Week 7/8 Final Project

Inverse Index with Pyspark

Trevor Thomas

College of Computer and Information Sciences, Regis University

MSDS 610: Data Engineering

Paul Andrus

June 28, 2020

Overview and Objectives

With the exponential growth in the size and complexity of data over the past several decades, new methods and programming paradigms have been developed to handle the computationally intensive processing of large data-sets. The MapReduce algorithm was developed by Google in the mid-2000's as a way to index the internet and improve their search algorithms. This led to the development of Hadoop, a distributed computation framework that was able to handle large quantities of data on affordable and easily scalable hardware. Hadoop revolutionized the data culture because it removed the need to throw out some forms of unstructured data due to hardware and schema constraints. The one major weakness for Hadoop was speed – its method for handling fault tolerance on cheaper hardware (that was expected to fail eventually) led to redundant and regular reading and writing to disk. Spark came along in the late 2000's to address that weakness. Spark also used a distributed framework to handle large amounts of data, but it processed everything in memory, effectively allowing for real-time analysis.

The objective of this project is to utilize Spark to create an inverted index of Stack Overflow posts based on their tags. These tags are general concepts relevant to each individual post – some posts have many tags, while others have none. I will take a year's worth of tagged Stack Overflow posts (omitting those without tags), and use MapReduce within Spark to create an index where each respective tag is a key, and a list of each post id where that tag is referenced is the value pair. The tools I intend to use to accomplish this are SQL, Python, PySpark (Spark-Python API), Anaconda, Hadoop, and a Linux terminal. I will first download and install Hadoop from the Apache website and configure the framework. Then I plan to obtain the Stack Overflow data from data.stackexchange.com using a SQL query, saving the output to a .csv file. I will load this file into HDFS, install PySpark using `pip`, and then pull the data from HDFS into a PySpark shell. Next, I will utilize two different MapReduce processes to complete the inverted index – the first will process the data exclusively using the Spark

RDD data structure, and the second will do the same with the PySpark Data Frame structure. Finally I will analyze the output, compare the RDD and Data Frame methods, and save the final product to a .csv file. Each of these steps is detailed below, followed by a discussion of the project and its takeaways.

Methods and Results

Install Hadoop

Installing and configuring Hadoop on a local machine is an involved process that entails setting up a non-root user, establishing passwordless SSH for the Hadoop user, ensuring the appropriate version on Java is installed, downloading and extracting the Hadoop distribution itself, and finally configuring system files. When the installation is complete, Hadoop will be organized in a pseudo-distributed mode where it will launch separate JVMs to establish a "cluster" on my machine, with each node using its own Java process. The passwordless SSH allows the master node to connect to each node of the virtual cluster to mimic what a truly distributed Hadoop environment would look like. The processing power will be understandably less than with a true cluster, but it will be more than sufficient for the purpose of storing files in HDFS for eventual connection to Spark. There are many different instruction sets for this process available online with slight differences, but I utilized the directions by Vladimir Kaplarevic from the IT support company PhoenixNAP (Kaplarevic 2020).

Set up Hadoop User with Passwordless SSH

Setting up a new user without root privileges was an optional but recommended step for improving security and efficient management of the environment (Kaplarevic 2020). I started by logging into my standard user account with root access to install OpenSSH and create the new user profile called hdoop. Then I used `su` to switch profiles to the new user, 'hdoop,' in my terminal.

```
dpredbeard@busy-child:~$ sudo apt install openssh-server openssh-client -y
dpredbeard@busy-child:~$ sudo adduser hdoop
```

```
dpredbeard@busy-child:~$ su - hdoop

hdoop@busy-child:~$
```

Next, I established permission for passwordless SSH to the localhost. As indicated previously, this allows the local Hadoop environment to launch and connect multiple JVMs to create the pseudo-cluster. This involved generating an SSH key pair with `ssh-keygen`, saving them to the '/.ssh/authorized_keys' directory with `cat`, and granting permission for the new user profile to access them without a password using `chmod`.

```
hdoop@busy-child:~$ ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa

hdoop@busy-child:~$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

hdoop@busy-child:~$ chmod 0600 ~/.ssh/authorized_keys
```

Download, Install, and Configure Hadoop

I first used `wget` to download the binary file for the Hadoop 2.10.0 (most recent version of Hadoop 2.x), and then I called `tar` to extract the files.

```
hdoop@busy-child:~$ wget https://downloads.apache.org/hadoop/common/hadoop-
      2.10.0/hadoop-2.10.0.tar.gz

hdoop@busy-child:~$ tar xzf hadoop-2.10.0.tar.gz
```

To configure the environment, I needed to alter a few system files as well as provide system-specific information to several of the new Hadoop files. I used the Vim editor within my terminal for this by using the `vi` command followed by the various file names I needed to access. First I had to add a large block of text to the '.bashrc' file to configure several environment variables. The code block below starts with the command used to open the file in the Vim editor followed by the block of text added to the .bashrc file.

```
hdoop@busy-child:~$ vi .bashrc

#Hadoop Related Options
export HADOOP_HOME=/home/hdoop/hadoop-2.10.0
export HADOOP_INSTALL=$HADOOP_HOME
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
```

```
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export PATH=$PATH:$HADOOP_HOME/sbin:$HADOOP_HOME/bin
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib/native"
```

I closed the file after saving the changes and used `source ~/.bashrc` to apply the changes to my current terminal session. Next, I needed to add my local Java path to the 'hadoop-env.sh' file. As with before, the following code first lists the terminal commands and then shows the text added to the file separated with an empty line. All subsequent blocks of code in this configuration section will follow the same format.

```
hdoop@busy-child:~$ vi $HADOOP_HOME/etc/hadoop/hadoop-env.sh

export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

I also needed to add a block of text to the core-site.xml file.

