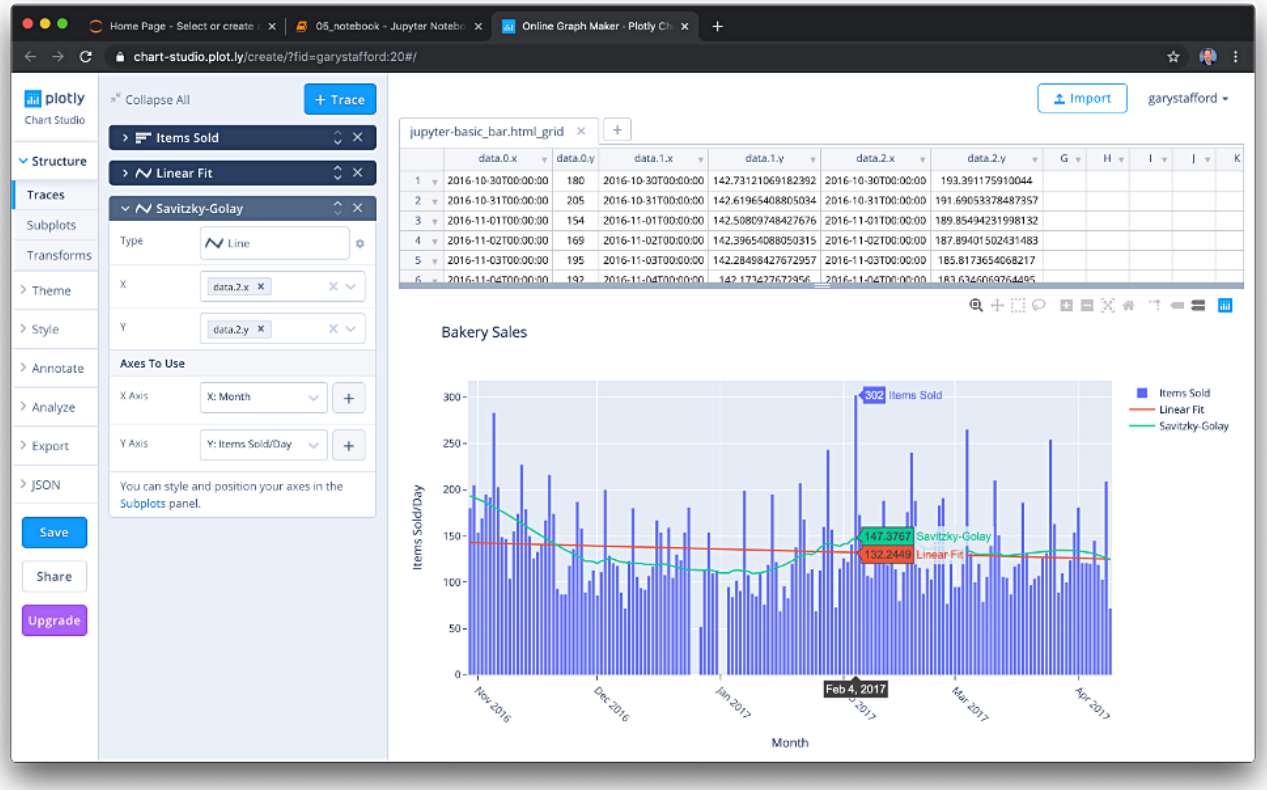
```
echo "PLOTLY_USERNAME=your-username" >> .env
echo "PLOTLY_API_KEY=your-api-key" >> .env
```

Shown below, we use Plotly to construct a bar chart of daily bakery items sold. The chart uses SciPy and NumPy to construct a linear fit (regression) and plot a line of best fit for the bakery data and overlaying the vertical bars. The chart also uses SciPy's Savitzky-Golay Filter to plot the second line, illustrating a smoothing of our bakery data.





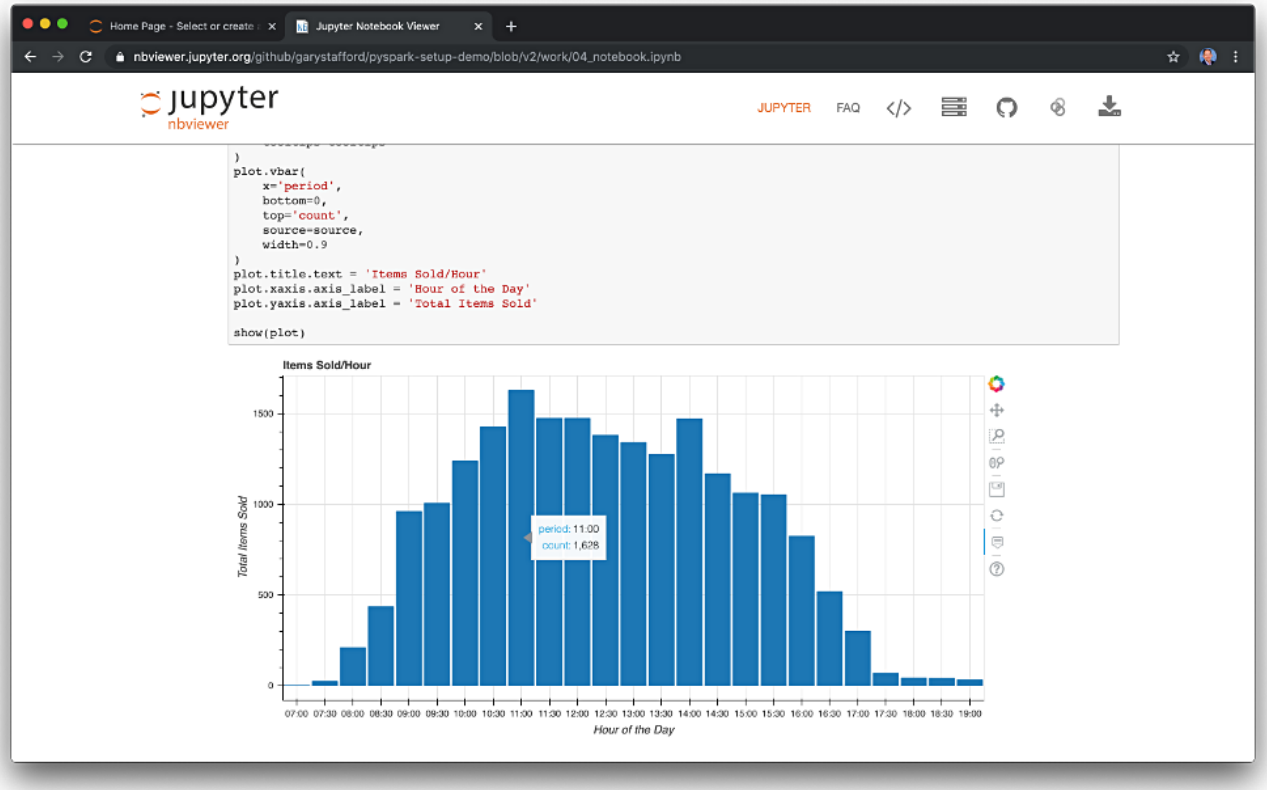
Plotly also provides Chart Studio Online Chart Maker. Plotly describes Chart Studio as the world's most modern enterprise data visualization solutions. We can enhance, stylize, and share our data visualizations using the free version of Chart Studio Cloud.



