

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



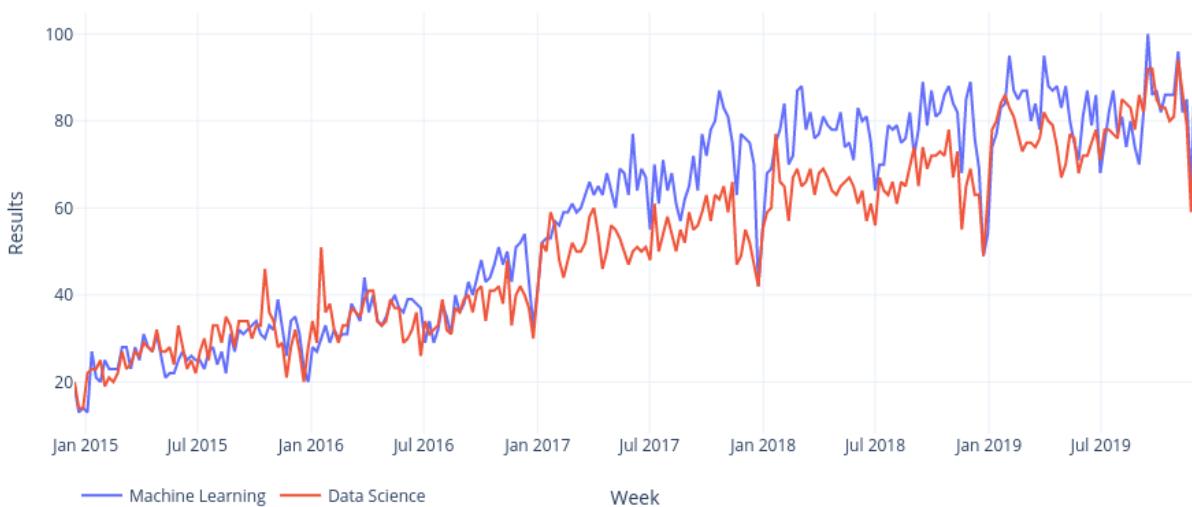
Gary A. Stafford

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.

Machine Learning and Data Science Search Results: 5-Year Trend



(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 *Julia*, *Python*, 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.



## 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.

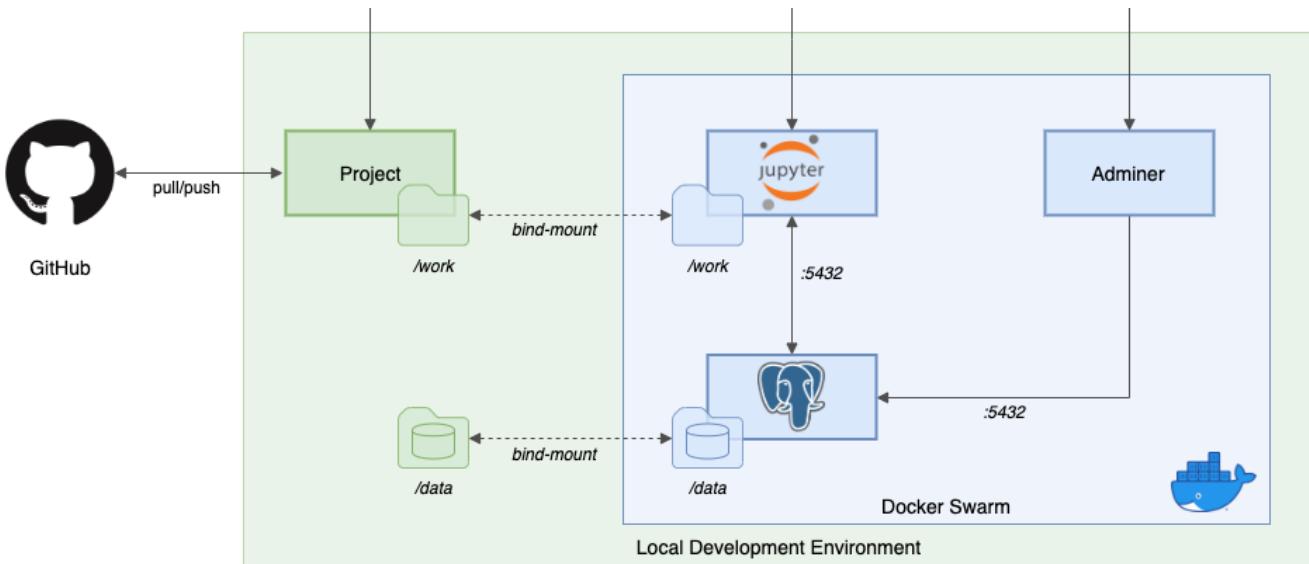
## Demonstration

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
39   networks:
40     - demo-net
41
42   volumes:
43     - $HOME/data/postgres:/var/lib/postgresql/data
44
45   deploy:
46     restart_policy:
47       condition: on-failure
48
49 adminer:
50   image: adminer:latest
51   ports:
52     - "8080:8080/tcp"
53   networks:
54     - demo-net
55
56   deploy:
57     restart_policy:
58       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 screenshot shows a terminal window with the following output:

```

garystaf@e483e767cbac: ~/Documents/projects/pyspark-setup-demo
$ docker stack deploy -c stack.yml jupyter
Creating network jupyter_demo-net
Creating service jupyter_spark
Creating service jupyter_postgres
Creating service jupyter_adminer
$ docker stack ps jupyter
ID          NAME      IMAGE           NODE      DESIRED STATE   CURRENT STATE
runtuk71zmfg  jupyter_adminer.1  adminer:latest  docker-desktop  Running
umyStLynn@m  jupyter_postgres.1  postgres:12-alpine  docker-desktop  Running
ui4xt552fx79  jupyter_spark.1   jupyter/all-spark-notebook:latest  docker-desktop  Running
$ prod * < 10:34:12
$ prod * < 10:34:15

```

The terminal shows the creation of the stack and its services, followed by a listing of the stack's services with their current state.

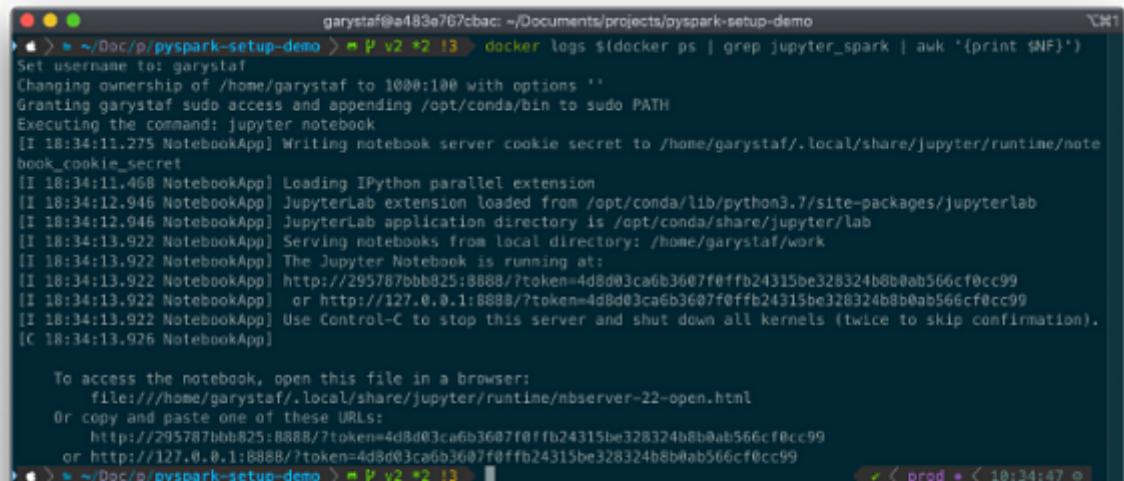
The `jupyter/all-spark-notebook` Docker image is large, approximately 5 GB. Depending on your Internet connection, if this is the first time you have pulled this image, the stack may take several minutes to enter a running state. Although not required, I usually pull Docker images in advance.

```
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.

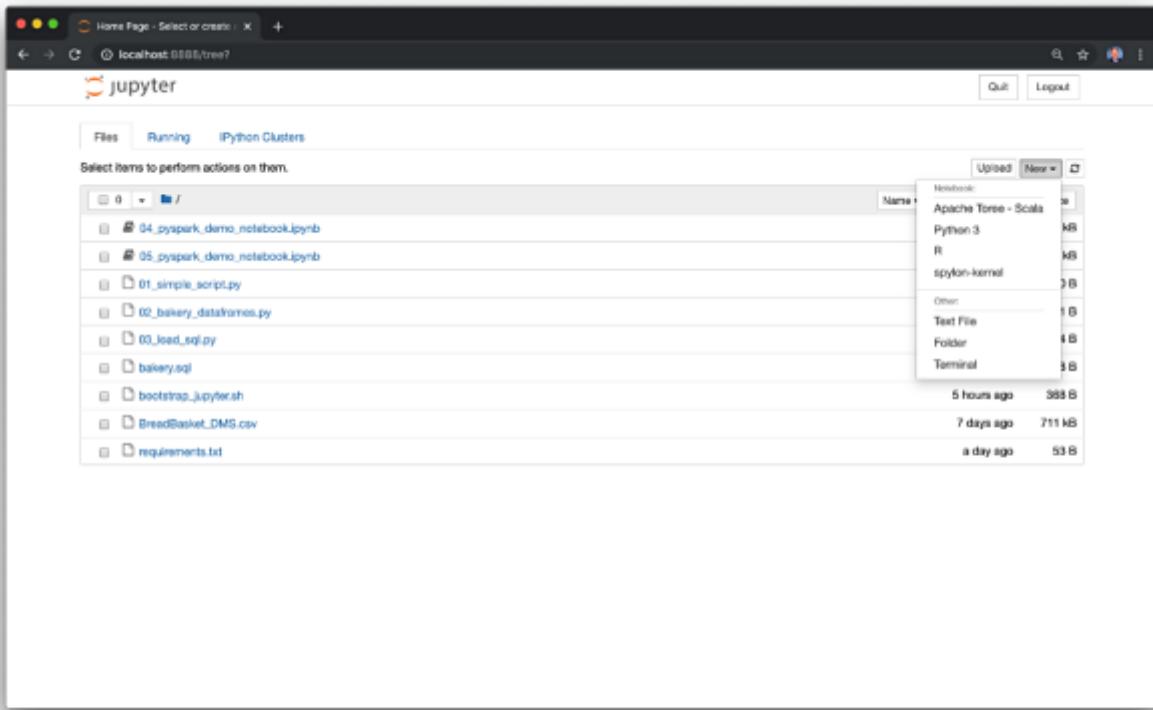


The screenshot shows a terminal window with the following text output:

