Inverted Index Project

Lacey Conrad
MSDS 610
Regis University

June, 2020

1 Introduction

Inverted indexes are powerful tools to allow for fast text searches, such as those performed in most search engines. In simple terms, it is a data structure that directs a word to a website, document, or some other identifier (Jain, 2019). It is an evolved form of MapReduce, and as such, the output will be a key, value pair. Search engines are examples of inverted index implementation. If a search engine had to scour the entire internet every time a search was executed, it would be a very lengthy search. Instead, an index has been created behind the scenes out of available data to link key words with their respective documents. To create the index, first a forward index is created from lists of words in a document. Then, this list is inverted to form the inverted index. When inverted, the index now lists documents per word, which saves search time, memory, and other computational resources. Querying this index is substantially more efficient than querying the forward index ("Inverted index," 2005).

There are two types of inverted indexes: a record-level inverted index and a word-level inverted index. Record-level inverted indexes are lists of documents for each key (word). Word-level inverted indexes also indicate the position of a word within the document (Jain, 2019) but require more processing power and memory during creation ("Inverted index," 2005). In this project, we will be focusing on a record-level inverted index.

Data engineering, as a discipline, constantly looks for new ways to develop quicker, more efficient, and easier to use technology. In this project I will use Spark, which is a processing framework that looks to keep many of MapReduce's positive points (scalable, distributed, fault-tolerant processing), while improving on its efficiency and ease of use. Spark accomplishes this by caching data in memory across multiple parallel operations, unlike MapReduce which reads and writes from a disk (McDonald, 2018). The following image shows the amount of reading and writing to disk that occurs when multiple MapReduce jobs are chained together (each arrow is a read or write):

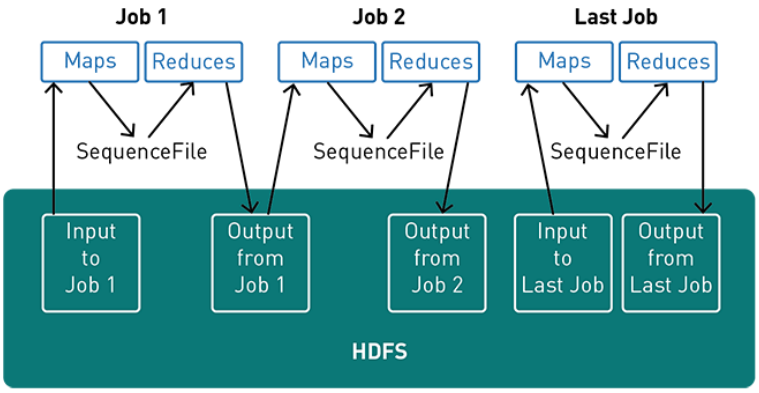

Figure 1: The chaining together of multiple MapReduce jobs. For each job, data is read from HDFS into the map process, written to and read from a sequence file, and then written to a output file from the conclusion of the reducer. (from McDonald, 2018)

The next figure illustrates the execution of a application on a Spark cluster. The process is as follows (from McDonald, 2018):

1. The SparkSession object coordinates the execution of a application as independent processes.
2. The resource manager assigns tasks to workers, one task per partition.
3. A tasks unit of work is applied to the dataset in the partition, and outputs the partitions new dataset.
4. Results are sent to the driver application (or can be saved to disk).