Jupyter Notebook Viewer

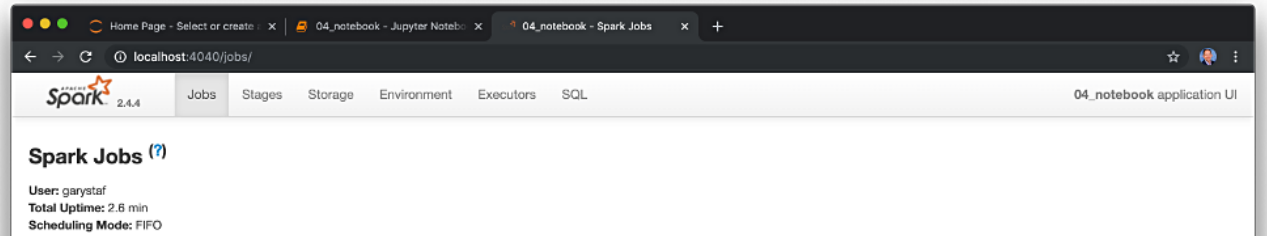
Notebooks can also be viewed using nbviewer, an open-source project under Project Jupyter. Thanks to Rackspace hosting, the nbviewer instance is a free service.

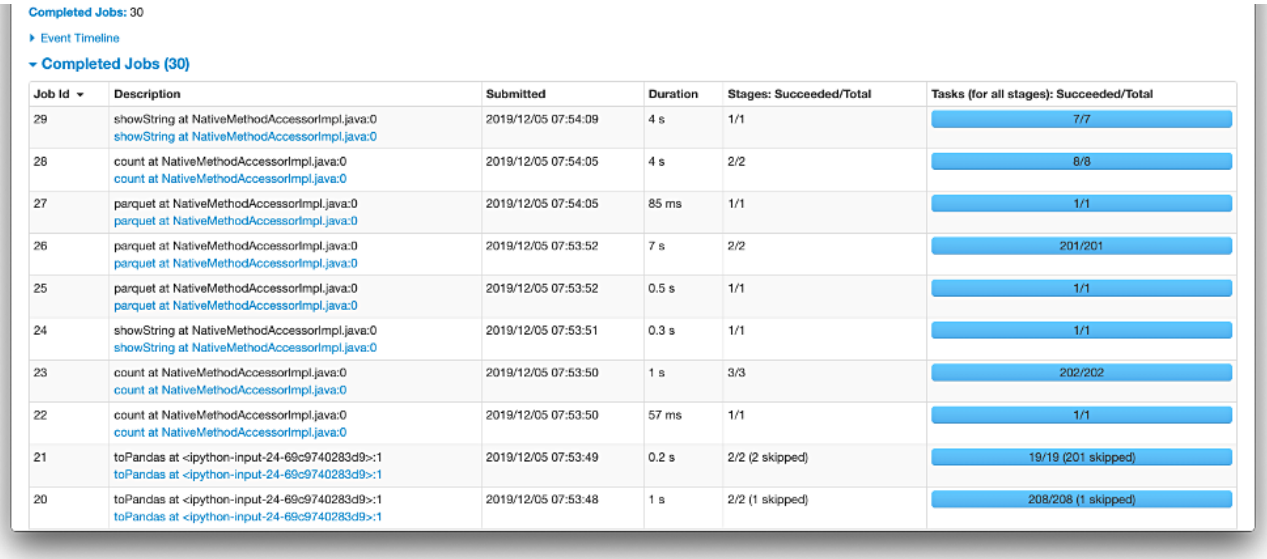
Using nbviewer, below, we see the output of a cell within the 04_notebook.ipynb notebook. View this notebook, here, using nbviewer.



Monitoring Spark Jobs

The Jupyter Docker container exposes Spark's monitoring and instrumentation web user interface. We can review each completed Spark Job in detail.