```
hdoop@busy-child:~$ vi $HADOOP_HOME/etc/hadoop/core-site.xml

<configuration>
<property>
  <name>hadoop.tmp.dir</name>
  <value>/home/hdoop/tmpdata</value>
</property>
<property>
  <name>fs.default.name</name>
  <value>hdfs://127.0.0.1:9000</value>
</property>
</configuration>
```

Another block had to go in the hdfs-site.xml file.

```
hdoop@busy-child:~$ vi $HADOOP_HOME/etc/hadoop/hdfs-site.xml

<configuration>
<property>
  <name>dfs.data.dir</name>
  <value>/home/hdoop/dfsdata/namenode</value>
</property>
<property>
  <name>dfs.data.dir</name>
  <value>/home/hdoop/dfsdata/datanode</value>
</property>
<property>
  <name>dfs.replication</name>
  <value>1</value>
</property>
</configuration>
```

I added the following to the mapred-site.xml file.

```
hdoop@busy-child:~$ vi $HADOOP_HOME/etc/hadoop/mapred-site.xml

<configuration>
<property>
  <name>mapreduce.framework.name</name>
  <value>yarn</value>
</property>
</configuration>
```

And, finally, I added the text below to the yarn-site.xml file, concluding the necessary file alterations.

```
hdoop@busy-child:~$ vi $HADOOP_HOME/etc/hadoop/yarn-site.xml

<configuration>
<property>
  <name>yarn.nodemanager.aux-services</name>
  <value>mapreduce_shuffle</value>
</property>
<property>
  <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
  <value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
<property>
  <name>yarn.resourcemanager.hostname</name>
  <value>127.0.0.1</value>
</property>
<property>
  <name>yarn.acl.enable</name>
  <value>0</value>
</property>
<property>
  <name>yarn.nodemanager.env-whitelist</name>
  <value>JAVA_HOME,HADOOP_COMMON_HOME,HADOOP_HDFS_HOME,
  HADOOP_CONF_DIR,CLASSPATH_PERPEND_DISTCHACHE,HADOOP_YARN_HOME,
  HADOOP_MAPRED_HOME</value>
</property>
</configuration>
```

With all required alterations completed, I simply had to configure the Hadoop namenode with `hdfs namenode -format`, and then I was able to launch Hadoop by moving into the '/hadoop-2.10.0/sbin/' directory and executing `./start-dfs.sh` and `./start-yarn.sh`.

```
hdoop@busy-child:~$ hdfs namenode -format

hdoop@busy-child:~$ cd hadoop-2.10.0/sbin/

hdoop@busy-child:~/hadoop-2.10.0/sbin$ ./start-dfs.sh

Starting namenodes on [localhost]
```

```
localhost: starting namenode, logging to
/home/hdoop/hadoop-2.10.0/logs/hadoop-hdoop-namenode-busy-child.out
localhost: starting datanode, logging to
/home/hdoop/hadoop-2.10.0/logs/hadoop-hdoop-datanode-busy-child.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /home/hdoop/hadoop-
2.10.0/logs/hadoop-hdoop-secondarynamenode-busy-child.out

hdoop@busy-child:~/hadoop-2.10.0/sbin$ ./start-yarn.sh

starting yarn daemons
starting resourcemanager, logging to /home/hdoop/hadoop-2.10.0/logs/yarn-
hdoop-resourcemanager-busy-child.out
localhost: starting nodemanager, logging to
/home/hdoop/hadoop-2.10.0/logs/yarn-hdoop-nodemanager-busy-child.out
```

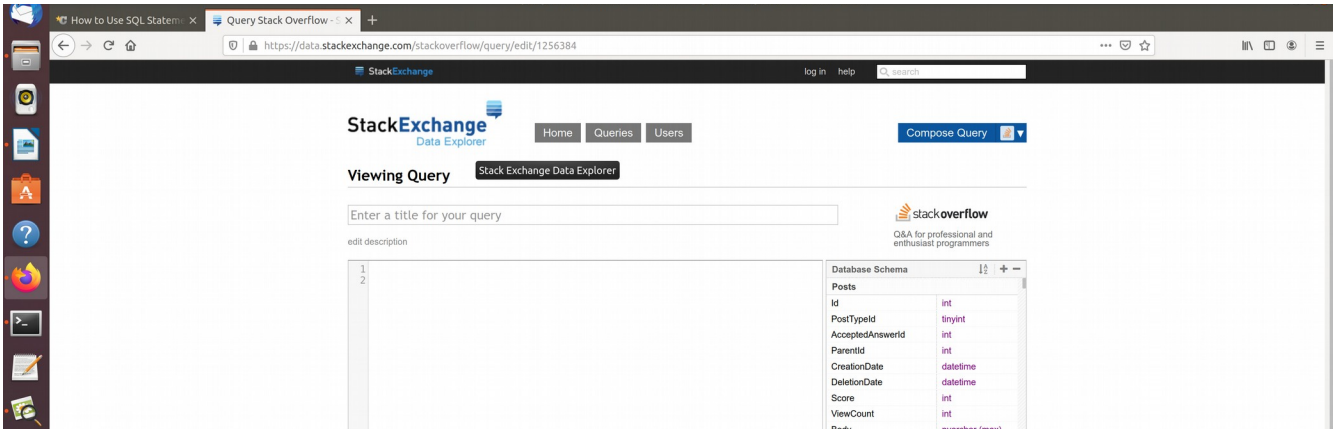
The output indicated that all processes started successfully, but as a final check I ran `jps` to confirm.