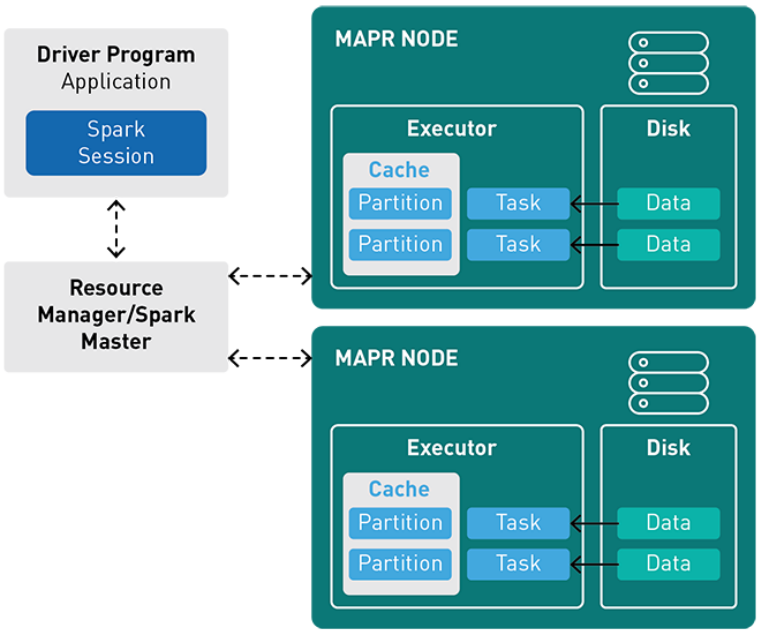

Figure 2: How a Spark application runs on a cluster. (from McDonald, 2018)

In this lab, we are utilizing data from Stack Overflow to create an inverted index of tags and post Id's in PySpark. To obtain the data, I went to `data.stackexchange.com` and created a query that returns the PostId and Tags of

a large collection of posts. After reading in the data, I stripped and split the Tags column into separate words. I then used the flatMap method to create a list of tag words with their post Id in the form of a tuple (tag word, post Id). Then, I used groupByKey to collect all of the keys that share a tag. Since the previous step formed a list of resultiterable datatypes, I converted these back into the value of the post Id's. Finally, I saved my inverted index as a text file.

2 Methods and Code

This project was performed using a PySpark kernel in Jupyter Notebook using the Google Cloud Platform. The output produced as a result of testing code, data verification or to visualize certain steps will be included in the methods section. A portion of the inverted index will be illustrated in the results section.

I first needed to obtain the data I would be using to create the inverted index. In this project, data from Stack Overflow was used. I navigated to the Stack Overflow data explorer's query page, located at `https://data.stackexchange.com/stackoverflow/query/new`. This project is aiming to create an inverted index out of Post IDs for each Tag encountered. Basically, if the tag was <Python>, the inverted index will return all the Post IDs that include <Python> as one of it's tags.

```
-- This SQL code returns the Post ID and Tags of the most active 50,000
-- posts in May, 2015, which do not have an empty Tags column.

SELECT TOP(50000) Id, Tags
FROM posts
WHERE CreationDate < '2015-06-01'
        and CreationDate > '2015-05-01'
        and Tags != '';
```

This query was ran in the StackOverflow Data Explorer, and the data was downloaded to my local PC as a CSV file. I then started up my Google Cloud cluster as instructed in the video linked in the project description. I clicked on my Google bucket, and dragged my StackOverflow data into the bucket. Next, I navigated to "Jupyter Lab", which was linked in "Web Interfaces." I started a new notebook, and started my program by loading Spark and SparkContext, followed by regex_replace.

```
# Loading in Spark and SparkContext:
spark
sc

# Importing in regexp_replace, which will help with
# text cleaning:
from pyspark.sql.functions import regexp_replace
```

Table 1: The output produced when loading Spark and SparkContext.

SparkSession - hive	
SparkContext	
Spark UI	
Version	v2.3.4
Master	yarn
AppName	PySparkShell

I obtained my Google Bucket address by copying the address associated with my cluster in the "Configuration" menu. The StackOverflow data from my Googled bucket was then loaded as a dataframe using Spark's read function. To verify that the data looked similar to what I observed on StackOverflow, I used the `show` command. It is important to note here, Spark does not load the entire data set until it has to. I used the `show` command to print out the first five rows of the dataframe. I also used the `type` command to make sure the data read in as a `pyspark.sql.dataframe.Dataframe`.

```
# Reading my StackOverflow data from Google bucket into the dataframe
# entitled "df":
df = spark.read.csv("gs://dataproc-staging-us-central1-
        330795877686-cs3qbyk4/Lacey_Conrad_FinalQuery.csv",
                                        mode="DROPMALFORMED",
                                        inferSchema=True,
                                        header=True)

# Determining the data type of the dataframe "df":
type(df)

# Displaying the first 5 lines of the "df" dataframe:
df.show(5)
```

Table 2: The output of running `type(df)`, indicating its datatype.