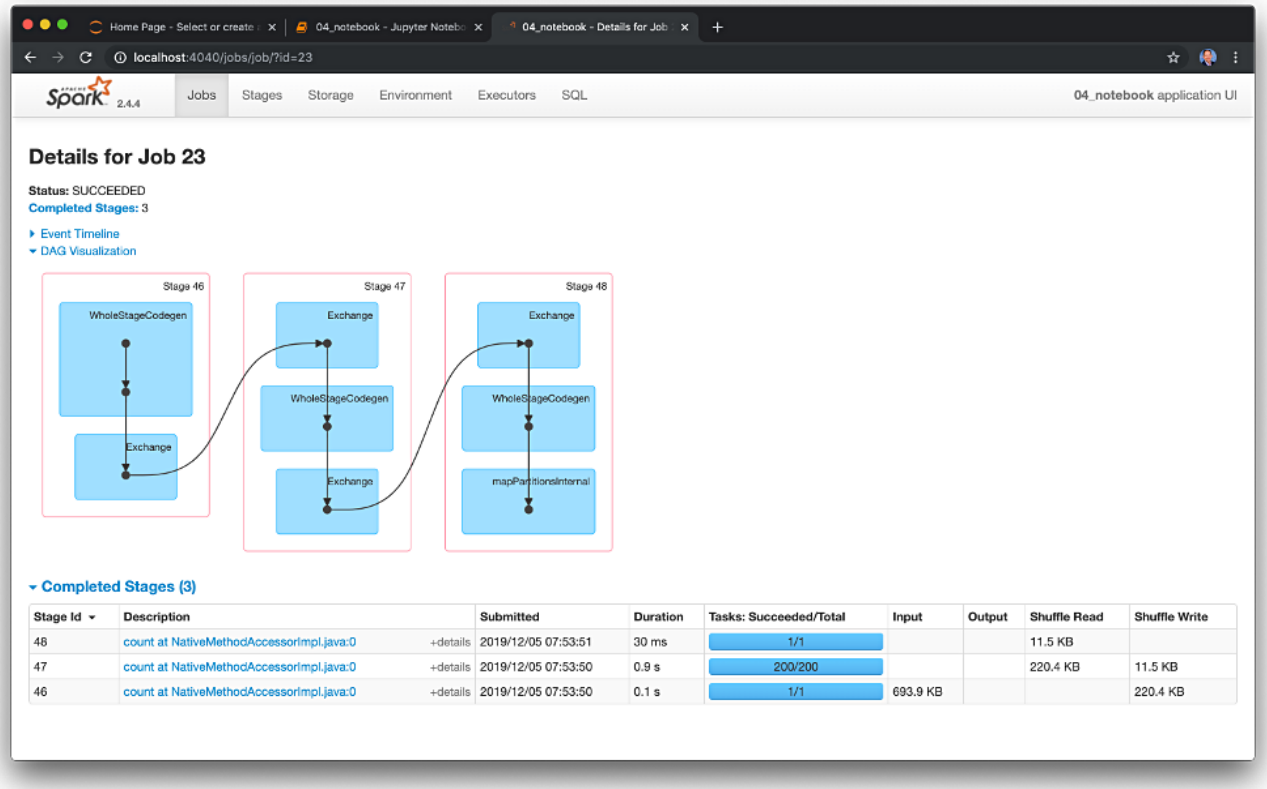
Completed Jobs: 30

Event Timeline

Completed Jobs (30)

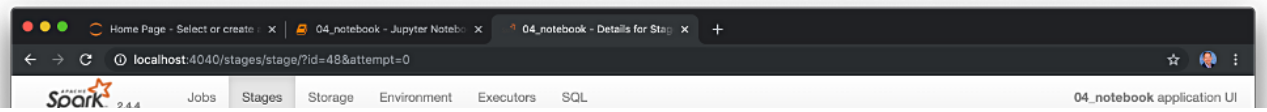
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
29	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2019/12/05 07:54:09	4 s	1/1	7/7
28	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2019/12/05 07:54:05	4 s	2/2	8/8
27	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2019/12/05 07:54:05	85 ms	1/1	1/1
26	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2019/12/05 07:53:52	7 s	2/2	201/201
25	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2019/12/05 07:53:52	0.5 s	1/1	1/1
24	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2019/12/05 07:53:51	0.3 s	1/1	1/1
23	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2019/12/05 07:53:50	1 s	3/3	202/202
22	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2019/12/05 07:53:50	57 ms	1/1	1/1
21	toPandas at <ipython-input-24-69c9740283d9>:1 toPandas at <ipython-input-24-69c9740283d9>:1	2019/12/05 07:53:49	0.2 s	2/2 (2 skipped)	19/19 (201 skipped)
20	toPandas at <ipython-input-24-69c9740283d9>:1 toPandas at <ipython-input-24-69c9740283d9>:1	2019/12/05 07:53:48	1 s	2/2 (1 skipped)	208/208 (1 skipped)

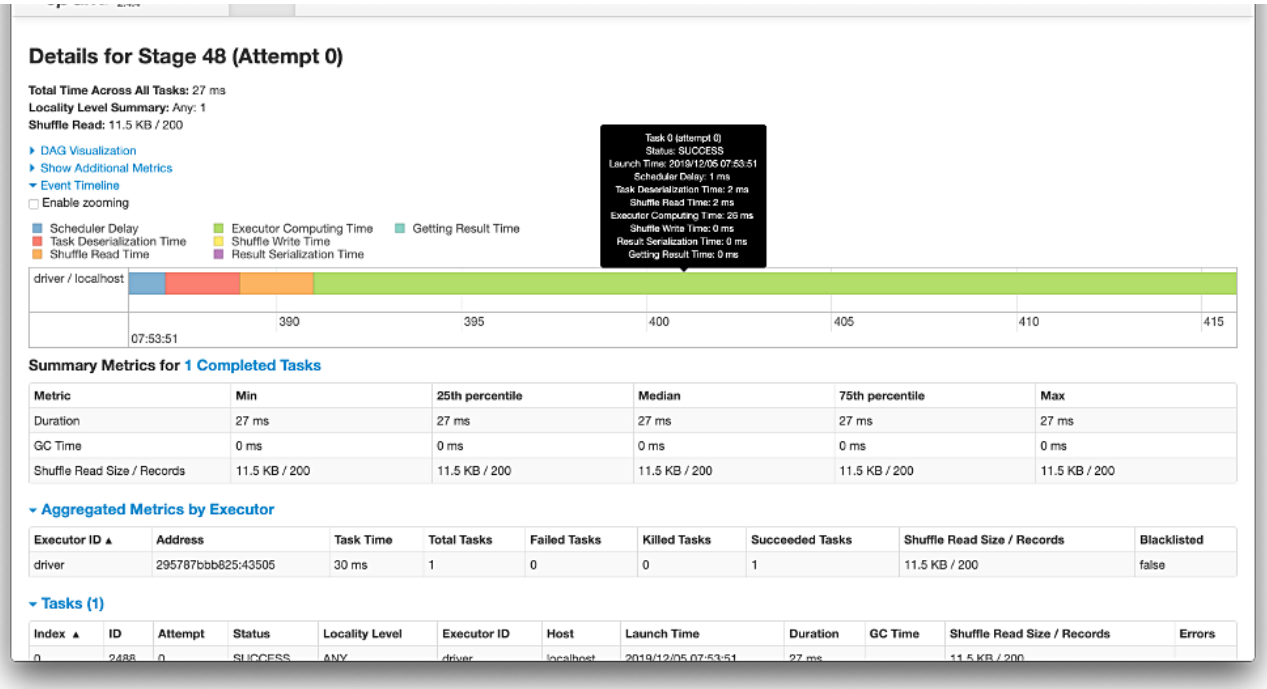
We can review details of each stage of the Spark job, including a visualization of the DAG (Directed Acyclic Graph), which Spark constructs as part of the job execution plan, using the DAG Scheduler.