```
hdoop@busy-child:~$ jps

5152 ResourceManager
4704 DataNode
5329 NodeManager
4485 NameNode
4967 SecondaryNameNode
7694 Jps
```

The output shows all configurations worked and the Hadoop environment was up and running. At this point I needed to obtain the data in order to store it in HDFS and ultimately connect it to Spark.

Obtain Data and Load into HDFS

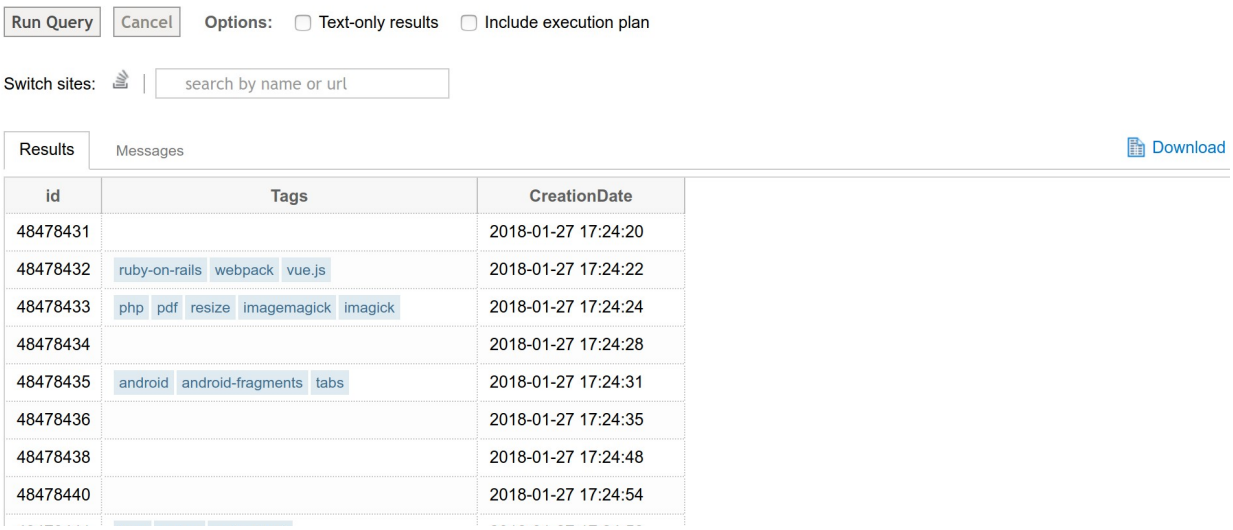
The Stack Overflow data I needed was located at data.stackexchange.com, so I went there and navigated to the Stack Overflow section. This brought up a shell for running SQL queries as well as a description of the schema.



The data I needed was the “id.” “Tags,” and “CreationDate” columns from the “Posts” table. To limit the size of the results, I restricted the output to posts from 2018 with the following query.

```
SELECT id, Tags, CreationDate From Posts
WHERE CreationDate LIKE '%2018%'
```

The query was successful, but the results contained far too many null values for posts with no tags.

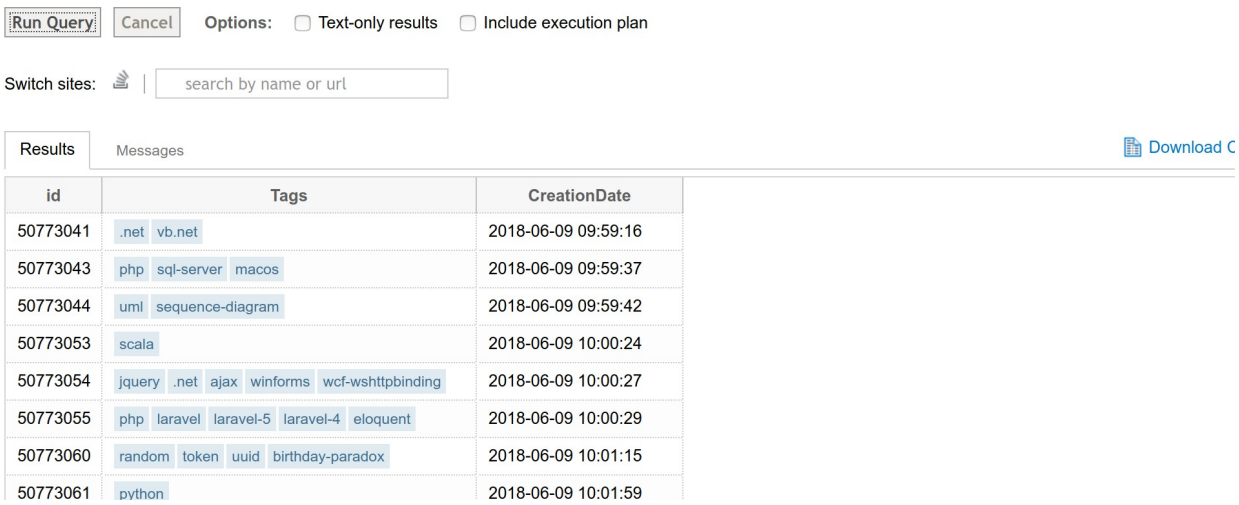


id	Tags	CreationDate
48478431		2018-01-27 17:24:20
48478432	ruby-on-rails webpack vue.js	2018-01-27 17:24:22
48478433	php pdf resize imagemagick imagick	2018-01-27 17:24:24
48478434		2018-01-27 17:24:28
48478435	android android-fragments tabs	2018-01-27 17:24:31
48478436		2018-01-27 17:24:35
48478438		2018-01-27 17:24:48
48478440		2018-01-27 17:24:54

Because my goal was to create an inverted index based on the post tags, I decided to omit the observations without a tag by expanding the `WHERE` statement as follows.