`type(df)`
pyspark.sql.dataframe.DataFrame

Table 3: Using the command `show()` to print the first 5 rows from the df dataframe.

`df.show(5)`	
Id	Tags
30246138	<html><webvtt>
30246145	<html><css>
30246147	<javascript><jque...
30246148	<c#><windows-phon...
30246149	<vb.net>

One of the other advantages to using the `show` command was that I could see how my data were stored. The tags clearly are separated from each other by brackets, and not white space. These brackets will need to be stripped away in order to fully use the `split` function later, and to make sure extra tags weren't created because they still have a bracket connected to the tag (basically, I wanted to make sure there weren't "python" and "python>" tags. These would have been two separate tags without text cleaning). After some research, it appeared that `regexp_replace` from the `pyspark.sql.functions` library uses regex to find specific characters and replace them with whatever output is desired. I called this function on the "Tags" column, and had it replace any brackets ("<" or ">") with a space (" "). Adding the space was important, as this would once again facilitate the use to the `split` function later in the program. To check that this method worked, I once again used the `show` command to inspect the first five elements of the dataframe.

```
# Using regexp_replace to parse through the "Tags" column and replace
# any instance of a bracket (<,>) with a space:
df2 = df.withColumn('Tags', regexp_replace(df['Tags'], '[<>]', ' '))

# Using the command show to inspect the outcome of the previous step:
df2.show(5)
df2.select("Tags").show()
type(df2)
```

Table 4: The first 5 rows of the df2 dataframe after the removal of unwanted brackets.

`df2.show(5)`	
Id	Tags
30246138	html webvtt
30246145	html css
30246147	javascript jque...
30246148	c# windows-phon...
30246149	vb.net

Table 5: Display the contents of the Tags column after the removal of brackets.

`df2.select("Tags").show()`
Tags
html webvtt |
html css
javascript jque...
c# windows-phon...
vb.net
javascript jque...
c# json linq-t...
python profilin...
jquery
php ajax json ...
java api rest ...
pandas
google-oauth
python python-3...
c# sql-server
...

Table 6: Checking the datatype of the df2 dataframe.

`type(df2)`
pyspark.sql.dataframe.DataFrame

Since the goal of this inverted index was to make an index based off of the tags in the data I collected, I wanted to make sure there were tags in each row. I wrote my SQL query in such a way that the data I downloaded shouldn't have any rows with blank tags, and I was fairly confident that all rows had at least one tag. However, I wanted to be certain there weren't any rows with missing tags so I also performed some data cleaning using `regexp_replace` replacements. To verify I didn't create any empty tags, I used the lambda function described in our project handout to create a resilient distributed dataset (RDD) that returned an RDD object with no empty "Tags" rows. This function simply created an RDD that used the `filter` command to only return rows where, essentially, `'Tags' != None`.

Lambda functions are used in a few places in this project. These are small anonymous functions that can take any number of arguments, but can only have one expression. They have the form of `lambda` *`arguments`* `:` `expression`.

For example, the following function multiples two numbers, both of which are passed in as argumentsc ("Python lambda," 2020):

```
# a lambda function that multiples two numbers:
x = lambda a, b : a * b

# passing in two arguments to the lambda function:
print(x(5, 10))

# The result of running this function with 5 and 10
# will be 50.
```

The following is the code that creates an RDD out of the df2 dataframe. This RDD includes only data filtered using a lambda function, which returns the rows where the Tags column is not empty:

```
# The following creates the blank_tags variable which uses a lambda function
# to parse through the df2 dataframe and only return rows where the
# Tags column is not empty:
blank_tags = df2.rdd.filter(lambda x: x.Tags is not None)

# Since the blank_tags function should also convert the dataframe into an RDD,
# I checked the datatype of blank_tags:
type(blank_tags)
```

Table 7: Checking the datatype of the blank_tags variable to verify that it is now an RDD.