```
garystaf@e483e767cbac: ~/Documents/projects/pyspark-setup-demo
$ docker logs $(docker ps | grep jupyter_spark | awk '{print $NF}')
Set username to: garystaf
Changing ownership of /home/garystaf to 1000:100 with options ''
Granting garystaf sudo access and appending /opt/conda/bin to sudo PATH
Executing the command: jupyter notebook
[I 18:34:11.275 NotebookApp] Writing notebook server cookie secret to /home/garystaf/.local/share/jupyter/runtime/notebook_cookie_secret
[I 18:34:11.468 NotebookApp] Loading IPython parallel extension
[I 18:34:12.946 NotebookApp] JupyterLab extension loaded from /opt/conda/lib/python3.7/site-packages/jupyterlab
[I 18:34:12.946 NotebookApp] JupyterLab application directory is /opt/conda/share/jupyter/lab
[I 18:34:13.922 NotebookApp] Serving notebooks from local directory: /home/garystaf/work
[I 18:34:13.922 NotebookApp] The Jupyter Notebook is running at:
[I 18:34:13.922 NotebookApp] http://295787bbb825:8888/?token=4d8d03ca6b3607f0ffb24315be328324b8b0ab566cf0cc99
[I 18:34:13.922 NotebookApp] or http://127.0.0.1:8888/?token=4d8d03ca6b3607f0ffb24315be328324b8b0ab566cf0cc99
[I 18:34:13.922 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 18:34:13.926 NotebookApp]

To access the notebook, open this file in a browser:
  file:///home/garystaf/.local/share/jupyter/runtime/nbserver-22-open.html
Or copy and paste one of these URLs:
  http://295787bbb825:8888/?token=4d8d03ca6b3607f0ffb24315be328324b8b0ab566cf0cc99
  or http://127.0.0.1:8888/?token=4d8d03ca6b3607f0ffb24315be328324b8b0ab566cf0cc99
```

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*).

```
1 #!/bin/bash
2
3 set -ex
4
5 # install required python packages
6 python3 -m pip install --user --upgrade pip
7 python3 -m pip install -r requirements.txt --upgrade
8
9 # download latest postgres driver jar
10 POSTGRES_JAR="postgresql-42.2.8.jar"
11 if [ -f "$POSTGRES_JAR" ]; then
12     echo "$POSTGRES_JAR exist"
13 else
14     wget -nv "https://jdbc.postgresql.org/download/${POSTGRES_JAR}"
15 fi
16
17 # spark-submit logging level from INFO to WARN
18 sudo cp log4j.properties /usr/local/spark/conf/log4j.properties
19
20 # update/upgrade and install htop
21 sudo apt-get update -y && sudo apt-get upgrade -y
22 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`.

```
1 #!/usr/bin/python3
2
3 import random
4
5 technologies = [
6     'PySpark' 'Python' 'Spark' 'Scala' 'Java' 'Project Juniper' 'R'
```

```
PySpark, Python, Spark, Scala, Java, Project Jupyter, R
7 ]
8
9 print("Technologies: %s\n" % technologies)
10
11 technologies.sort()
12 print("Sorted: %s\n" % technologies)
13
14 print("I'm interested in learning about %s." % random.choice(technologies))
```

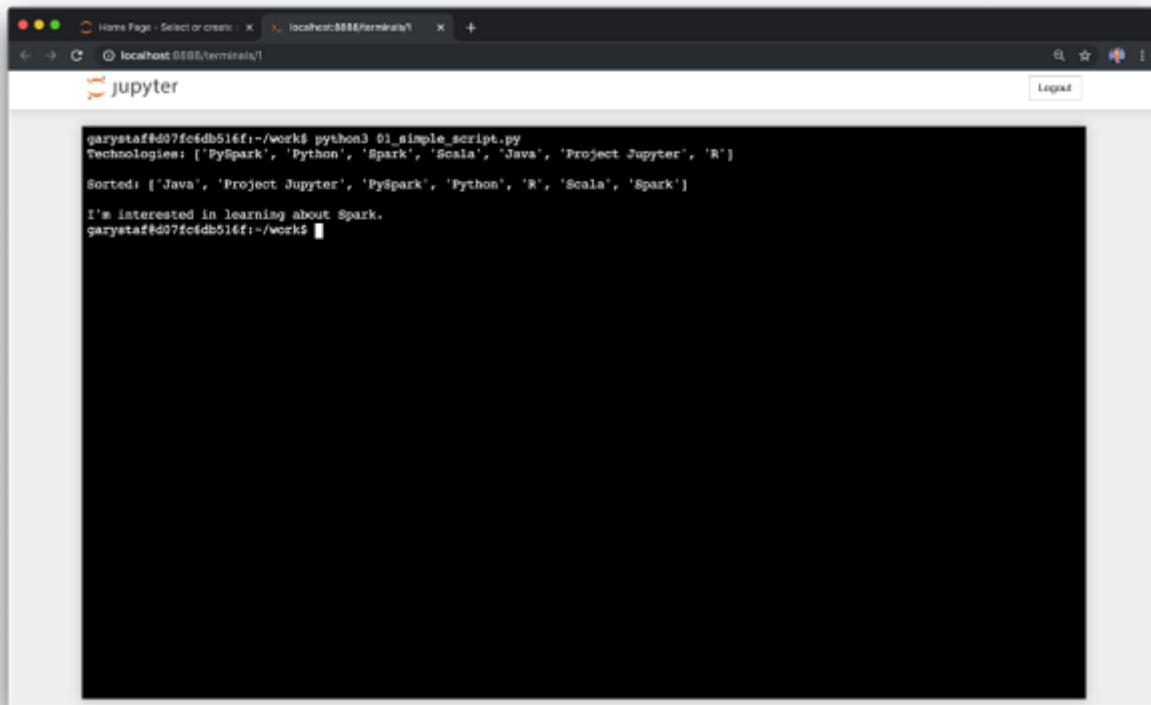
01\_simple\_script.py hosted with ❤ by GitHub

[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.



```
garytaff@d07fc4db516f:~/work$ python3 01_simple_script.py
Technologies: ['PySpark', 'Python', 'Spark', 'Scala', 'Java', 'Project Jupyter', 'R']
Sorted: ['Java', 'Project Jupyter', 'PySpark', 'Python', 'R', 'Scala', 'Spark']
I'm interested in learning about Spark.
garytaff@d07fc4db516f:~/work$
```

## 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 screenshot shows a web browser window with the URL <https://www.kaggle.com/xvivancos/transactions-from-a-bakery>. The page title is "Transactions from a bakery". The dataset has been updated 2 months ago (Version 1) by Xavier. It has 278 voters. Below the title, there are tabs for Data, Overview, Kernels, Discussion, and Activity. The Data tab is selected, showing a file named "BreadBasket\_DMS.csv" (21.3k x 4). The About this file section describes it as a dataset containing more than 6,000 transactions from a bakery. The Columns section details three columns: Date (categorical, YYYY-MM-DD format), Time (categorical, HH:MM:SS format), and Transaction (quantitative, allows differentiation between transactions sharing the same values in the first two fields). There are also API and Download All buttons.

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...				
1	Date	Time	Transaction	Item
2	2016-10-30	09:58:11	1	Bread
3	2016-10-30	10:05:34	2	Scandinavian
4	2016-10-30	10:05:34	2	Scandinavian
5	2016-10-30	10:07:57	3	Hot chocolate
6	2016-10-30	10:07:57	3	Jam
7	2016-10-30	10:07:57	3	Cookies
8	2016-10-30	10:08:41	4	Muffin
9	2016-10-30	10:13:03	5	Coffee
10	2016-10-30	10:13:03	5	Pastry
11	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.

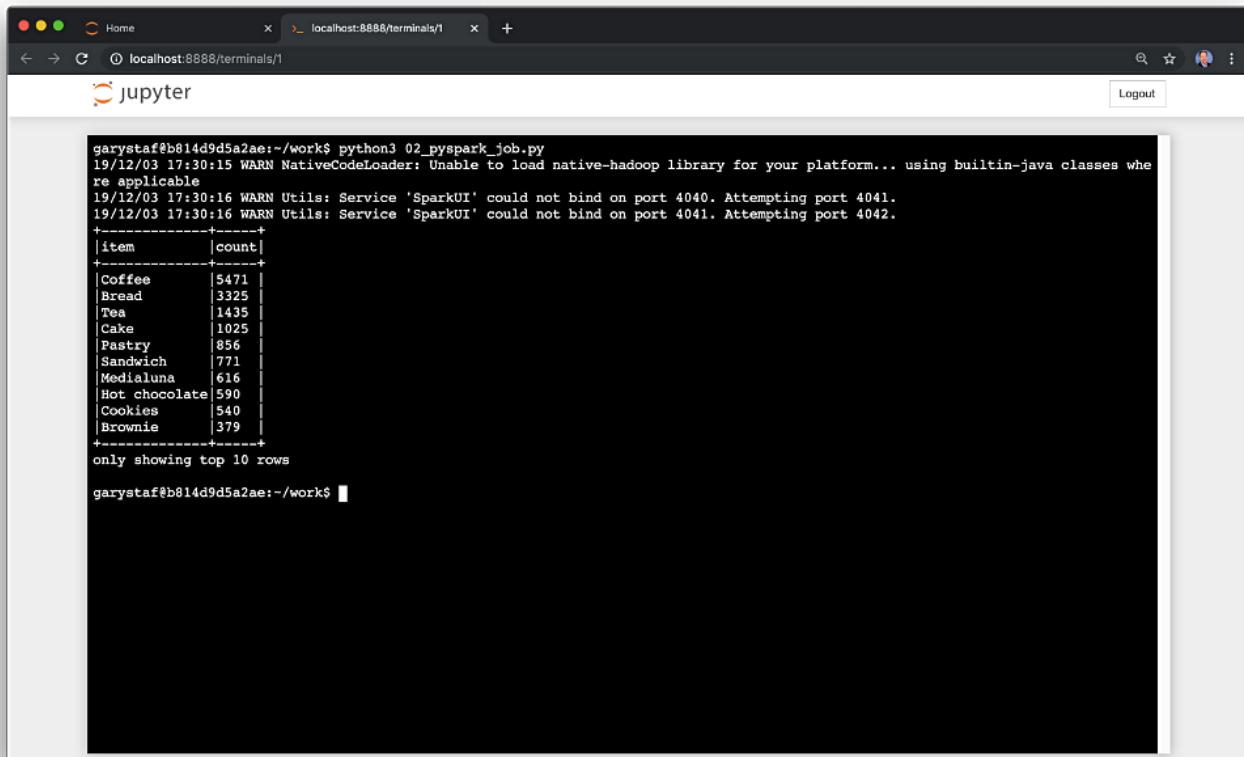
The project includes a simple Python PySpark ETL script, 02\_pyspark\_job.py. The ETL script loads the original Kaggle Bakery dataset from the CSV file into memory, into a Spark DataFrame. The script then performs a simple Spark SQL query, calculating the total quantity of each type of bakery item sold,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

```
1  #!/usr/bin/python3
2
3  from pyspark.sql import SparkSession
4
5  spark = SparkSession \
6    .builder \
7    .appName('spark-demo') \
8    .getOrCreate()
9
10 df_bakery = spark.read \
11   .format('csv') \
12   .option('header', 'true') \
13   .option('delimiter', ',') \
14   .option('inferSchema', 'true') \
15   .load('BreadBasket_DMS.csv')
16
17 df_sorted = df_bakery.cube('item').count() \
18   .filter('item NOT LIKE \'NONE\'') \
19   .filter('item NOT LIKE \'Adjustment\'') \
20   .orderBy(['count', 'item'], ascending=[False, True])
21
22 df_sorted.show(10, False)
23
24 df_sorted.coalesce(1) \
25   .write.format('csv') \
26   .option('header', 'true') \
27   .save('output/items-sold.csv', mode='overwrite')
```