```
SELECT id, Tags, CreationDate from Posts
WHERE CreationDate LIKE '%2018%' AND Tags IS NOT NULL
```

The results were much better, with each observation containing at least one tag.



id	Tags	CreationDate
50773041	.net vb.net	2018-06-09 09:59:16
50773043	php sql-server macos	2018-06-09 09:59:37
50773044	uml sequence-diagram	2018-06-09 09:59:42
50773053	scala	2018-06-09 10:00:24
50773054	jquery .net ajax winforms wcf-wshttpbinding	2018-06-09 10:00:27
50773055	php laravel laravel-5 laravel-4 eloquent	2018-06-09 10:00:29
50773060	random token uuid birthday-paradox	2018-06-09 10:01:15
50773061	python	2018-06-09 10:01:59

I then clicked “Download CSV” to save the results to my /home directory in .csv format, and returned to the terminal to launch Hadoop and load the files into HDFS. To start Hadoop, I returned to the directory that contained the launch files ‘start-dfs.sh’ and ‘start-yarn.sh’ and ran them both. Once launched, I created the directories in HDFS using `hdfs dfs -mkdir` commands and then loaded the Stack Overflow data with `hdfs dfs -copyFromLocal`.

```
hdoop@busy-child:~/hadoop-2.10.0/sbin$ ./start-dfs.sh
hdoop@busy-child:~/hadoop-2.10.0/sbin$ ./start-yarn.sh
hdoop@busy-child:~/hadoop-2.10.0/sbin$ cd

hdoop@busy-child:~$ hdfs dfs -mkdir /stackOF
hdoop@busy-child:~$ hdfs dfs -mkdir /stackOF/input

hdoop@busy-child:~$ hdfs dfs -copyFromLocal
/home/hdoop/Documents/Final_Project/QR2.csv /stackOF/input

hdoop@busy-child:~$ hdfs dfs -ls /stackOF/input
Found 1 items
-rw-r--r--   1 hdoop supergroup    3351237 2020-06-27 09:31
/stackOF/input/QR2.csv
```

With the Hadoop environment fully configured and the data loaded successfully into HDFS, I was ready to install PySpark and begin processing.

Install PySpark and Create Inverse Index

Install PySpark

Unlike Hadoop, Pyspark was exceptionally easy to install and no configurations were necessary. I simply used `pip install pyspark` and everything else was taken care of. I was able to launch the PySpark interpreter by entering `pyspark` in my terminal, creating a new Spark Context that I could use immediately without import.

```
hdoop@busy-child:~$ pip install pyspark
hdoop@busy-child:~$ pyspark

Python 3.7.6 (default, Jan  8 2020, 19:59:22)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
20/06/27 11:28:51 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-
defaults.properties
```

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.0.0
      /_/

Using Python version 3.7.6 (default, Jan  8 2020 19:59:22)
SparkSession available as 'spark'.
>>>
```

Create Inverse Index Using RDDs

Spark has three primary data structures with which to work – Resiliant Distributed Datasets (RDDs), Data Frames (DFs), and and Datasets. RDDs were the original data structure used when Spark was created, with DFs and Datasets coming later to address specific user needs. I worked with RDDs first, as this was the primary example provided in the project instructions, but I later duplicated the results using DFs for comparison between the two. To start, still in the PySpark shell, I loaded the data into an RDD using the `sc.textFile()` command and displayed the first 5 objects with the `.take(5)` method. Because the data was loaded in HDFS, I used the Hadoop file path starting with 'hdfs://localhost:9000'.

```
>>> rdd_01 = sc.textFile('hdfs://localhost:9000/stackOF/input/QR2.csv')

>>> rdd_01.take(5)

['id,Tags,CreationDate', '"52214594","<r><shiny>","2018-09-07 02:48:44"',
'"52214598","<ios><swift><cocoa>","2018-09-07
02:49:04"','"52214602","<python><scheduled-tasks><airflow>","2018-09-07
02:50:20"','"52214603","<javascript><html><contentful>","2018-09-07
02:50:39"']
```

Looking at the objects in the RDD, there were several steps that needed to be completed to clean the data and transform it into a usable format. First, I needed to remove the quotation marks around the post ids. Additionally, I needed to remove the '<', '>', and other quotation marks from the Tags items. Then I needed to remove the CreationDate data and change the order such that each item is a tuple with the Tags item first, followed by the id item. Finally, I had to change each Tags item into a

list where each tag is its own element. I built the following function for this called ‘cleaner’ to accomplish each of these steps concurrently.

```
>>> def cleaner(rdd):
...     j = rdd.map(lambda i: i.split(','))
...     x = j.map(lambda k: (k[1].replace('><', ';')\
            .replace('<', '')\
            .replace('>', '').replace('"', ''), \
            k[0].replace('"', '')))
...     z = x.map(lambda r: (r[0].split(';'), r[1]))
...     return z
```