`type(blank_Tags)`
pyspark.rdd.PipelinedRDD

Next, I would recreate the `split_strip` function used in the project handout. This function takes in a string and returns a list of a list of tags with their associated PostId. To do this, it first strips the string of any white space around the string, and then it splits the string into its constituent words/tags using the split function. The output is then a nested list ([’tag’,’tag’...], PostId). For example, ([’html’,”, u’css’], 30246145). I then ran my RDD through this function, and observed the results using `take`:

```
# This function returns a tuple of the form ([Tag(s)], Post ID).  It accomplishes
# this by taking in a string of data and stripping white spaces around the
# words (=tags) and then splitting them on white space.  This will create
# a list out of all the tags for a given PostId.
def split_strip(x):
        return (x.Tags.strip().split(' '), x.Id)

# Creating a variable "strip" and running the RDD
# blank_tags through the split_strip function.
strip = blank_tags.map(split_strip)

# Inspecting the results of the split_strip function:
strip.take(5)
```

Table 8: Printing the output of the split_strip function on the blank_tags RDD.

`strip.take(5)`
([u'html', u'', u'webvtt'], 30246138), ([u'html', u'', u'css'], 30246145), ([u'javascript', u'', u'jquery', u'', u'ajax', u'', u'get', u'', u'ip'], 30246147), ([u'c#', u'', u'windows-phone-8.1'], 30246148), ([u'vb.net'], 30246149)

Since I now had a nested list of tags that were associated with a given PostId, I needed to convert this list into a new list where each tag was individually linked in a key-value pair with its given PostId. For example, in table 8 we see that the the tags "html" and "webvtt" are both linked to PostId 30246138 in the form ([u'html', u'', u'webvtt'], 30246138). My next step needed to be creating a new list where each tag was linked individually to a PostId. In my example, I want to separate ([’html’, u’webvtt’], 30246138) into a list of (’html’, 30246138) and (’webvtt’, 30246138). To perform this task, I needed to "flatten" the list, which can be accomplished by using flatMap. flatMap will act like the map() function in a typical MapReduce scenario, however, it will also flatten in the manner described previously. Recall that mapping in MapReduce involved linking a key to a value. To facilitate this, I utilized a lambda function that iterates through each word in the list of tags of a given PostId (`for word in x[0]`), and links it with the PostId (`(word, x[1])`).

After producing a list of tags with their PostIds, it was possible to combine identical tags in a manner similar to the reduce step in MapReduce. Here, when the common keys were reduced, their corresponding PostIds were be added to the end of a list for that tag. We started with a list of the form (’tag1’, PostId1), (’tag1’, PostId2), (’tag1’, PostId3), and wanted to end up with (’tag1’, PostId1, PostId2,Post3). For example, in table 9 we notice that in our list, we have (’html’, 30246138) and (’html’, 30246145). The goal for the next step was to reduce this list down to one entry of the form (’html’, 30246138, 30246145). Before this final step, however, our data will not have the actual PostIds listed, but instead will look like <pyspark.resultiterable.ResultIterable at 0x7fa81adba850> instead of the PostIds. This was corrected for in the following step.

```
# The following is using the flatMap function to assign the relevant
# PostId to each tag associate with it.  It uses a lambda function
# in order to iterate through each word in the list of a given Post Id
# and linking it to its PostId.
word_to_id = strip.flatMap(lambda x: [(word, x[1]) for word in x[0]])

# Printing out the first 10 lines of word_to_id to verify
# correct execution of the flatMap and lambda functions:
word_to_id.take(10)

# Using groupByKey to produce a list of tuples of the form
# (tag, PostId) by reducing the list created in word_to_id:
grouped_by_word = word_to_id.groupByKey()

# Printing out the first 10 lines of grouped_by_word to verify corrected
# execution of the groupByKey command:
grouped_by_word.take(10)
```

Table 9: The first 10 lines of output after applying the `word_to_id` function on the `strip` RDD. This produces a list of tags (including duplicate tags) and their linked PostId.

`word_to_id.take(10)`
(u'html', 30246138),
(u'', 30246138),
(u'webvtt', 30246138),
(u'html', 30246145),
(u'', 30246145),
(u'css', 30246145),
(u'javascript', 30246147),
(u'', 30246147),
(u'jquery', 30246147),
(u'', 30246147)

Table 10: The output of applying the `grouped_by_word` function on `word_to_id`. This condenses duplicate versions of a tag done to one version, with a list of associated PostIds.