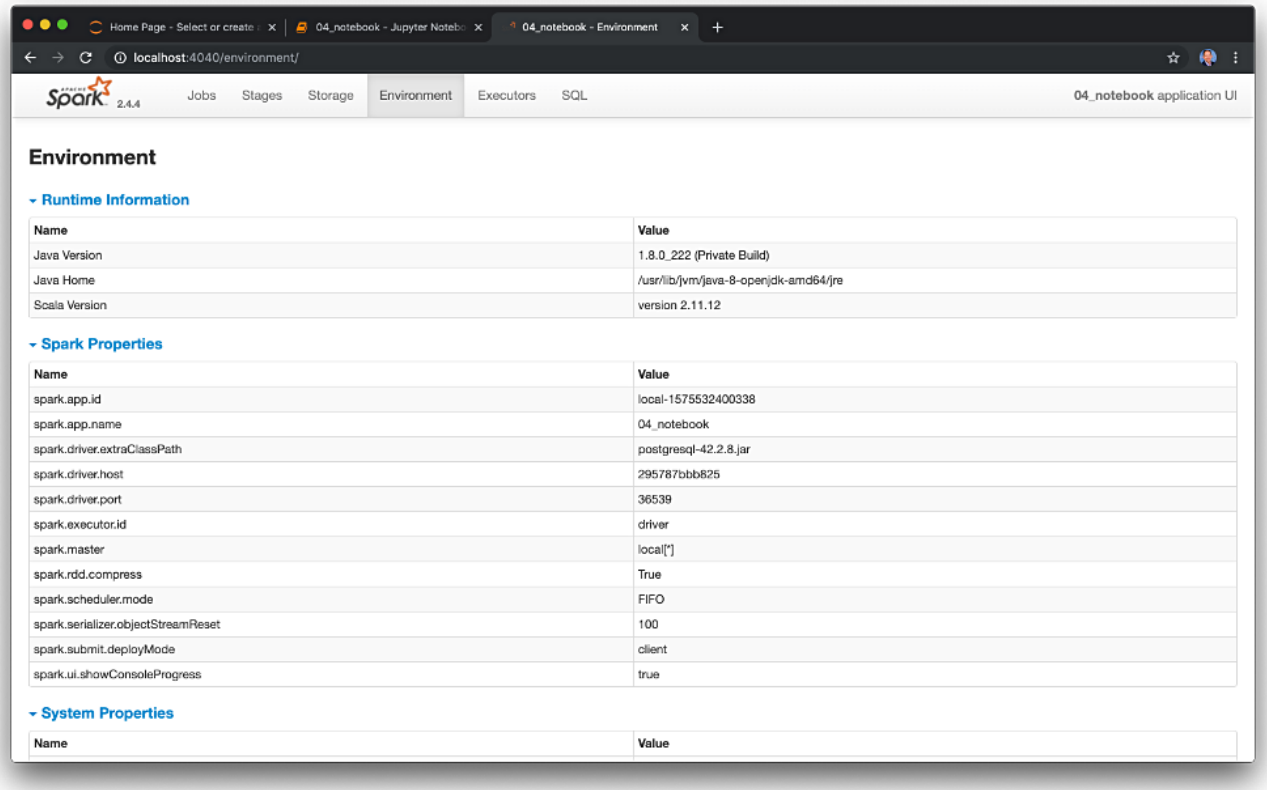
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
48	count at NativeMethodAccessorImpl.java:0 +details	2019/12/05 07:53:51	30 ms	1/1			11.5 KB	
47	count at NativeMethodAccessorImpl.java:0 +details	2019/12/05 07:53:50	0.9 s	200/200			220.4 KB	11.5 KB
46	count at NativeMethodAccessorImpl.java:0 +details	2019/12/05 07:53:50	0.1 s	1/1	693.9 KB			220.4 KB

We can also review the task composition and timing of each event occurring as part of the stages of the Spark job.



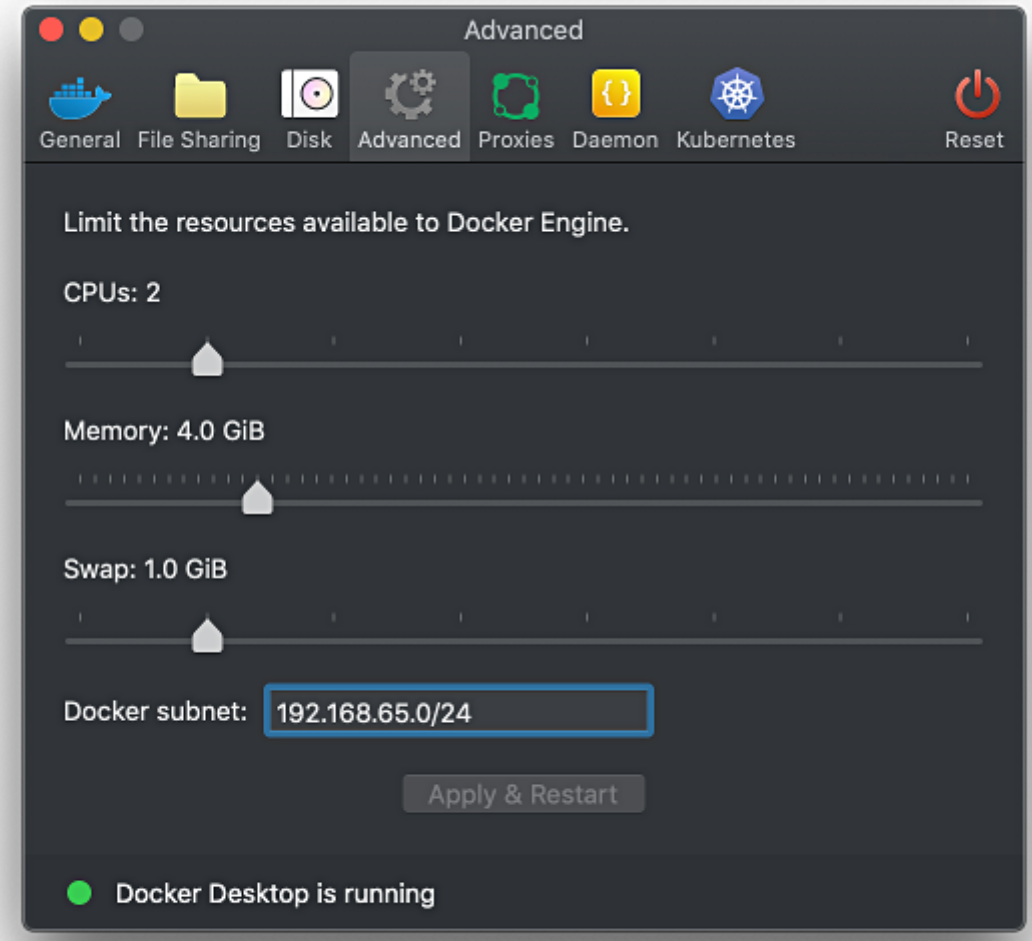


We can also use the Spark interface to review and confirm the runtime environment configuration, including versions of Java, Scala, and Spark, as well as packages available on the Java classpath.