The first step calls the `.map()` method on the original RDD and splits it using commas as the delimiter, returning a new RDD where each element is its own list-like object. The second step (split over several lines) returns another RDD where the order of the list-like objects are changed such that ‘Tags’ is first and ‘id’ is second, removing ‘CreationDate.’ This step also cleans each ‘Tags’ object by replacing ‘><’ with ‘;’ (temporarily) and removing the quotation marks, ‘<’, and ‘>’ from the ends of each string. Finally, the last step splits each ‘Tags’ object into another list-like object, returning a new RDD made of tuple objects that contain two elements – the first is the list of tags associated with a post as a list, and the second is the post id. Calling this function with the original RDD as the argument and then calling the .take(5) method on the output reveals the new format of the data after each step has been completed.

```
>>> rdd_02 = cleaner(rdd_01)
>>> rdd_02.take(5)

[(['Tags'], 'id'), (['r', 'shiny'], '52214594'), (['ios', 'swift', 'cocoa'],
'52214598'), (['python', 'scheduled-tasks', 'airflow'], '52214602'),
(['javascript', 'html', 'contentful'], '52214603')]
```

In this format, the data is ready for the MapReduce process to build the inverted index. I created two more functions, ‘mapper’ and ‘reducer’ to split the data into a one-to-one list of individual tags and post ids and then group the ids into a list based on each tag, respectively.

```
>>> def mapper(iter):
...     op = []
...     iter = iter.collect()
...     for i in iter:
```

```
...             for p in range(len(i[0])):
...                     op.append((i[0][p], i[1]))
...     rdd = sc.parallelize(op)
...     return rdd
```

To start, ‘mapper()’ creates a new empty list ‘op’ and then converts the RDD object passed as an argument into a Python list using `.collect()`. Next, it opens a `for` loop that will iterate through each object in the list. A second `for` loop is nested in the first, this one iterating through each element of the list of ‘Tags’. For each item in a given list, it appends the tag to the empty list ‘op’, paired with the post id. Once completed, the function converts the list back into an RDD with `sc.parallelize()` and returns that RDD. By running ‘mapper()’ with the cleaned RDD as input and again calling .take(5), one can see the new RDD is made of a one-to-one mapping of tags to ids.

```
>>> rdd_03 = mapper(rdd_02)
>>> rdd_03.take(5)

[('Tags', 'id'), ('r', '52214594'), ('shiny', '52214594'), ('ios',
'52214598'), ('swift', '52214598')]
```

The ‘reducer’ function, on the other hand, was substantially simpler. All it did was take the new one-to-one mapping and group each by the tag keys and formatted the list of post ids to display as a list.

```
>>> def reducer(mapped):
...     x = mapped.groupByKey().map(lambda j: (j[0], list(j[1])))
...     return x
```

When I passed the output of the ‘mapper’ function to the ‘reducer’ and called the `.take()` method, it displayed the inverted index successfully. Because a single object in the output contained a very long list of post ids, the output was too long to copy/paste even the first whole object. I will show the first few lines of the output for context though.

```
>>> rdd_04 = reducer(rdd_03)
>>> rdd_04.take(1)

[('r', ['52214594', '50954487', '50926826', '50926851', '50946768',
'50927397', '50969948', '50947216', '50938181', '50958735', '50975289',
'50954421', '50954471', '50964069', '50964078', '50964106', '50987622',
'50934654', '50923468', '50934419', '50934450', '50969878', '50981322',
'50981360', '50993441', '50954315', '50987521', '51002990', '50997768',
'51030958', '51031095', '52166473',...
```

As seen above, the tag ‘r’ is followed by a list of each post id that originally contained ‘r’ as a tag. There were significantly more post ids in this list than shown, but it makes sense that a topic as broad as the programming language R would be referenced many times in a given year on Stack Overflow. The full output containing all objects can be seen in the accompanying ‘TrevorT_Invertdex_RDDoutput.csv’ file.

Create Inverse Index using DFs

Unlike RDDs, Data Frames have a diverse set of methods with which one can handle the cleaning, mapping, and reducing of the data efficiently. In fact, I was able to build a single function that took the original data as an argument and returned the complete inverted index with no additional steps. Before I could do that, I needed to import the library `pyspark.sql.functions` so I could utilize the `regexp_replace()` and `.explode()` functions within my code.

```
>>> from pyspark.sql.functions import *
>>> df_01 = spark.read.csv('hdfs://localhost:9000/stackOF/input/QR2.csv')

>>> def df_mapreducer(df):
...     x = df.withColumn('_c1', regexp_replace('_c1', '><', ';'))\
...     .withColumn('_c1', regexp_replace('_c1', '<', ''))\
...     .withColumn('_c1', regexp_replace('_c1', '>', ''))\
...     .select(col('_c0').alias('id'), split(col('_c1'),
            ';').alias('Tags'))\
...     .select(explode('Tags').alias('Tag'), 'id').groupBy('Tag')\
...     .agg(collect_list('id').alias('ids'))
...     return x
...
>>> df_02 = df_mapreducer(df_01)
>>> df_02.show(5)

+---------------+--------------------+
|            Tag|                 ids|
+---------------+--------------------+
| android-alarms|[48327950, 513245...|
|android-manifest|[51009216, 510309...|
|  angular-chart|          [51059690]|
|      arguments|[50997478, 521711...|
|            avx|[48372852, 501498...|
+---------------+--------------------+
only showing top 5 rows
```