`grouped_by_word.take(10)`
(u'', <pyspark.resultiterable.ResultIterable at 0x7fa81adba850>),
(u'h.265', <pyspark.resultiterable.ResultIterable at 0x7fa81ad74450>),
(u'h.264', <pyspark.resultiterable.ResultIterable at 0x7fa81ad74590>),
(u'biopython', <pyspark.resultiterable.ResultIterable at 0x7fa81ad745d0>),
(u'screen-resolution',
<pyspark.resultiterable.ResultIterable at 0x7fa81ad74610>),
(u'userscripts', <pyspark.resultiterable.ResultIterable at 0x7fa81ad74650>),
(u'prefix', <pyspark.resultiterable.ResultIterable at 0x7fa81ad74690>),
(u'tcp-ip', <pyspark.resultiterable.ResultIterable at 0x7fa81ad746d0>),
(u'netcdf', <pyspark.resultiterable.ResultIterable at 0x7fa81ad74710>),
(u'sublime-text-plugin',
<pyspark.resultiterable.ResultIterable at 0x7fa81ad74750>)

To convert the list back into the form tag, PostId (instead of the resultiterable), I used python dictionary comprehension using the map function in addition to a lambda function which converted the second item in the tuple back into a list of PostIds(`list(x[1])`).

```
# Using the map function and a lambda function to convert the
# resultiterable datatype back into a list of PostIds:
inverse_index = grouped_by_word.map(lambda x: {x[0]: list(x[1])})

# Printing the first 10 items of the inverse_index to verify
# the correct conversion to a list of PostIds:
inverse_index.take(10)
```

The final step in the creation of my inverted index was to save the index to my google bucket. I did this in the following manner:

```
client = storage.Client()
inverse_index.saveAsTextFile('dataproc-staging-us-central1-330795877686-
        cs3qbyk4/inverted_index2')
```

3 Results and Output

The first 10 lines of the inverted index created from StackOverflow Tags with linked PostIds during the month of May, 2015, is shown below:

Table 11: The inverted index of Tags and their linked PostIds created from StackOverflow data collected for the month of May, 2015.

`inverse_index.take(10)`
{u”: [30246138, 30246145, 30246147, 30246147, ...]}, {u’h.265’: [30431535]}, {u’h.264’: [30126921, 30431535, 30209602, 30345495, 30190321, 30352670, 30360651, 30365138]}, {u’biopython’: [30181545, 30340573, 30054745, 30065446, 30192495, 30352885, 30363769]}, {u’screen-resolution’: [30270810, 30345858]}, {u’userscripts’: [30128372, 30206185]}, {u’prefix’: [30126737, 30428581, 30134705, 30187513, 30196853]}, {u’tcp-ip’: [30423859, 30424758, 30427647, 30275901, 30351258, 30361950]}, {u’netcdf’: [30179586, 30193089, 30350814]}, {u’sublime-text-plugin’: [30428292, 30002658, 30050378, 30065807]}]

From table 11, we can see the first 10 items in the inverted index. Each item in the index is listed in the form of ’Tag’: [PostId 1, PostId 2, ...]. In order to find what tags are in the index, simply look at the first item in the tuple, for example the tags ’screen-resolution,’ ’biopython,’ and ’userscripts.’ In a list next to each tag we see the PostIds of the Posts linked to that tag. For example, for the tag ’screen-resolution’ we see the numbers 30270810 and 30345858, which are the two posts associated with this tag during the month of May, 2015. The tags can have any number of PostIds, from one and over, but they must have at least 1 PostId or they would not have been included in the list. There is an unusual tag that I believe to be an empty tag: ”. This tag seemed to be the most frequently encountered tag, and in some cases it appears to have been found more than one time in some posts. I will discuss what could have created this tag in the Analysis section

4 Analysis

I am unsure what created the empty tag, but it could be related to the manner in which the regexp_replace replaced brackets (< >) with white spaces. It is possible that I may have created a white space tag from areas where there might have been an extra space at the beginning or end of a tag, for example, ’< python>’. I also believe the way

the split_strip function works may be to strip the white space from the beginning and end of the string, *not* the word/tag. In my code, I am splitting on space ' ', which could create conditions for creating empty tags. If the original tags looked like < Python>< Java>, removing the brackets would create ' Python Java.' Then, splitting on space would look like '', 'Python', '', 'Java.'