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.



The screenshot shows a Jupyter terminal window titled "localhost:8888/terminals/1". The terminal session starts with the command "python3 02\_pyspark\_job.py". The output shows several warning messages from the NativeCodeLoader and SparkUI services. Below the warnings, the results of a DataFrame print operation are displayed, showing the top 10 items and their counts:

item	count
Coffee	5471
Bread	3325
Tea	1435
Cake	1025
Pastry	856
Sandwich	771
Medialuna	616
Hot chocolate	590
Cookies	540
Brownie	379

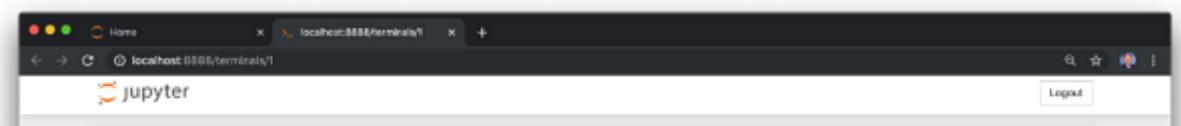
Only showing top 10 rows

garystaf@b814d9d5a2ae:~/work\$

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.



```

garystaf@b814d9d5a2ae:~/work$ $SPARK_HOME/bin/spark-submit 02_pyspark_job.py
19/12/03 17:32:05 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
19/12/03 17:32:07 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
19/12/03 17:32:07 WARN Utils: Service 'SparkUI' could not bind on port 4041. Attempting port 4042.
+-----+-----+
|item |count|
+-----+-----+
|Coffee| 5471 |
|Bread | 3325 |
|Tea   | 1435 |
|Cake  | 1025 |
|Pastry| 856  |
|Sandwich| 771 |
|Medialuna| 616 |
|Hot chocolate| 590 |
|Cookies| 540  |
|Brownie| 379  |
+-----+-----+
only showing top 10 rows
garystaf@b814d9d5a2ae:~/work$ 

```

Using the following commands, we can view the resulting CSV 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.

The screenshot shows a Jupyter Notebook interface running on a local host. The terminal window displays the command-line output of a PySpark job. The output shows the top 10 items sold, with Coffee at 5,471 and Bread at 3,325. Below the terminal, the file system is shown with the generated CSV files and their CRC checksums.

```

garystaf@b814d9d5a2ae:~/work$ $SPARK_HOME/bin/spark-submit 02_pyspark_job.py
19/12/03 17:32:05 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
19/12/03 17:32:07 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
19/12/03 17:32:07 WARN Utils: Service 'SparkUI' could not bind on port 4041. Attempting port 4042.
+-----+-----+
|item |count|
+-----+-----+
|Coffee| 5471 |
|Bread | 3325 |
|Tea   | 1435 |
|Cake  | 1025 |
|Pastry| 856  |
|Sandwich| 771 |
|Medialuna| 616 |
|Hot chocolate| 590 |
|Cookies| 540  |
|Brownie| 379  |
+-----+-----+
only showing top 10 rows
garystaf@b814d9d5a2ae:~/work$ ls -alh output/items-sold.csv/
total 12K
drwxr-xr-x 6 garystaf users 192 Dec  3 17:32 .
drwxr-xr-x 3 garystaf users  96 Dec  3 17:32 ..
-rw-r--r-- 1 garystaf users 1.4K Dec  3 17:32 part-00000-f4cb2d4d-35b0-4e69-b47e-9f1470a60742-c000.csv
-rw-r--r-- 1 garystaf users  20 Dec  3 17:32 .part-00000-f4cb2d4d-35b0-4e69-b47e-9f1470a60742-c000.csv.crc
-rw-r--r-- 1 garystaf users   0 Dec  3 17:32 _SUCCESS
-rw-r--r-- 1 garystaf users   8 Dec  3 17:32 _SUCCESS.crc
garystaf@b814d9d5a2ae:~/work$ head -5 output/items-sold.csv/*.csv
item,count
Coffee,5471
Bread,3325
Tea,1435
Cake,1025
garystaf@b814d9d5a2ae:~/work$ 

```

# 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.

```
1  #!/usr/bin/python3
2
3  import psycopg2
4
5  # connect to database
6  connect_str = 'host=postgres port=5432 dbname=bakery user=postgres password=postgres1234'
7  conn = psycopg2.connect(connect_str)
8  conn.autocommit = True
9  cursor = conn.cursor()
10
11 # execute sql script
12 sql_file = open('bakery.sql', 'r')
13 sqlFile = sql_file.read()
14 sql_file.close()
15 sqlCommands = sqlFile.split(';')
16 for command in sqlCommands:
17     print(command)
18     if command.strip() != '':
19         cursor.execute(command)
20
21 # import data from csv file
22 with open('BreadBasket_DMS.csv', 'r') as f:
23     next(f) # Skip the header row.
24     cursor.copy_from(
25         f,
26         'transactions',
27         sep=',',
28         columns=('date', 'time', 'transaction', 'item')
29     )
30 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);
```

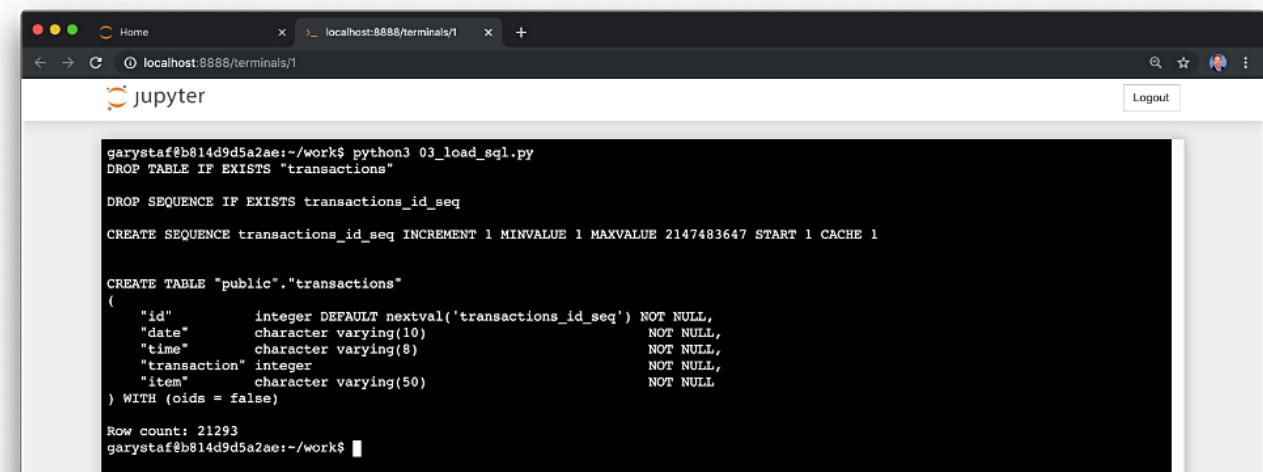
bakery.sql hosted with ❤ by GitHub

[view raw](#)

To execute the script, run the following command.

```
python3 03_load_sql.py
```

This should result in the following output, if successful.