While it is nice to be able to take the raw data as input and produce the desired results with a single function, it is worth noting that the code reduction in this method comes at the cost of readability. I will break everything down line by line so it is clear what is happening.

```
>>> from pyspark.sql.functions import *
>>> df_01 = spark.read.csv('hdfs://localhost:9000/stackOF/input/QR2.csv')
```

Probably the most self-explanatory lines, this imported the library required and loaded the data from HDFS into a PySpark Data Frame.

```
>>> def df_mapreducer(df):
...     x = df.withColumn('_c1', regexp_replace('_c1', '><', ';'))\
...     .withColumn('_c1', regexp_replace('_c1', '<', ''))\
...     .withColumn('_c1', regexp_replace('_c1', '>', ''))\
```

The first step took the DF passed to the function and used `.withColumn()` to replace all instances of '><' with ';' in the 'Tags' column (similar to the first step when working with an RDD). Because the data was originally in .csv format, the columns were named the default '_c0', '_c1', and '_c2' rather than 'id', 'Tags', and 'CreationDate' – these were correctly labeled later in the function. The second and third steps again used `.withColumn()`, this time to remove '<' and '>' from the ends of the 'Tags' column ('_c1').

```
...     .select(col('_c0').alias('id'), split(col('_c1'),
            ';').alias('Tags'))\
...     .select(explode('Tags').alias('Tag'), 'id').groupBy('Tag')\
...     .agg(collect_list('id').alias('ids'))
...     return x
```

The top command (spread over two lines) uses `.select()` to choose the '_c0' and '_c1' columns, rename them to 'id' and 'Tags,' and split the 'Tags' column on the ';' delimiter. The next line again selects both columns, uses `explode()` on 'Tags' to split each item in a 'Tags' list to its own row (changing the name to 'Tag' because there is only one per line now), and uses `.groupBy()` to collect the one-to-one mapping of tags and ids together according to the 'Tag' column. This completes the transformations, but the combination of the `explode()` and `.groupBy()` methods returns an object called GroupedData that is not very usable. The final line uses the `.agg()` method with the

`collect_list()` function (which is a dedicated function of the `.agg` method) to return the DF column of grouped ids, allowing for duplicates. This added step makes it so the returned object is still a DF.

```
>>> df_02 = df_mapreducer(df_01)
>>> df_02.show(5)

+----------------+--------------------+
|             Tag|                 ids|
+----------------+--------------------+
|  android-alarms|[48327950, 513245...|
|android-manifest|[51009216, 510309...|
|   angular-chart|          [51059690]|
|       arguments|[50997478, 521711...|
|             avx|[48372852, 501498...|
+----------------+--------------------+
only showing top 5 rows
```

Finally, with the 'df_mapreducer' function built, I simply passed the original variable 'df_01' as an argument and assigned the result to 'df_02.' Then I used `.show(5)` to return the top five rows of the inverted index, albeit in a different order than with the RDD (it appears alphabetic but when checking the full output, it is not). This output was significantly easier to read than that of the RDD method, allowing the user to see multiple rows at the same time because of the condensed DF format.

Analyze and Save the Output

First, I wanted to see what the top 5 tags were for the data-set. Still using the DF created in the previous step, I imported the `pandas` library and converted the PySpark DF to a `pandas` DF called df_03. I added a column to the DF containing the length of the list of post ids (meaning the number of posts that referenced a given tag) and sorted by it, and then called `.head()` on that new DF.