Local Spark Performance

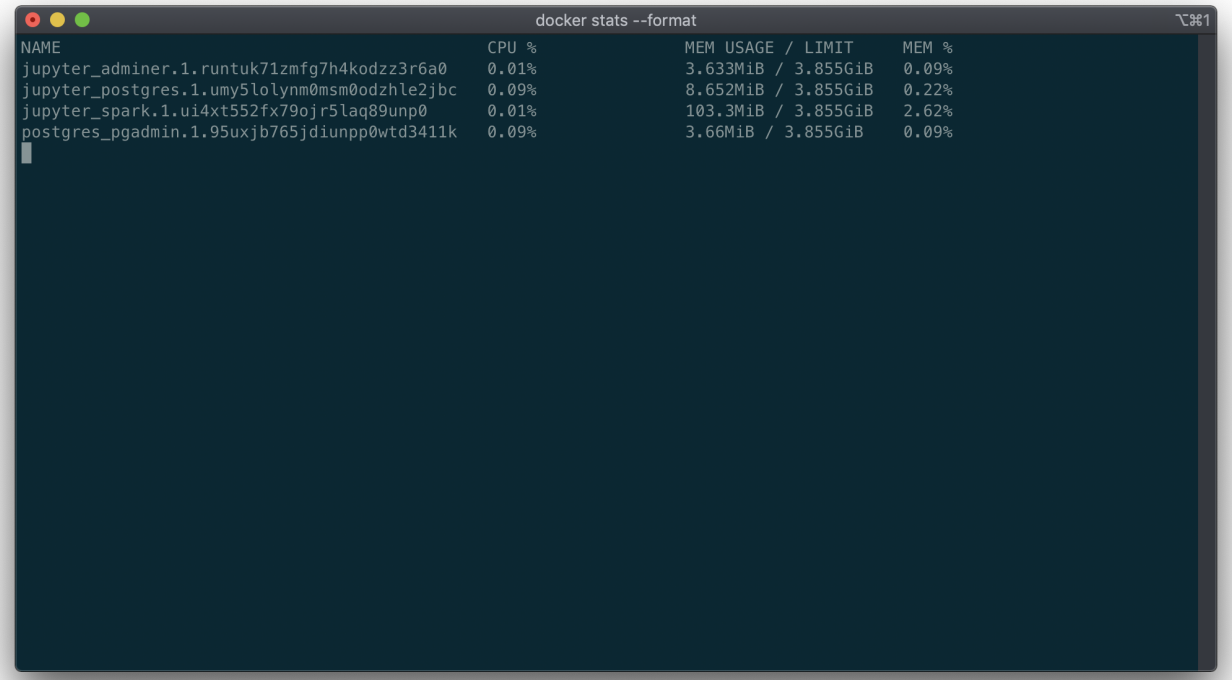
Running Spark on a single node within the Jupyter Docker container on your local development system is not a substitute for a true Spark cluster, Production-grade, multi-node Spark clusters running on bare metal or robust virtualized hardware, and managed with Hadoop YARN, Apache Mesos, or Kubernetes. In my opinion, you should only adjust your Docker resources limits to support an acceptable level of Spark performance for running small exploratory workloads. You will not realistically replace the need to process big data and execute jobs requiring complex calculations on a Production-grade, multi-node Spark cluster.



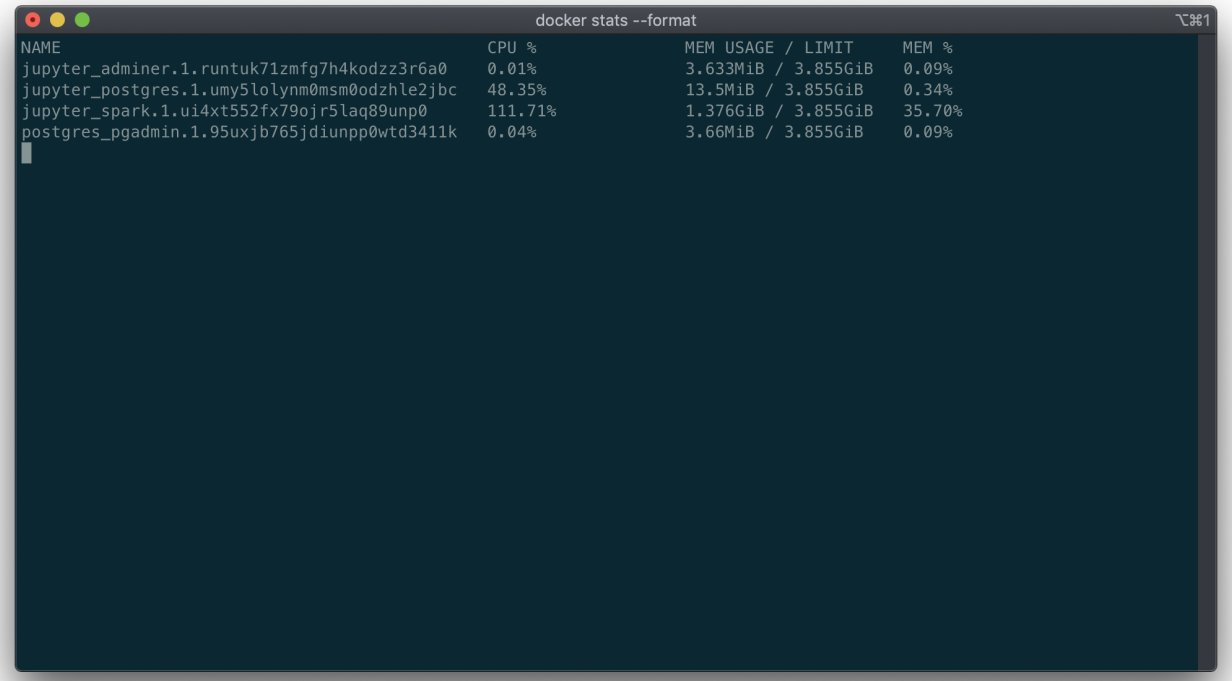
We can use the following docker stats command to examine the container's CPU and memory metrics.

```
docker stats --format "table
{{.Name}}\t{{.CPUPerc}}\t{{.MemUsage}}\t{{.MemPerc}}"
```

Below, we see the stats from the Docker stack's three containers showing little or no activity.



Compare those stats with the ones shown below, recorded while a notebook was reading and writing data, and executing Spark SQL queries. The CPU and memory output show spikes, but both appear to be within acceptable ranges.