A screenshot of a terminal window titled 'jupyter' on a Mac OS X desktop. The window shows the command 'python3 03\_load\_sql.py' being run. The terminal output displays the SQL code for creating a sequence and a table named 'transactions'. The table has columns 'id', 'date', 'time', 'transaction', and 'item'. The 'id' column is defined as an integer with a default value from the sequence 'transactions\_id\_seq', and it is marked as NOT NULL. The other four columns ('date', 'time', 'transaction', 'item') are defined as character varying types with specific lengths (10, 8, and 50 respectively), and they are also marked as NOT NULL. The output concludes with 'Row count: 21293'.

```
garystaf@b814d9d5a2ae:~/work$ python3 03_load_sql.py
DROP TABLE IF EXISTS "transactions"

DROP SEQUENCE IF EXISTS transactions_id_seq

CREATE SEQUENCE transactions_id_seq INCREMENT 1 MINVALUE 1 MAXVALUE 2147483647 START 1 CACHE 1;

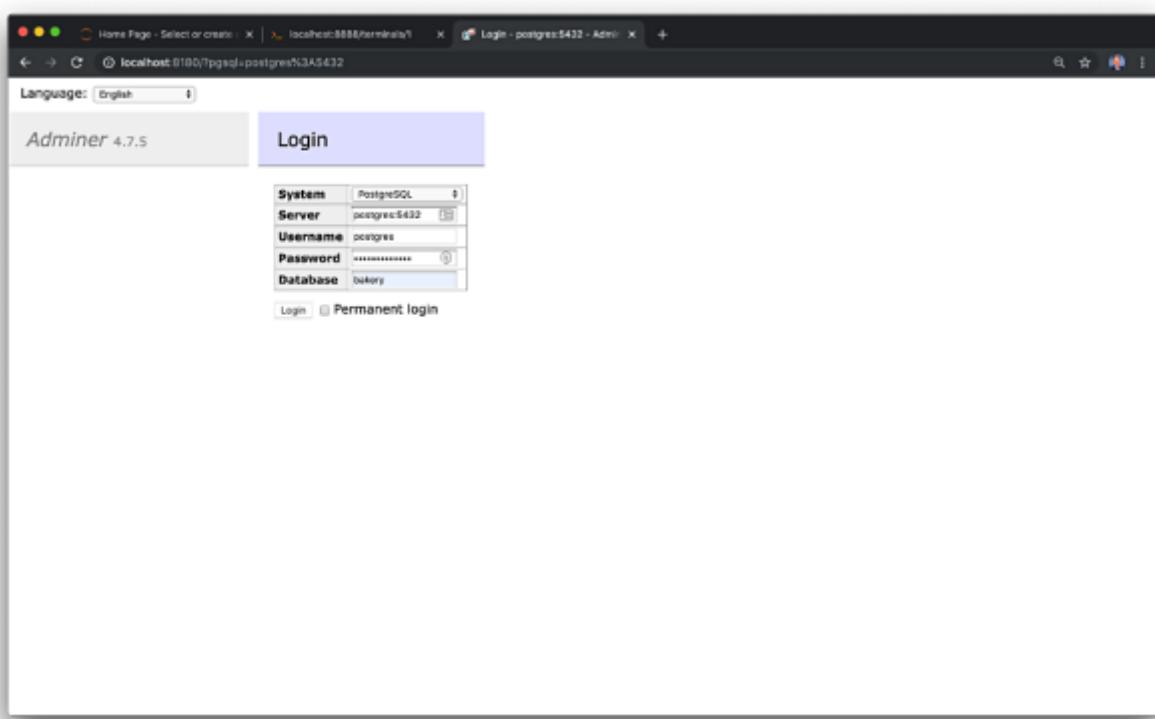
CREATE TABLE "public"."transactions"
(
    "id"          integer DEFAULT nextval('transactions_id_seq') NOT NULL,
    "date"        character varying(10)                      NOT NULL,
    "time"        character varying(8)                       NOT NULL,
    "transaction" integer                                NOT NULL,
    "item"        character varying(50)                     NOT NULL
) WITH (oids = false)

Row count: 21293
garystaf@b814d9d5a2ae:~/work$
```

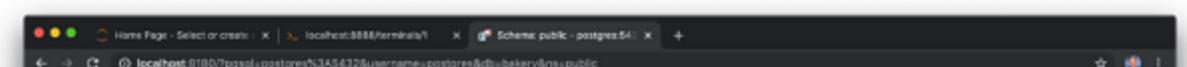
# Adminer

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.



Language: English

DB: bakery Schema: public

SQL command Import Export Create table

select transactions

Schema: public

Alter schema Database schema

Tables and views

Search data in tables (1) transactions

Table	Engine	Collation	Data Length <sup>1</sup>	Index Length <sup>1</sup>	Data Free	Auto Increment	Rows <sup>1</sup>	Comment <sup>1</sup>
transactions	table	en_US.UTF-8	1,466,368	24,576	7	?	21,293	
1 in total								

Selected (0)

Vacuum | Optimize | Translate | Drop | Move to other database: public | Move

Create table | Create view

Routines

Create function

Sequences

Name
bakery_basket_id_seq
transactions_id_seq

Create sequence

User types

Create type

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

Language: English

DB: bakery Schema: public

SQL command Import Export Create table

select transactions

Select: transactions

Select data Show structure Alter table New item

Select Search Sort Limit 10 100 Action Select

SELECT \* FROM "transactions" LIMIT 10 (21,293 rows) Edit

Modify	Id	Date	Time	transaction	Item
edit	1	2016-10-30	09:58:11	1	Bread
edit	2	2016-10-30	10:05:34	2	Scandinavian
edit	3	2016-10-30	10:05:34	2	Scandinavian
edit	4	2016-10-30	10:07:57	3	Hot chocolate
edit	5	2016-10-30	10:07:57	3	Jam
edit	6	2016-10-30	10:07:57	3	Cookies
edit	7	2016-10-30	10:08:41	4	Muffin
edit	8	2016-10-30	10:13:03	5	Coffee
edit	9	2016-10-30	10:13:03	5	Pastry
edit	10	2016-10-30	10:13:03	5	Bread

Load more data

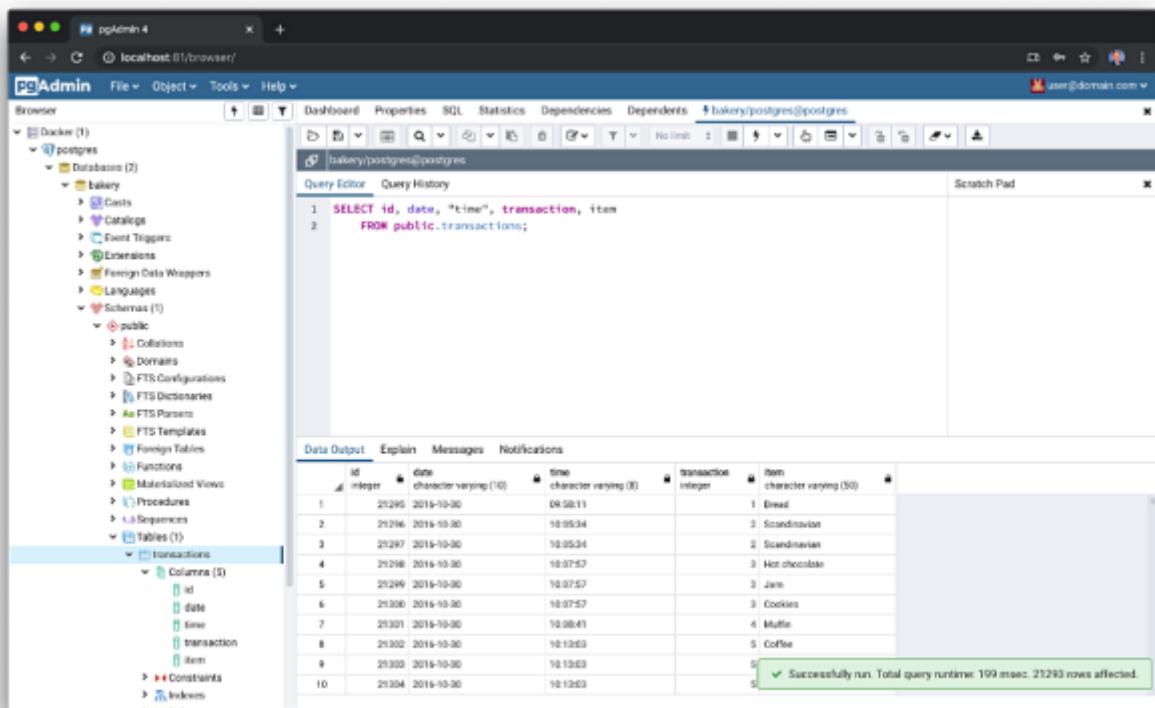
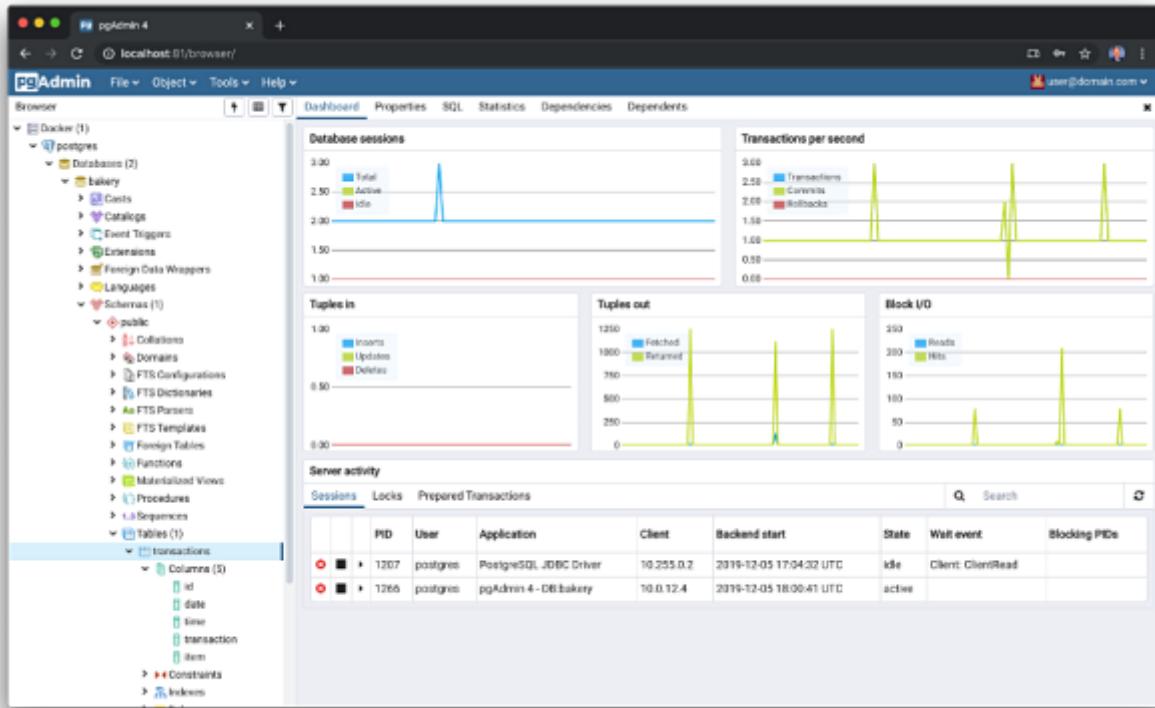
Page 1 2 3 4 5 ... last ~ 21,293 rows Modify Selected (0) Export (~ 21,293)

Import

## 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.

The screenshot shows a Jupyter Notebook interface with the title "PySpark Demo Notebook". The table of contents under "Demo" lists:

1. Read CSV-Format File
2. Run PostgreSQL Script
3. Load PostgreSQL Data
4. Create a New Record
5. Append Record to Database Table
6. Overwrite Data to Database Table
7. Analyze and Graph Data with BokehJS
8. Read and Write Data to Parquet

Prepared by: [Gary A. Nisbett](#)  
Associated article: [https://www.meijer.com/0008-671/](#)

**Read CSV-Format File**

Read CSV-format data file into a Spark DataFrame.

```
In [1]: from pyspark.sql import SparkSession  
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_schemas = StructType()  
        .StructField('date', StringType(), True),
```