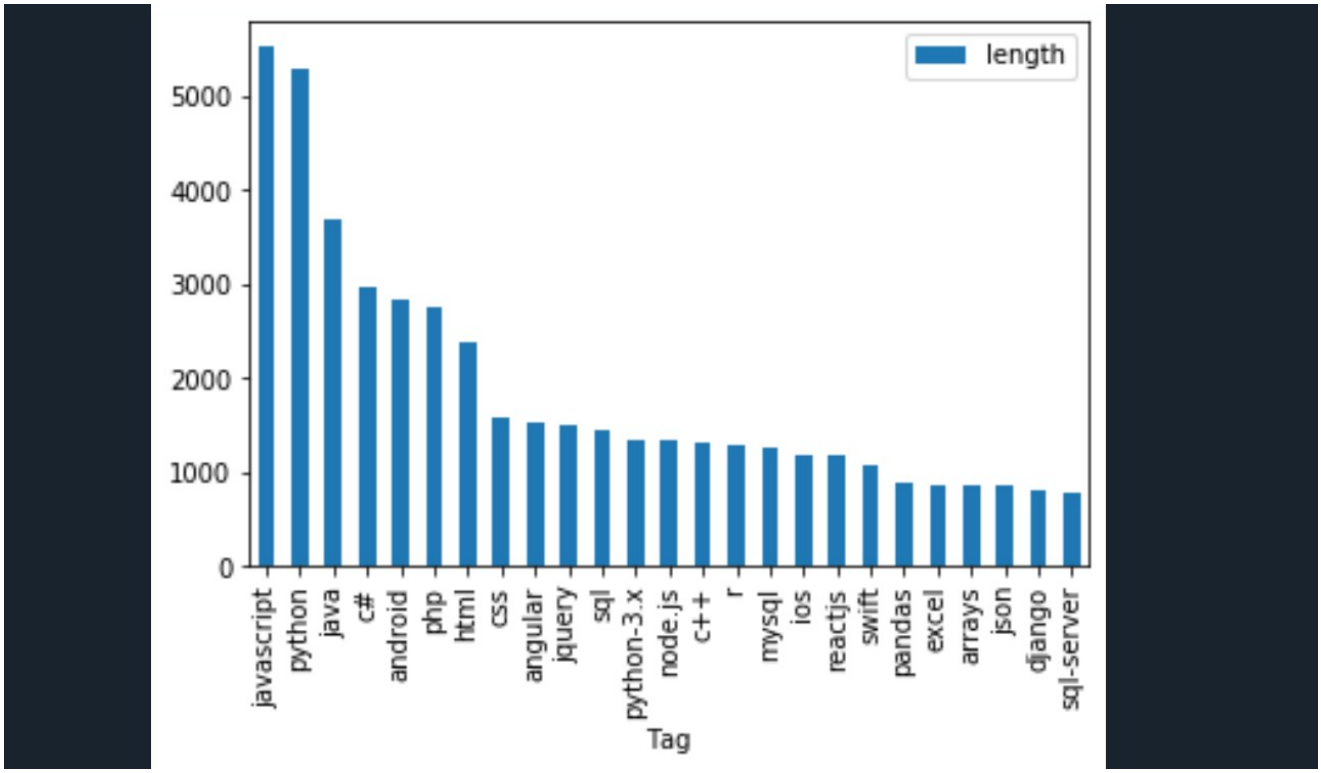
```
>>> df_03 = df_02.toPandas()
>>> df_03['length'] = df_03['ids'].str.len()

>>> sorted_df = df_03.sort_values(by='length', ascending=False)
>>> sorted_df.head()

          Tag                                                ids  length
   javascript  [50964142, 50964144, 50964188, 50964196, 50964...    5526
       python  [50964115, 50964193, 50938216, 50938235, 50938...    5281
         java  [50964120, 50926817, 50926885, 50926887, 50926...    3691
           c#  [50964210, 50938228, 50938239, 50926806, 50926...    2963
```

```
android  [50964159, 50964216, 50926829, 50926857, 50926...    2842
```

The results are not very surprising given the current programming environment. Javascript is the most popular language for web development, and Python is the most popular language for data science, so seeing them at the top was expected while both fields see rapid growth. So much software is written in Java that its third place finish is unsurprising as well, although I did not anticipate C# being on the list. The extremely popular smart phone OS Android rounds out the most frequent tags in fifth place.

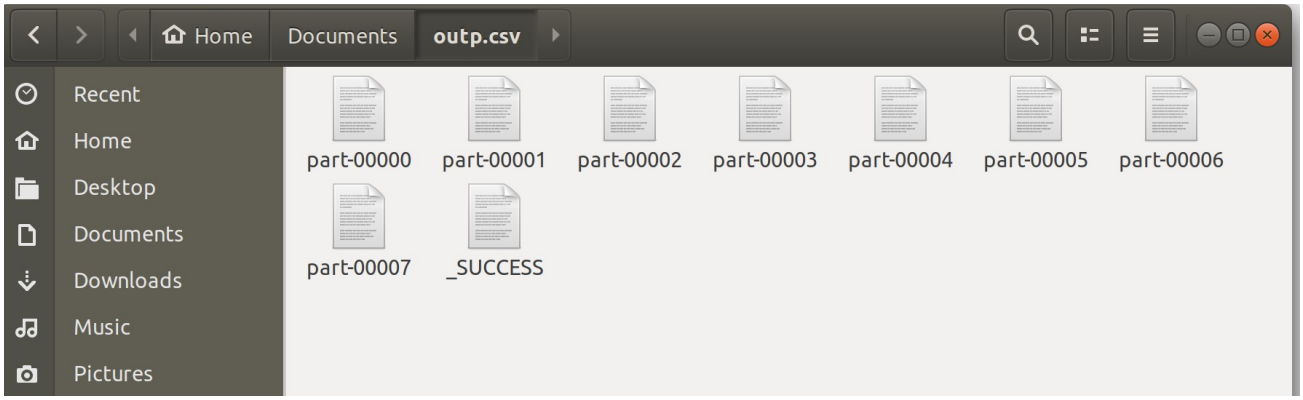
I wanted to visualize this tag popularity distribution, so I imported `matplotlib` and built a bar chart of the top 25 tags. I used the `.iloc[]` method to slice the 'sorted_df' variable to just the top 25 rows, and then used `.plot()` to assign the axes. The 'Tag' variable is on the x axis, and the new 'length' variable (representing the number of posts with that tag) is plotted along the y axis. I then ran `plt.show()` to display the results.

```
import matplotlib.pyplot as plt
sorted_df.iloc[0:25].plot(x = 'Tag', y = 'length', kind = 'bar')
plt.show()
```



Looking at the general themes here, there are a few takeaways. First, Python and Javascript are likely even more popular than their 1st and 2nd place finish indicates – Python's tags are split between 'Python,' 'pandas,' and 'Python-3.x' whereas Javascript tags are split between 'Javascript,' 'jquery,' and 'Node.js.' Programming languages dominate the top 10, but there is a bit of a shift toward data structures, databases, and data processing libraries after that.