5 Conclusion

Search engines are one technological convenience most people no longer think about. After all, we can search the entire internet by simply navigating to our favorite engine and entering a search term. Doing a quick Google search on how many websites exist in the world indicates that there are nearly 2 billion websites (specifically, in January of 2018, there were 1,805,260,010 websites) (Soni, 2019). The idea of 2 billion websites being scoured every time anyone uses a search engine clearly is silly, but then, how else would we get a list of results after our search? Enter the inverted index. Linking a users search query to an index that already exists cuts down on the processing time and resources required to search every web page.

Many fascinating topics emerge from the implementations of inverted indexes. As Google is synonymous with web searching (so much so that web searching is referred to a "Googling"), I decided to read a bit more into Google's history with using inverted indexes. Clearly, Google has sought year after year to improve their searches. Initially, it would take some time for new content added to the internet to make it into the Google indexes. More recently, Google has been creating updates that speed up indexing of content on the web. In a patent connected to Googles search index, they discuss 2 obstacles in providing "fresh" results:

1. There is a great deal of expense/overhead associated with rebuilding the document index each time the document repository is updated. They point out an example where small indexes are created from new or updated content, which are then merged to the larger index. This process can be slow (although they never say on what scale of time) (Slawski, 2020).

2. The necessity to process queries while concurrently rebuilding the document index. There is a need to synchronize the threads that execute queries with the threads that update the document repository. This can hamper efficient operation of the document repository if updates are performed frequently. Thus, "freshening" the data can actually slow down the performance of the search engine (Slawski, 2020).

The second point is very fascinating to me. That by trying to keep the index up to date, they can actually reduce performance of the search engine. Also, freshness does not simply refer to adding new content into the index; documents can be removed from the index as well, in a method called "treadmilling." In the inverted index, only the data at the front of the index can be deleted. Therefore, to removed unwanted documents, the documents are copied to the back end, and the entire data set is shifted forward one step. During this process, the unwanted document can be removed, saving the memory otherwise occupied by unwanted data (Slawski, 2020).

The creation of my inverted index was also very fascinating. Each step in the process was very clear, and it was surprisingly simple to create the index. This seems, in part, to be due to the number of built in functions and methods Spark has. With data collection, for example, you can either write a query to pull the specific subset of data you want, or you can use `filter` to specify what subset of data you want to use. Further on, we utilize the `flatMap` method to distribute the PostIds of a list of tags, to each tag (note: we used `flatMap` alongside a lambda function). Finally, we took advantage of the `group_by_key` method, which condenses common keys while collecting associated PostIds.

Something I only now realized is how often data scientists and data engineers work with text. Although I knew that data scientists did look at topics like sentiment analysis, I still figured most of a data scientists work came down to analytics and numbers. The "internet of things," is a very apt statement, since it seems virtually anything can be utilized numerous ways as data. This makes me realize even more how important data scientists and engineers are, and how much more data existed in the world than I thought originally.

6 References

1. *Inverted index.* (2005, November 10). Wikipedia, the free encyclopedia. Retrieved June 24, 2020, from `https://en.wikipedia.org/wiki/Inverted_index`

2. Jain, S. (2019, September 25). *Inverted index*. GeeksforGeeks. https://www.geeksforgeeks.org/inverted-index/

3. McDonald, C. (2018, October 17). *Spark 101: What is it, what it does, and why it matters*. mapr.com. https://mapr.com/blog/spark-101-what-it-what-it-does-and-why-it-matters/

4. *Python lambda*. (2020). W3Schools Online Web Tutorials. https://www.w3schools.com/python/python_lambda.asp

5. Slawski, B. (2020, May 31). *Caffeine: Google's indexer*. Go Fish Digital. https://gofishdigital.com/googles-indexer-caffeine/

6. Soni, R. (2019, February 15). *How many websites exist in the world today?* Quora - A place to share knowledge and better understand the world. https://www.quora.com/How-many-websites-exist-in-the-world-today