### Markdown for Notebook Documentation

The screenshot shows a Jupyter Notebook interface with the title "PySpark Demo Notebook". The table of contents under "Demo" lists:

1. Read CSV-Format File
2. Run PostgreSQL Script
3. Load PostgreSQL Data
4. Create a New Record
5. Append Record to Database Table
6. Overwrite Data to Database Table
7. Analyze and Graph Data with BokehJS
8. Read and Write Data to Parquet

Prepared by: [Gary A. Nisbett](#)  
Associated article: [https://www.meijer.com/0008-671/](#)

**Read CSV-Format File**

Read CSV-format data file into a Spark DataFrame.

```
In [1]: from pyspark.sql import SparkSession
```

```

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)
])

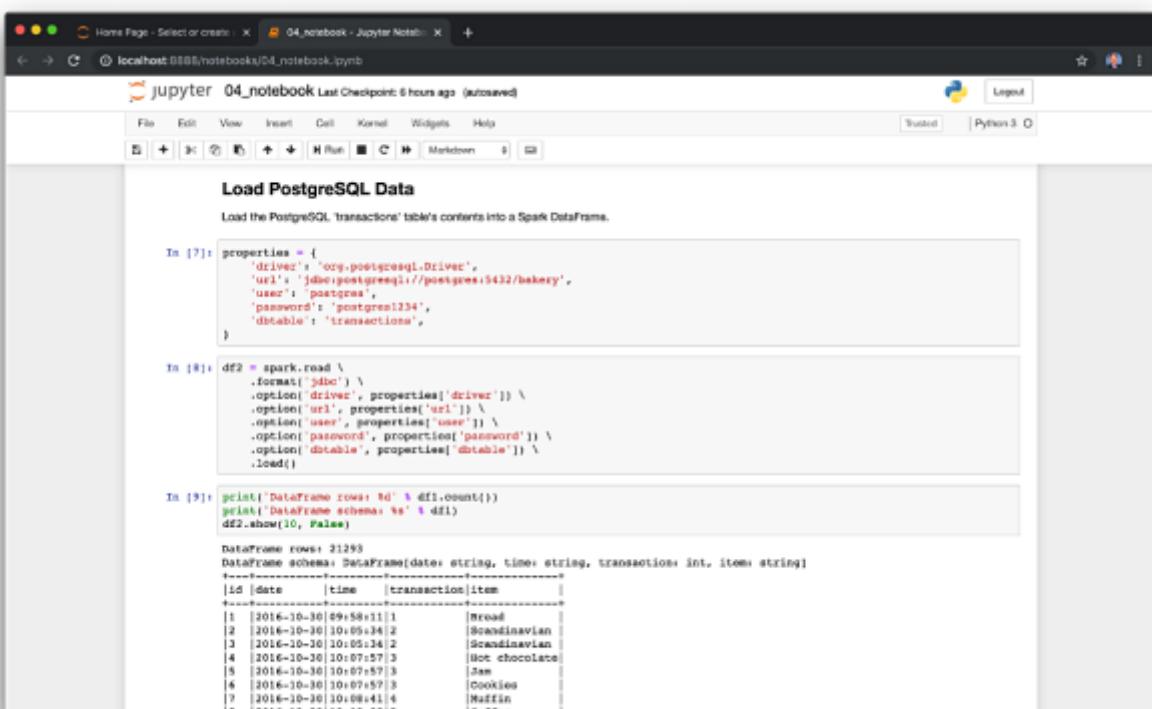
In [4]: df1 = spark.read \
    .format('csv') \
    .option('header', 'true') \
    .load('breadBasket.csv', schema=bakery_schema)

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

DataFrame rows: 21293
DataFrame schema: struct<date:string, time:string, transaction:int, item:string>
+-----+-----+-----+
|date |time |transaction|item |
+-----+-----+-----+
|2014-10-30|09:58:11|3|Bread
|2014-10-30|10:05:34|3|Scandinavian
|2014-10-30|10:05:34|2|Scandinavian

```

## Read CSV-Format Files into Spark DataFrame



The screenshot shows a Jupyter Notebook interface with the title "jupyter 04\_notebook". The code cell (In [7]) defines properties for reading from PostgreSQL:

```

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

```

The code cell (In [8]) reads the data from PostgreSQL:

```

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()

```

The code cell (In [9]) prints the DataFrame schema and shows the first 10 rows of data:

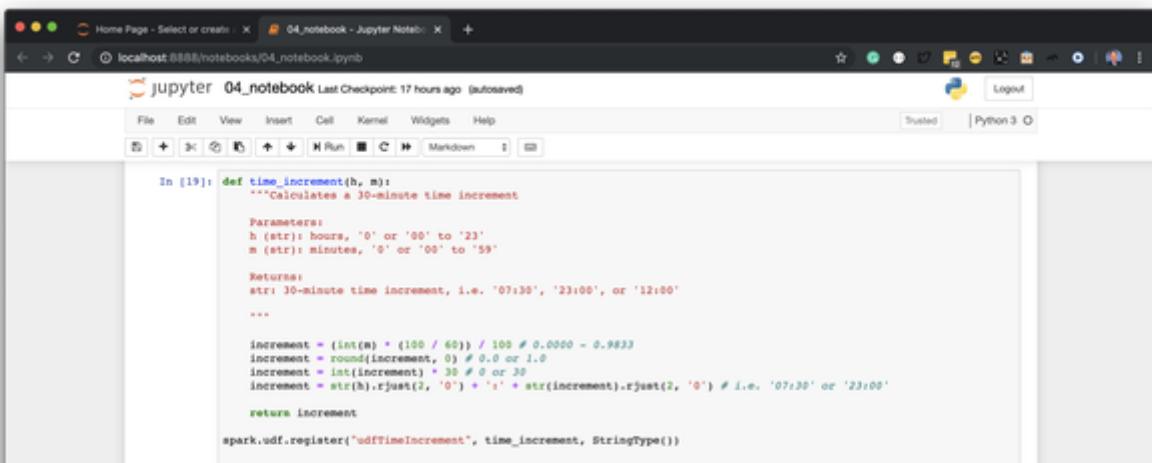
```

print('DataFrame rows: %d' % df1.count())
print('Dataframe schema: %s' % df1)
df2.show(10, False)

DataFrame rows: 21293
DataFrame schema: struct<date:string, time:string, transaction:int, item:string>
+-----+-----+-----+
|date |time |transaction|item |
+-----+-----+-----+
|3|2016-10-30|09:58:11|Bread
|2|2016-10-30|10:05:34|Scandinavian
|3|2016-10-30|10:05:34|Scandinavian
|4|2016-10-30|10:07:57|Hot chocolate
|5|2016-10-30|10:07:57|Jam
|6|2016-10-30|10:07:57|Cookies
|2|2016-10-30|10:08:41|Muffin
|8|2016-10-30|10:13:03|Coffees

```

## Load Data from PostgreSQL into Spark DataFrame



The screenshot shows a Jupyter Notebook interface with the title "jupyter 04\_notebook". The code cell (In [19]) defines a UDF named "udftimeIncrement" that calculates a 30-minute time increment:

```

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) * (60 / 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())

```

```

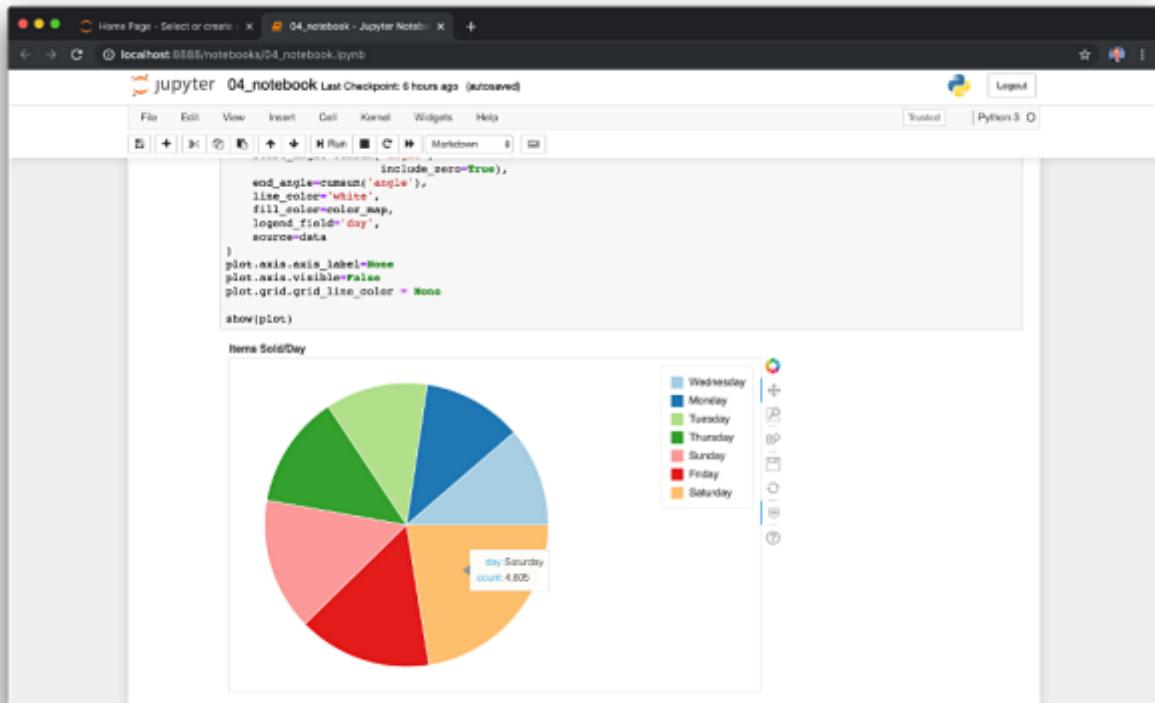
sql_query = """
WITH tmp_table AS (
    SELECT uddiffineIncrement(date_format(time, 'HH'), date_format(time, 'mm')) as period, count(*) as count
    FROM tmp_bakery
    WHERE item NOT LIKE 'NONE' AND item NOT LIKE 'Adjustment'
    GROUP BY period
    ORDER BY period ASC
)
SELECT period, count
FROM tmp_table
WHERE period BETWEEN '07:00' AND '19:00'
"""

df5 = spark.sql(sql_query)
df5.show(10, False)

+-----+
|period|count|
+-----+
|07:00 | 1 |
|08:00 | 1 |
|09:00 | 1 |
|10:00 | 1 |
|11:00 | 1 |
|12:00 | 1 |
|13:00 | 1 |
|14:00 | 1 |
|15:00 | 1 |
|16:00 | 1 |
+-----+