There are many ways to save the output from a Spark session. If the final product is an RDD, as with my first run, the `.saveAsTextFile(/path/filename)` method can be called to write the results to HDFS or a local disk. Much like the output for a MapReduce task in Hadoop, this creates a separate file for each partition used for processing.



This is not an issue, as the `cat` command can be used in the terminal to merge the results into a single file, but it is still not the most user-friendly option if intending to ultimately save the results in .csv format. Because this intermediary step saves the output as raw text first before converting to .csv, the surrounding parentheses in the tuples and lists make it into the columns of the final output, seen below.

1	('r'	['52214594'	'50954487'	'50926826'	'50926851'
2	('shiny'	['52214594'	'50958735'	'51002990'	'51030885'
3	('python'	['52214602'	'50954491'	'50954503'	'50954513'
4	('2d'	['52214614'	'48324956'	'52215034'	'48373552'
5	('free'	['52214614'	'48447201'	'51116076'	'51438322'
6	('datagram'	['52214657'])			
7	('oracle'	['50954500'	'50938298'	'50926917'	'50946811'
8	('highcharts'	['50954502'	'50946798'	'51016202'	'52168944'
9	('scikit-learn'	['50954503'	'50954513'	'50938205'	'52167659'

This problem can be circumvented by converting the final product to a `pandas` Data Frame and calling the `.to_csv()` method. This process avoids the excess punctuation in the previous method, and it does so while eliminating the additional steps of merging the output directory and converting the file. Another benefit is this works regardless of the data structure used for the output. If it is an RDD, it can be converted to a PySpark DF, which in turn can be converted to a pandas DF.

```
>>> df_03.to_csv('/home/hdoop/TrevorT_Invertdex_Dframeoutput.csv')
```

	A	B	C
1		_1	_2
2	0	r	['52214594', '50954487', '50926826', '50926851', '509467
3	1	shiny	['52214594', '50958735', '51002990', '51030885', '521677
4	2	python	['52214602', '50954491', '50954503', '50954513', '509545
5	3	2d	['52214614', '48324956', '52215034', '48373552', '483721
6	4	free	['52214614', '48447201', '51116076', '51438322', '501532
7	5	datagram	['52214657']

This output is clearly better than before. The previous file would have required some cleanup to remove the excess parentheses before each key value, but this second format is clean and ready to use immediately. The full output can be seen in accompanying 'TrevorT_Invertdex_DFrameoutput.csv' file.

Discussion

For this final project, I installed Hadoop and Spark locally and used them to create an inverted index of Stack Overflow posts using two different processing strategies. I collected the data from Stack Exchange, stored in in HDFS, pulled it into Spark, and used a PySpark shell to transform it into the desired result. I first accomplished this using Spark's original data structure, the RDD, and then again with a Spark Data Frame. I briefly analyzed and visualized the final results, and finally saved the output to a .csv file. Overall, the project was very much successful, and I learned a great deal throughout the process.

Comparing the use of RDDs and DFs is difficult because they are extremely different data structures. Initially, I preferred the DF because there are more diverse methods and transformations that can be applied to them. The entire MapReduce process was effectively handled by the `.explode()` and `.groupBy()` functions, allowing me to accomplish the entire transformation with a single function. As I continued to work with RDDs, however, I eventually found this preference for DFs shifting to a lack of a preference for either. The RDD was just a more foreign data structure than the DF due to my previous experience with R and `pandas` Data Frames. Once I became familiar with what I could (and more importantly, could not) do with an RDD, it was just as easy to complete the inverted index. One benefit to the RDD process was I found the code substantially easier to read. While the DF takes full advantage of various libraries and dot notation for calling methods, the RDD looked more like "real" code made of logical `for` loops and iterable indexing and slicing. Ultimately the distinction is relatively unimportant – an RDD can be converted into a DF easily and vice versa – but I feel I benefited significantly from doing it both ways.

There is no question I learned a substantial amount from completing this project. The local installation of Hadoop alone was extremely involved and took me far outside my comfort zone. A simple mistake made during the installation took hours to track down – and it turned out a large code block intended to alter my .bashrc file that I had copy/pasted from the web had "smart" quotes in it rather than the required straight quotes; tracking that down was easily the most difficult debugging I have ever done. Once Hadoop was up and running, there was a significant learning curve when working with RDDs. As discussed above, I eventually figured out how to work with them, but there was no shortage of error messages before that. Completing the assignment using different methods gave me great insight into the power and flexibility of Spark – I would absolutely prefer it to Hadoop even without its superior speed.

As always, there is much room for improvement and future research. In particular, I would like to learn more about ways to output various files and results. The method I used works extremely well for .csv format, but I would not know where to begin if I needed a format other than .csv or .txt. Additionally, it would be very interesting to collect additional Stack Overflow data and do a deeper analysis of the findings. I could analyze what the average length of posts are for each respective tag to determine the relative complexity of various topics based on the length of questions and answers. Tracking the changes in tag prevalence over time would also be an interesting topic. I can see a near infinite collection of ways to analyze this and other data using Spark, and I am excited to make it a regular part of my data science toolkit.

Reference

Kaplarevic, Vladimir (2020, May 11). *How to Install Hadoop on Ubuntu 18.04 or 20.04*. PhoenixNAP [official website]. https://phoenixnap.com/kb/install-hadoop-ubuntu