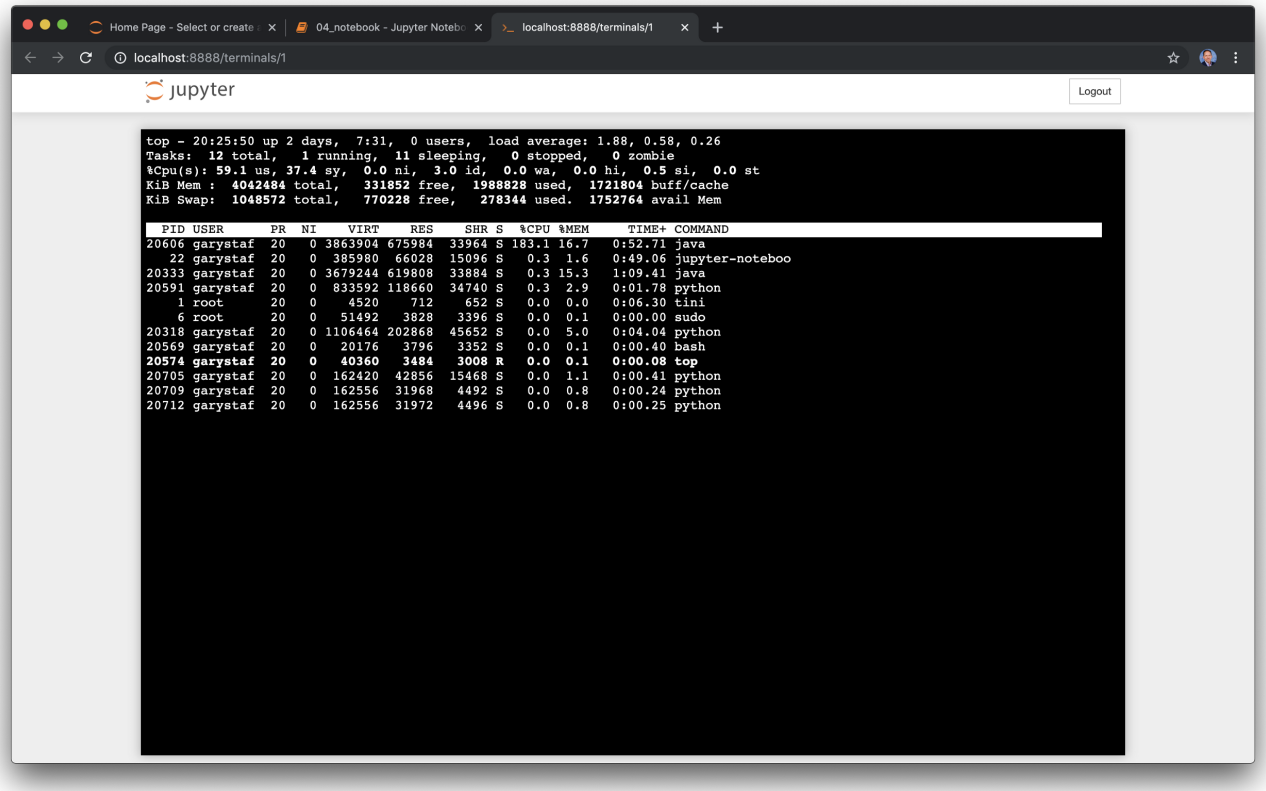


Linux Process Monitors

Another option to examine container performance metrics is top, which is pre-installed in our Jupyter container. For example, execute the following `top` command from a Jupyter terminal, sorting processes by CPU usage.

```
top -o %CPU
```

We should observe the individual performance of each process running in the Jupyter container.

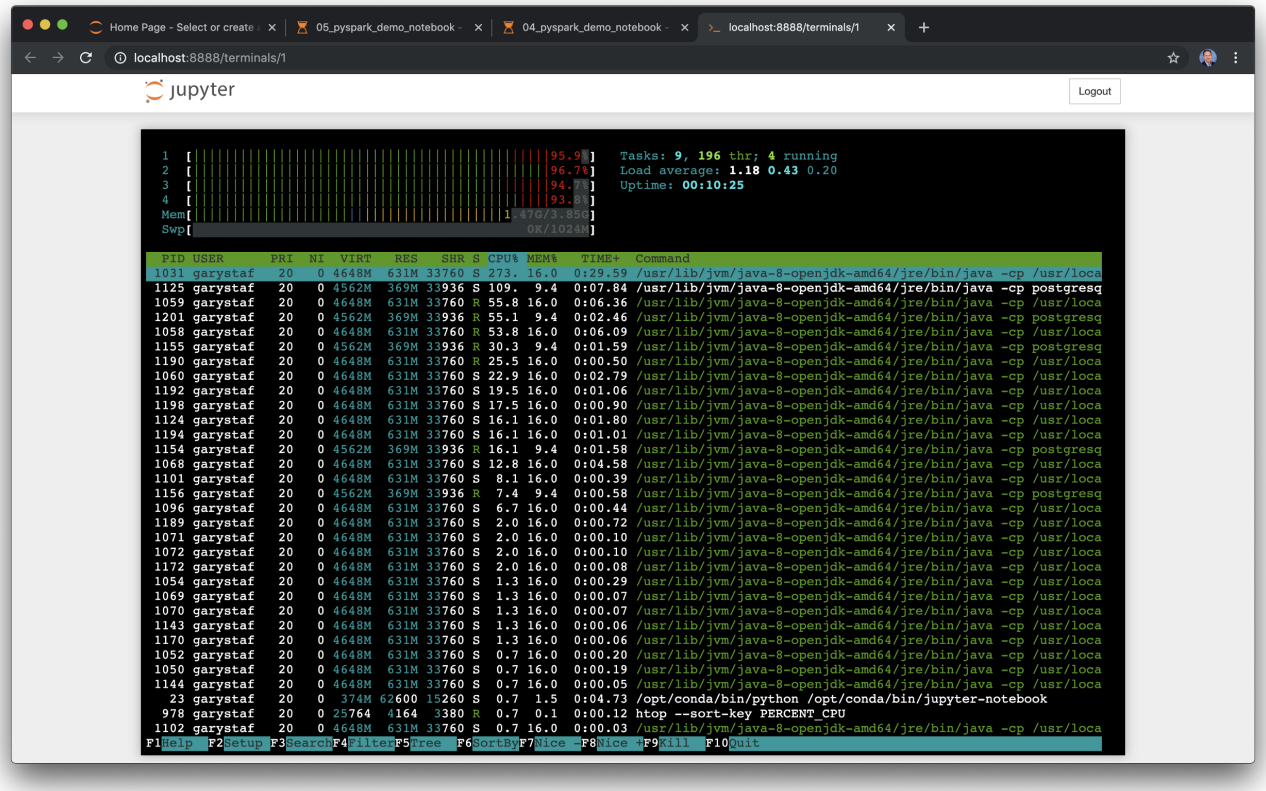


A step up from `top` is htop, an interactive process viewer for Unix. It was installed in the container by the bootstrap script. For example, we can execute the `htop` command from a Jupyter terminal, sorting processes by CPU % usage.