```

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.

```

16 df1.createOrReplaceTempView('tmp_bakery')
sql_query = 'SELECT * FROM tmp_bakery ' + \
            'ORDER BY transaction, date, time'
df4 = spark.sql(sql_query)
print('DataFrame rows: ', df4.count())
df4.show(10, False)

```

```

log4j.properties
postresql-42.2.8.jar
requirements.txt
.gitignore
LICENSE
README.md
stack.yml
External Libraries
Scratches and Consoles

175 print(DataFrame.show(10).take(10).count())
176 df4.show(10, False)
177 %%-
178
179 sql_query = "SELECT COUNT(DISTINCT item) AS item_count FROM tmp_bake"
180 df5 = spark.sql(sql_query)
181 df5.show(10, False)
182 sql_query = "SELECT item, count(*) as count " + \
183     "FROM tmp_bake" + \
184     "WHERE item NOT LIKE 'NONE'" + \
185     "GROUP BY item ORDER BY count DESC " + \
186     "LIMIT 10"
187
188 df5 = spark.sql(sql_query)
189 df5.show(10, False)
190
191 %% md
192
193 %% Graph Data with BokehJS
194 Create a vertical bar chart displaying DataFrame data
195
196 %%-
197
198 from bokeh.io import output_notebook, show
199 from bokeh.plotting import figure
200 from bokeh.models import ColumnDataSource
201 from bokeh.transform import factor_cmap

```

PyCharm 2019.2.5 (Professional Edition)

As does Visual Studio Code using the Python extension.

```

04_notebook.ipynb — pyspark-setup-demo

Append Record to Database Table
Append the contents of the DataFrame to the bakery PostgreSQL database's 'transactions' table.

(12) [M]
df3.write \
    .format('jdbc') \
    .option('driver', properties['driver']) \
    .option('url', properties['url']) \
    .option('user', properties['user']) \
    .option('password', properties['password']) \
    .option('dbtable', properties['dbtable']) \
    .mode('append') \
    .save()

(13) [M]
# should now contain one additional row of data
print("DataFrame rows: %d" % df2.count())
DataFrame rows: 21294

Overwrite Data to Database Table
Overwrite the contents of the CSV file-based DataFrame to the 'transactions' table.

(14) [M]
df1.write \
    .format('jdbc') \
    .option('driver', properties['driver']) \
    .option('url', properties['url']) \
    .option('user', properties['user']) \
    .option('password', properties['password']) \
    .option('dbtable', properties['dbtable']) \
    .option('truncate', 'true') \
    .mode('overwrite') \
    .save()

Analyze and Graph Data with BokehJS

```

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

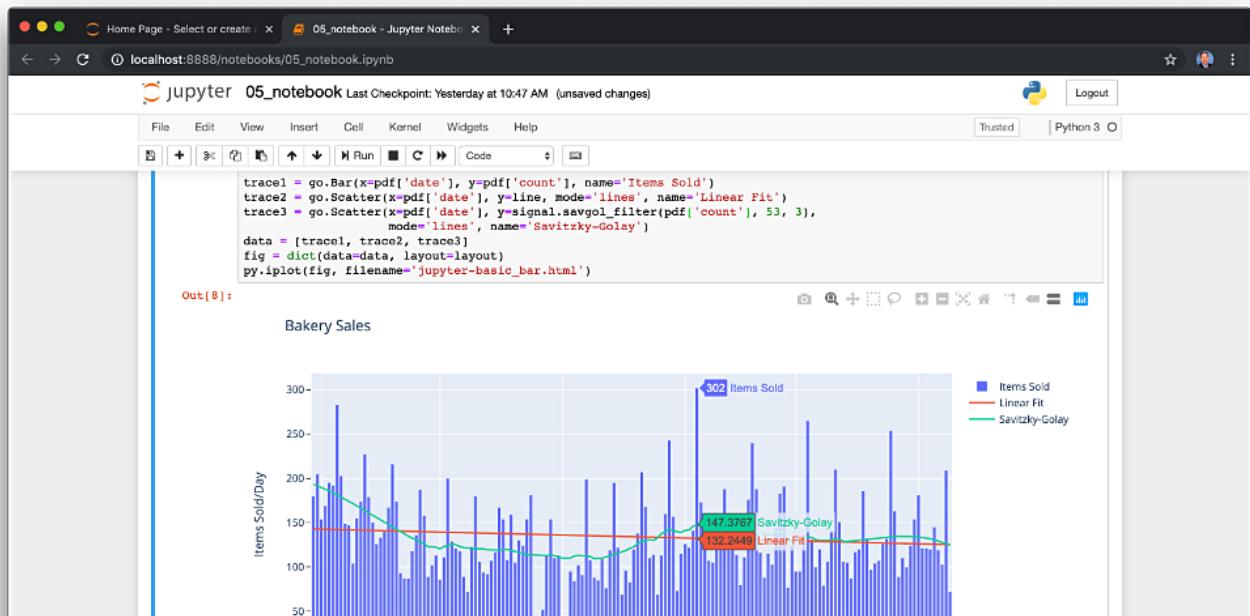
mathematics, science, and engineering, NumPy, the well-known Python package for scientific computing, and the Plotly Python Graphing Library. While NumPy and SciPy are included on the Jupyter Docker Image, the bootstrap script uses pip to install the required Plotly packages. Similar to Bokeh, shown 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 your 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.

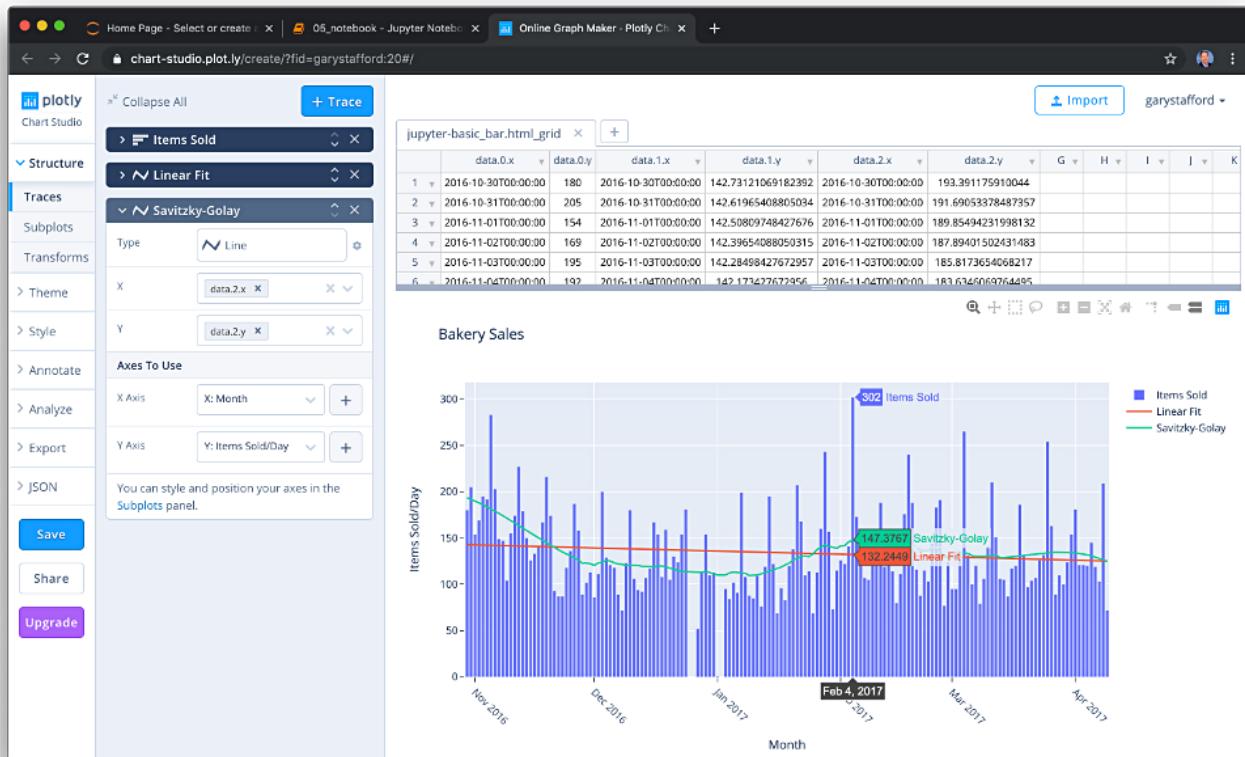
```
echo "PLOTLY_USERNAME=your-username" >> .env
echo "PLOTLY_API_KEY=your-api-key" >> .env
```

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

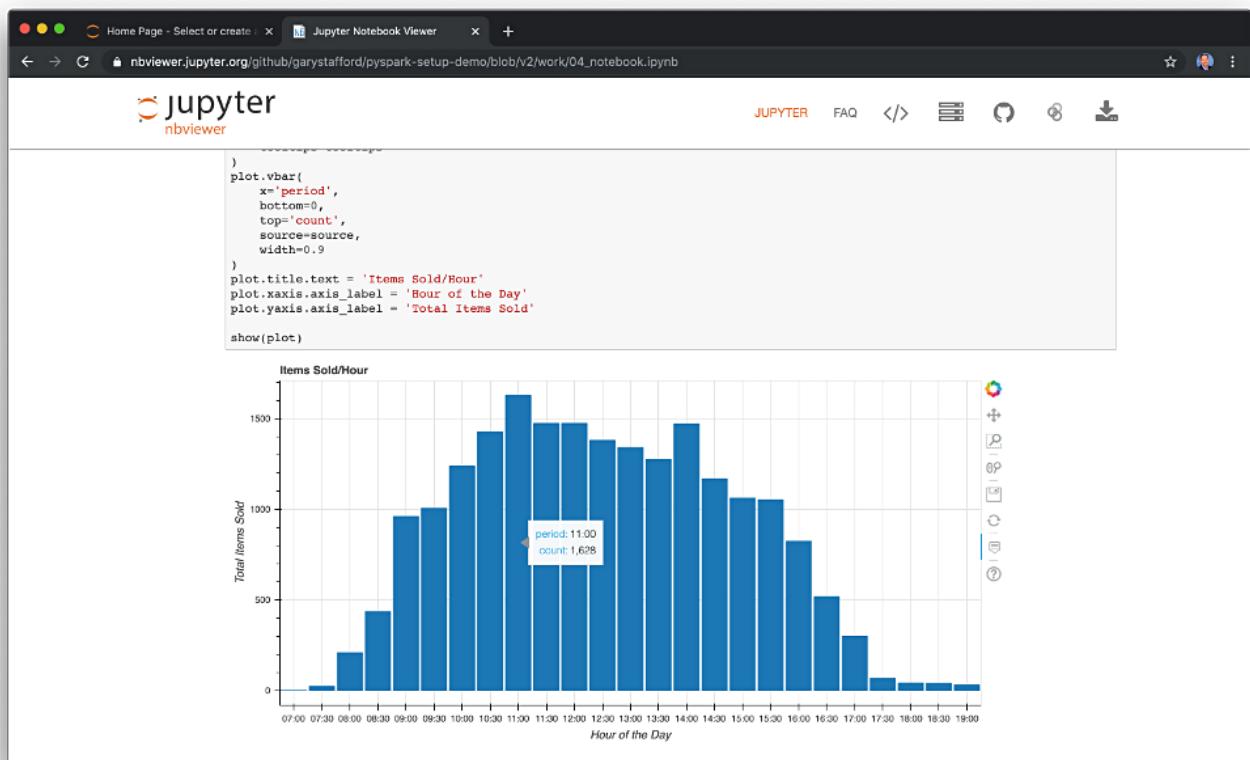
Notebooks can also be viewed using nbviewer, an open-source project under Project Jupyter. Thanks to Rackspace hosting, the nbviewer instance is a free service.



The screenshot shows the nbviewer interface with several sections:

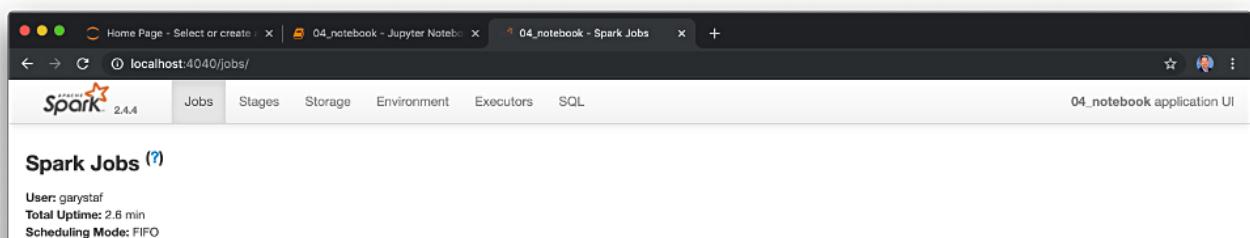
- Programming Languages** section:
  - IPython**: Shows a screenshot of an IPython notebook cell with code and output.
  - Ruby**: Shows a screenshot of an IRuby notebook cell with code and output.
  - IJulia**: Shows a screenshot of an IJulia preview cell with code and output.
- Books** section:
  - Python for Signal Processing**
  - O'Reilly Book**
  - Probabilistic Programming**

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.

localhost:4040/jobs/job?id=23

Jobs Stages Storage Environment Executors SQL 04\_notebook application UI

### Details for Job 23

Status: SUCCEEDED  
Completed Stages: 3

► Event Timeline  
▼ DAG Visualization

▼ Completed Stages (3)

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.

localhost:4040/stages/stage?id=48&attempt=0

Jobs Stages Storage Environment Executors SQL 04\_notebook application UI

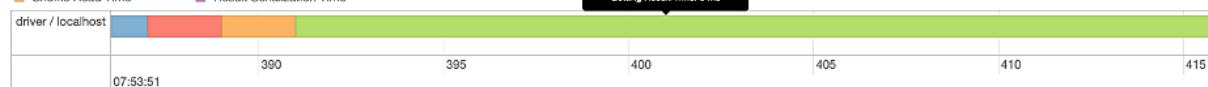
## Details for Stage 48 (Attempt 0)

Total Time Across All Tasks: 27 ms  
 Locality Level Summary: Any: 1  
 Shuffle Read: 11.5 KB / 200

- ▶ DAG Visualization
- ▶ Show Additional Metrics
- ▼ Event Timeline
- Enable zooming

Scheduler Delay   Executor Computing Time   Getting Result Time  
 Task Deserialization Time   Shuffle Write Time  
 Shuffle Read Time   Result Serialization Time

Task 0 (attempt 0)  
 Status: SUCCESS  
 Launch Time: 2019/12/05 07:53:51  
 Scheduler Delay: 1 ms  
 Task Deserialization Time: 2 ms  
 Shuffle Read Time: 2 ms  
 Executor Computing Time: 26 ms  
 Shuffle Write Time: 0 ms  
 Result Serialization Time: 0 ms  
 Getting Result Time: 0 ms



### Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	27 ms	27 ms	27 ms	27 ms	27 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Read Size / Records	11.5 KB / 200	11.5 KB / 200	11.5 KB / 200	11.5 KB / 200	11.5 KB / 200

### ▼ Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Shuffle Read Size / Records	Blacklisted
driver	295787bbb825:43505	30 ms	1	0	0	1	11.5 KB / 200	false

### ▼ Tasks (1)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Time	Duration	GC Time	Shuffle Read Size / Records	Errors
0	2488	0	SUCCESS	ANY	driver	localhost	2019/12/05 07:53:51	27 ms	0 ms	11.5 KB / 200	

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.

Home Page - Select or create | 04\_notebook - Jupyter Notebook | 04\_notebook - Environment

localhost:4040/environment/

Spark 2.4.4 Jobs Stages Storage Environment Executors SQL 04\_notebook application UI

## Environment

### Runtime Information

Name	Value
Java Version	1.8.0_222 (Private Build)
Java Home	/usr/lib/jvm/java-8-openjdk-amd64/jre
Scala Version	version 2.11.12

### Spark Properties

Name	Value
spark.app.id	local-1575532400338
spark.app.name	04_notebook
spark.driver.extraClassPath	postgresql-42.2.8.jar
spark.driver.host	295787bbb825
spark.driver.port	36539
spark.executor.id	driver
spark.master	local[]
spark.rdd.compress	True
spark.scheduler.mode	FIFO
spark.serializer.objectStreamReset	100
spark.submit.deployMode	client
spark.ui.showConsoleProgress	true

### System Properties

Name	Value

# 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.

```
NAME          CPU %      MEM USAGE / LIMIT      MEM %
jupyter_adminer.1.runtuk71zmf7h4kodzz3r6a0  0.01%      3.633MiB / 3.855GiB  0.09%
jupyter_postgres.1.umy5lolynm0msm0odzhle2jbc  0.09%      8.652MiB / 3.855GiB  0.22%
jupyter_spark.1.ui4xt552fx79ojr5laq89unp0    0.01%      103.3MiB / 3.855GiB  2.62%
postgres_pgadmin.1.95uxjb765jdiunpp0wtd3411k  0.09%      3.66MiB / 3.855GiB   0.09%
```

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.

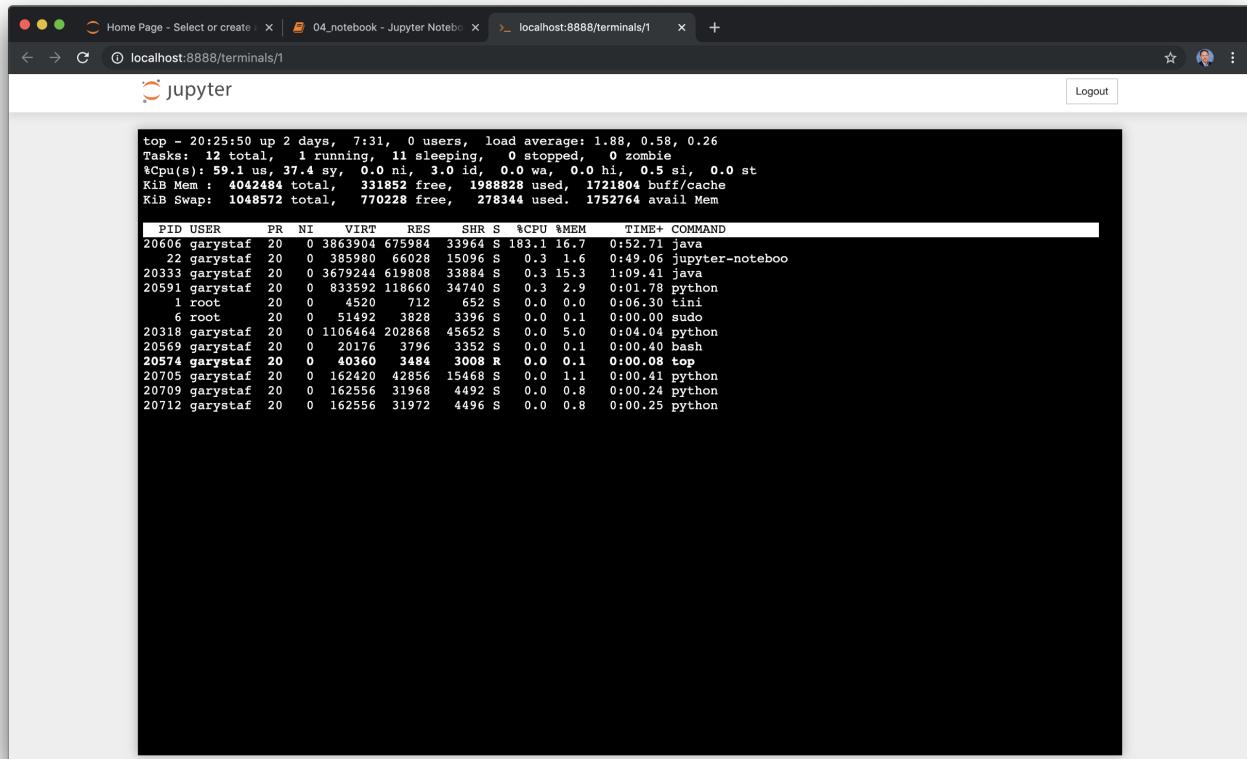
```
NAME          CPU %      MEM USAGE / LIMIT      MEM %
jupyter_adminer.1.runtuk71zmf7h4kodzz3r6a0  0.01%      3.633MiB / 3.855GiB  0.09%
jupyter_postgres.1.umy5lolynm0msm0odzhle2jbc  48.35%     13.5MiB / 3.855GiB   0.34%
jupyter_spark.1.ui4xt552fx79ojr5laq89unp0    111.71%    1.376GiB / 3.855GiB  35.70%
postgres_pgadmin.1.95uxjb765jdiunpp0wtd3411k  0.04%      3.66MiB / 3.855GiB   0.09%
```

# 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.



The screenshot shows a Jupyter terminal window with the title bar "localhost:8888/terminals/1". The window displays the output of the `top` command. The output shows system statistics and a list of processes. The processes are sorted by CPU usage (%CPU). The top process is a Java application named "java" with PID 20606, using 183.1% of one CPU. Other processes include "jupyter-notebook" (PID 22), "python" (PID 20591), "tini" (PID 1), "sudo" (PID 6), "python" (PID 20318), "bash" (PID 20569), "top" (PID 20574), "python" (PID 20705), "python" (PID 20709), and "python" (PID 20712).