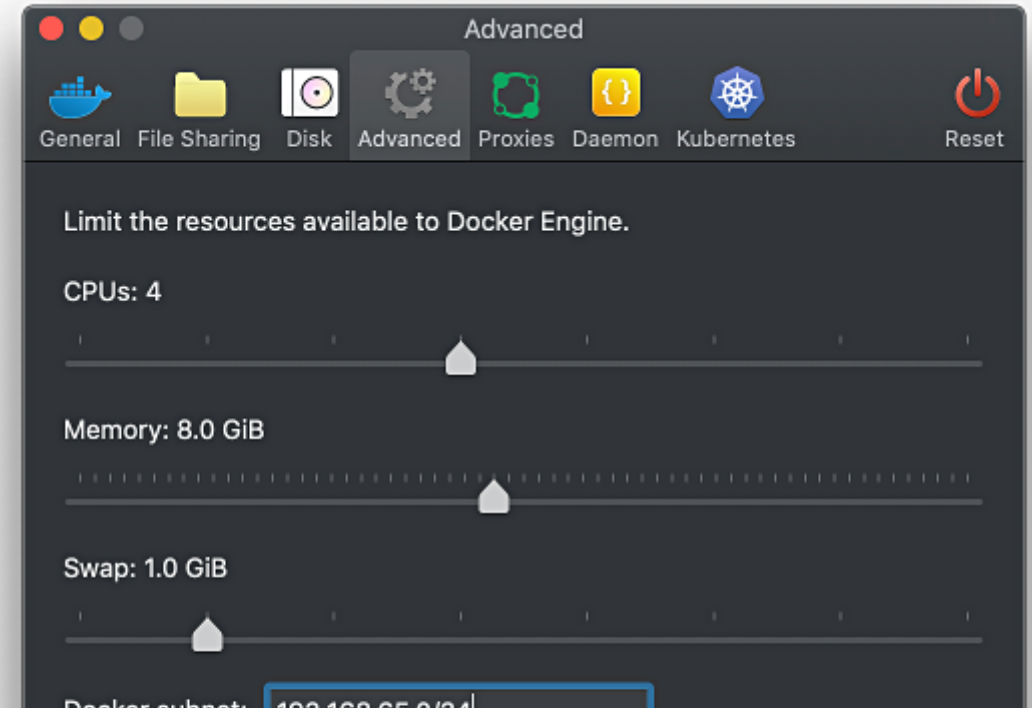
```
htop --sort-key PERCENT_CPU
```

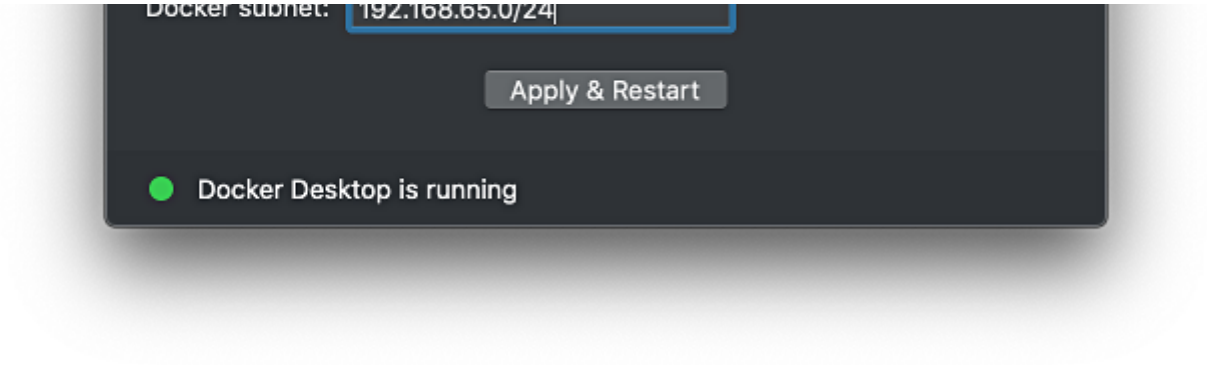
With `htop`, observe the individual CPU activity. Here, the four CPUs at the top left of the `htop` window are the CPUs assigned to Docker. We get insight into the way Spark is

using multiple CPUs, as well as other performance metrics, such as memory and swap.



Assuming your development machine is robust, it is easy to allocate and deallocate additional compute resources to Docker if required. Be careful not to allocate excessive resources to Docker, starving your host machine of available compute resources for other applications.





Notebook Extensions

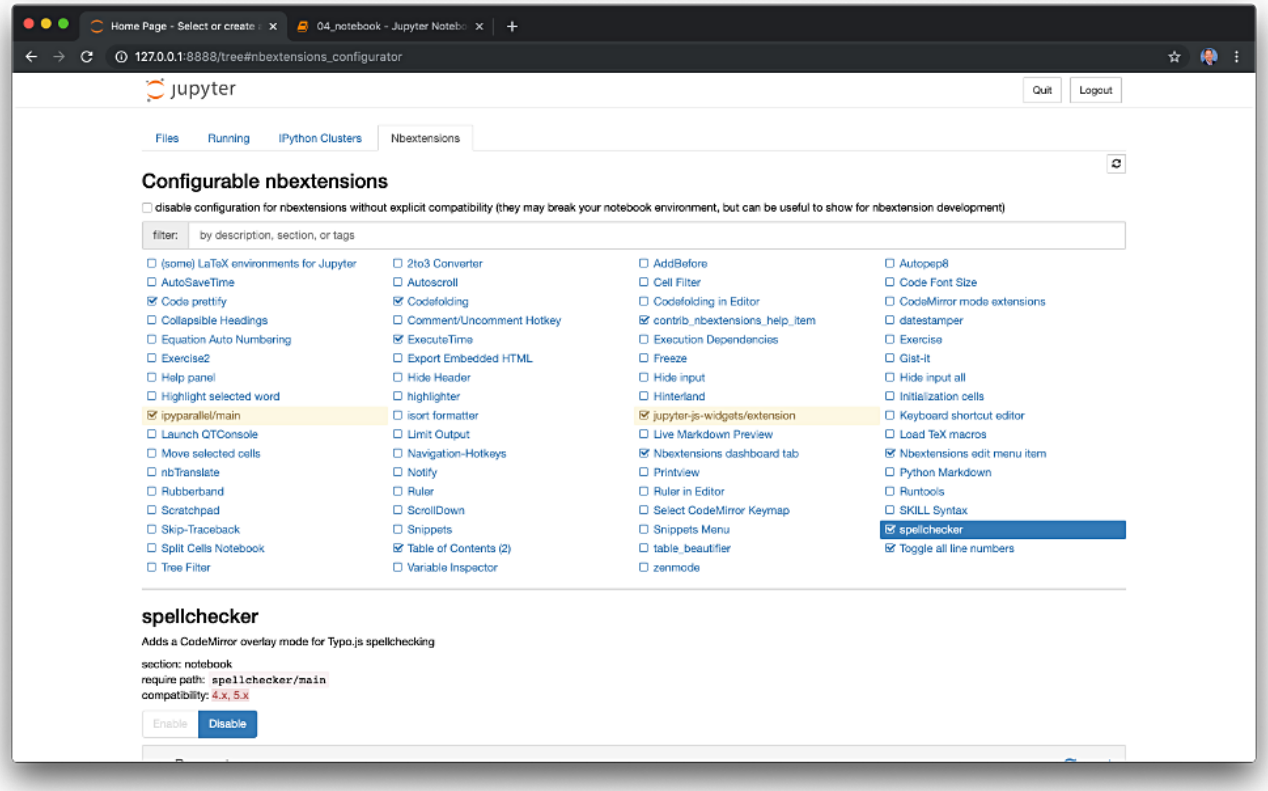
There are many ways to extend the Jupyter Docker Stacks. A popular option is jupyter-contrib-nbextensions. According to their website, the `jupyter-contrib-nbextensions` package contains a collection of community-contributed unofficial extensions that add functionality to the Jupyter notebook. These extensions are mostly written in JavaScript and will be loaded locally in your browser. Installed notebook extensions can be enabled, either by using built-in Jupyter commands, or more conveniently by using the jupyter_nbextensions_configurator server extension.