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
20606	garystaf	20	0	3863904	675984	33964	S	183.1	16.7	0:52.71	java
22	garystaf	20	0	385980	66028	15096	S	0.3	1.6	0:49.06	jupyter-noteboo
20333	garystaf	20	0	3679244	619808	33884	S	0.3	15.3	1:09.41	java
20591	garystaf	20	0	833592	118660	34740	S	0.3	2.9	0:01.78	python
1	root	20	0	4520	712	652	S	0.0	0.0	0:06.30	tini
6	root	20	0	51492	3828	3396	S	0.0	0.1	0:00.00	sudo
20318	garystaf	20	0	1104464	202868	45652	S	0.0	5.0	0:04.04	python
20569	garystaf	20	0	20176	3796	3352	S	0.0	0.1	0:00.40	bash
20574	garystaf	20	0	40360	3484	3008	R	0.0	0.1	0:00.08	top
20705	garystaf	20	0	162420	42856	15468	S	0.0	1.1	0:00.41	python
20709	garystaf	20	0	162556	31968	4492	S	0.0	0.8	0:00.24	python
20712	garystaf	20	0	162556	31972	4496	S	0.0	0.8	0:00.25	python

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.

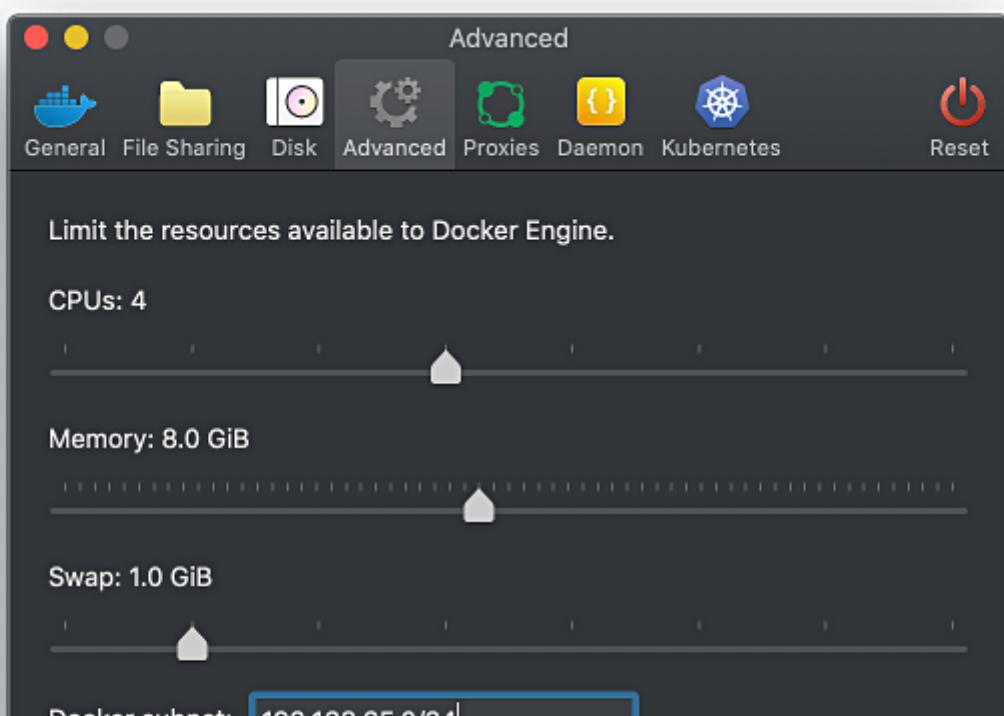
The screenshot shows a terminal window titled "jupyter" with the following content:

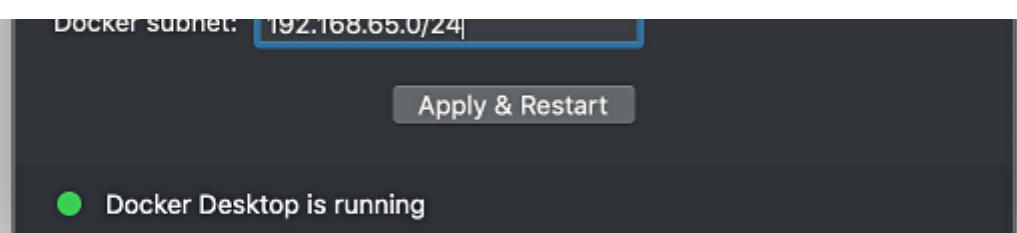
```
1 [          ] 95.9] Tasks: 9, 196 thr; 4 running
2 [          ] 96.7%] Load average: 1.18 0.43 0.20
3 [          ] 94.7%] Uptime: 00:10:25
4 [          ] 93.8%]
Mem[          ] 1 47G/3.85G
Swap[          ] 0K/1024M

PID USER PRI NI VIRT RES SHR CPU% MEM% TIME+ Command
1031 garystaf 20 0 4648M 631M 33760 S 27.3 16.0 0:07.84 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1125 garystaf 20 0 4562M 369M 33936 S 109. 9.4 0:07.84 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp postgresq
1059 garystaf 20 0 4648M 631M 33760 R 55.8 16.0 0:06.36 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1201 garystaf 20 0 4562M 369M 33936 R 55.1 9.4 0:02.46 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp postgresq
1058 garystaf 20 0 4648M 631M 33760 R 53.8 16.0 0:06.09 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1155 garystaf 20 0 4562M 369M 33936 R 30.3 9.4 0:01.59 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp postgresq
1190 garystaf 20 0 4648M 631M 33760 R 25.5 16.0 0:00.50 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1060 garystaf 20 0 4648M 631M 33760 S 22.9 16.0 0:02.79 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1192 garystaf 20 0 4648M 631M 33760 S 19.5 16.0 0:01.06 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1198 garystaf 20 0 4648M 631M 33936 R 17.5 16.0 0:00.90 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1124 garystaf 20 0 4648M 631M 33760 S 16.1 16.0 0:01.80 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1194 garystaf 20 0 4648M 631M 33760 S 16.1 16.0 0:01.01 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1154 garystaf 20 0 4562M 369M 33936 R 16.1 9.4 0:01.58 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp postgresq
1068 garystaf 20 0 4648M 631M 33760 S 12.8 16.0 0:04.88 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1101 garystaf 20 0 4648M 631M 33760 S 8.1 16.0 0:00.39 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1156 garystaf 20 0 4562M 369M 33936 R 7.4 9.4 0:00.58 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp postgresq
1096 garystaf 20 0 4648M 631M 33760 S 6.7 16.0 0:00.44 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1189 garystaf 20 0 4648M 631M 33760 S 2.0 16.0 0:00.72 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1071 garystaf 20 0 4648M 631M 33760 S 2.0 16.0 0:00.10 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1072 garystaf 20 0 4648M 631M 33760 S 2.0 16.0 0:00.10 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1172 garystaf 20 0 4648M 631M 33760 S 2.0 16.0 0:00.08 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1054 garystaf 20 0 4648M 631M 33760 S 1.3 16.0 0:00.29 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1069 garystaf 20 0 4648M 631M 33760 S 1.3 16.0 0:00.07 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1070 garystaf 20 0 4648M 631M 33760 S 1.3 16.0 0:00.07 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1143 garystaf 20 0 4648M 631M 33760 S 1.3 16.0 0:00.06 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1170 garystaf 20 0 4648M 631M 33760 S 1.3 16.0 0:00.06 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1052 garystaf 20 0 4648M 631M 33760 S 0.7 16.0 0:00.20 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1050 garystaf 20 0 4648M 631M 33760 S 0.7 16.0 0:00.19 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
1144 garystaf 20 0 4648M 631M 33760 S 0.7 16.0 0:00.05 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
23 garystaf 20 0 374M 62600 15260 S 0.7 1.5 0:04.73 /opt/conda/bin/python /opt/conda/bin/jupyter-notebook
978 garystaf 20 0 25764 4164 3380 R 0.7 0.1 0:00.12 htop --sort-key PERCENT_CPU
1102 garystaf 20 0 4648M 631M 33760 S 0.7 16.0 0:00.03 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -cp /usr/loc
```

F1 Help F2 Setup F3 Search F4 Filter F5 Tree F6 Sort By F7 Nice F8 Nice F9 Kill F10 Quit

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
```

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’.

The screenshot shows the Jupyter nbextensions\_configurator interface. At the top, there's a navigation bar with tabs for 'Files', 'Running', 'IPython Clusters', and 'Nbextensions'. Below the navigation bar is a search bar labeled 'filter: by description, section, or tags'. The main content area is titled 'Configurable nbextensions' and contains a large list of checkboxes for various extensions. The 'spellchecker' extension is checked and highlighted with a blue background. Other extensions listed include 'Codefolding', '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.

The screenshot shows the Jupyter Notebook interface. The menu bar includes 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Navigate', 'Widgets', and 'Help'. The 'View' menu has several new items: 'Codefolding', 'Execution Time', 'Table of Contents', 'Line Numbers', and 'Toggle All'. The code editor shows a Python cell with syntax highlighting and line numbers. The cell content is as follows:

```

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')

    return increment # i.e. '07:30' or '23:00'

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

```

```
27     " WHERE item NOT LIKE 'NONE' AND item NOT LIKE 'Adjustment' " \
28     " GROUP BY period " \
29     " ORDER BY period ASC" \
30   ) " \
31   "SELECT period, count " \
32   "FROM tmp_table" \
33   "WHERE period BETWEEN '07:00' AND '19:00'" \
34 
35 df5 = spark.sql(sql_query)
36 df5.show(10, False)
executed in 3.40s, finished 19:45:53 2019-12-05
```

## 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](#)

[About](#)   [Help](#)   [Legal](#)