The project contains an alternate Docker stack file, stack-nbext.yml. The stack uses an alternative Docker image, `garystafford/all-spark-notebook-nbext:latest` , This Dockerfile, which builds it, uses the `jupyter/all-spark-notebook:latest` image as a base image. The alternate image adds in the `jupyter-contrib-nbextensions` and `jupyter_nbextensions_configurator` packages. Since Jupyter would need to be restarted after `nbextensions` is deployed, it cannot be done from within a running `jupyter/all-spark-notebook` container.

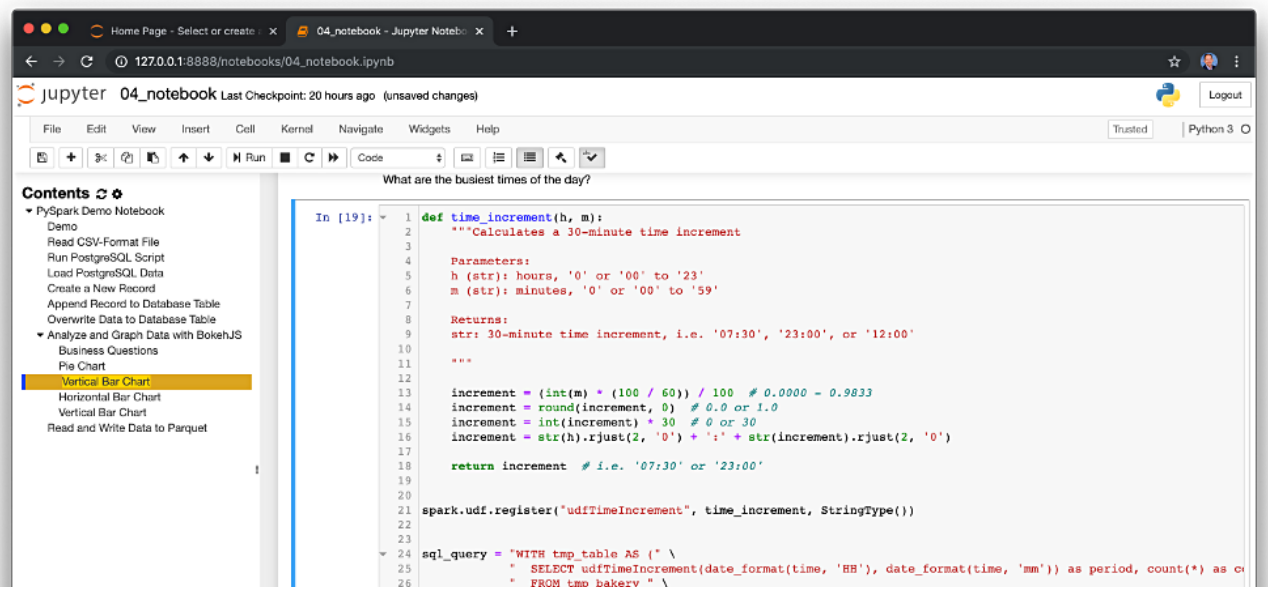
```
1  FROM jupyter/all-spark-notebook:latest
2
3  USER root
4
5  RUN pip install jupyter_contrib_nbextensions \
6        && jupyter contrib nbextension install --system \
7        && pip install jupyter_nbextensions_configurator \
8        && jupyter nbextensions_configurator enable --system \
9        && pip install yapf # for code pretty
10
11 USER $NB_UID
```

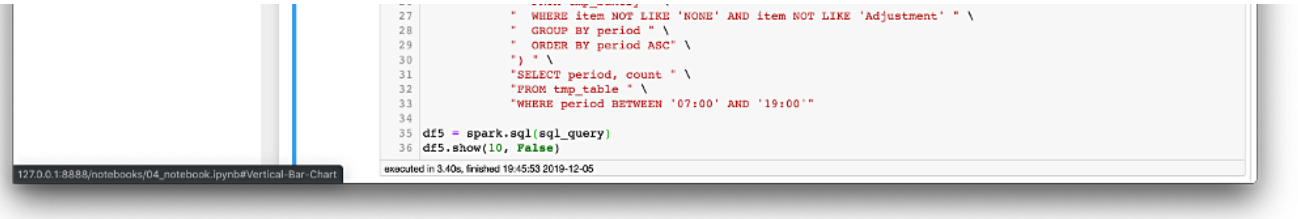
Dockerfile hosted with ♡ by GitHub view raw

Using this alternate stack, below in our Jupyter container, we see the sizable list of extensions available. Some of my favorite extensions include ‘spellchecker’, ‘Codefolding’, ‘Code pretty’, ‘ExecutionTime’, ‘Table of Contents’, and ‘Toggle all line numbers’.



Below, we see five new extension icons that have been added to the menu bar of 04_notebook.ipynb. You can also observe the extensions have been applied to the notebook, including the table of contents, code-folding, execution time, and line numbering. The spellchecking and pretty code extensions were both also applied.





Conclusion

In this brief post, we have seen how easy it is to get started learning and performing data analytics using Jupyter notebooks, Python, Spark, and PySpark, all thanks to the Jupyter Docker Stacks. We could use this same stack to learn and perform machine learning using Scala and R. Extending the stack's capabilities is as simple as swapping out this Jupyter image for another, with a different set of tools, as well as adding additional containers to the stack, such as MySQL, MongoDB, RabbitMQ, Apache Kafka, and Apache Cassandra.

. . .

All opinions expressed in this post are my own and not necessarily the views of my current or past employers, their clients.

Docker Spark Jupyter Notebook Pyspark Big Data