2. Hadoop and Spark Setup

2.1 Hadoop Setup

Command:

hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar grep input output 'dfs[a-z.]+'

```
[hadoop@ip-172-31-56-22 hadoop]$ bin/hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar grep input output 'dfs[a-z.]+'
17/04/03 18:51:19 INFO impl.TimelineClientImpl: Timeline service address: http://ip-172-31-56-22.ec2.internal:8188/ws/v1/timeline/
17/04/03 18:51:19 INFO client.RMProxy: Connecting to ResourceManager at ip-172-31-56-22.ec2.internal/172.31.56.22:8032
17/04/03 18:51:20 INFO input.FileInputFormat: Total input paths to process : 8
17/04/03 18:51:20 INFO lzo.GPLNativeCodeLoader: Loaded native gpl library
17/04/03 18:51:20 INFO lzo.LzoCodec: Successfully loaded & initialized native-lzo library [hadoop-lzo rev 30eccced8ce8c483445f0aa3175ce725831ff06b]
17/04/03 18:51:20 INFO mapreduce.JobSubmitter: number of splits:8
17/04/03 18:51:20 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1491245129359_0001
17/04/03 18:51:20 INFO impl.YarnClientImpl: Submitted application application_1491245129359_0001
17/04/03 18:51:21 INFO mapreduce.Job: The url to track the job: http://ip-172-31-56-22.ec2.internal:20888/proxy/application_1491245129359_0001/
17/04/03 18:51:21 INFO mapreduce.Job: Running job: job_1491245129359_0001
17/04/03 18:51:29 INFO mapreduce.Job: Job job_1491245129359_0001 running in uber mode : false
17/04/03 18:51:29 INFO mapreduce.Job:  map 0% reduce 0%
17/04/03 18:51:37 INFO mapreduce.Job:  map 50% reduce 0%
17/04/03 18:51:40 INFO mapreduce.Job:  map 75% reduce 0%
17/04/03 18:51:41 INFO mapreduce.Job:  map 100% reduce 0%
17/04/03 18:51:44 INFO mapreduce.Job:  map 100% reduce 9%
17/04/03 18:51:46 INFO mapreduce.Job:  map 100% reduce 17%
17/04/03 18:51:47 INFO mapreduce.Job:  map 100% reduce 26%
17/04/03 18:51:48 INFO mapreduce.Job:  map 100% reduce 60%
17/04/03 18:51:49 INFO mapreduce.Job:  map 100% reduce 94%
17/04/03 18:51:50 INFO mapreduce.Job:  map 100% reduce 100%
17/04/03 18:51:51 INFO mapreduce.Job: Job job_1491245129359_0001 completed successfully
17/04/03 18:51:51 INFO mapreduce.Job: Counters: 51
        File System Counters
                FILE: Number of bytes read=1377
                FILE: Number of bytes written=5489617
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=37515
                HDFS: Number of bytes written=3864
                HDFS: Number of read operations=129
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=70
        Job Counters
                Killed map tasks=1
                Launched map tasks=8
                Launched reduce tasks=35
                Data-local map tasks=5
                Rack-local map tasks=3
                Total time spent by all maps in occupied slots (ms)=2635470
```

```
[hadoop@ip-172-31-56-22 hadoop]$ hdfs dfs -cat output/part-r-00000
2       dfs.hosts.exclude
2       dfs.datanode.available
1       dfs.namenode.https
1       dfsadmin
1       dfs.namenode.plugins
1       dfs.datanode.max.xcievers
1       dfs.permissions.superusergroup
1       dfs.namenode.handler.count
1       dfs.datanode.plugins
1       dfs.replication
1       dfs.data.dir
1       dfs.namenode.safemode.extension
1       dfs.encryption.key.provider.uri
1       dfs.server.datanode.fsdataset.
1       dfs.datanode.du.reserved
1       dfs.namenode.name.dir
1       dfs.name.dir
1       dfs.webhdfs.enabled
1       dfs.namenode.http
1       dfs.datanode.fsdataset.volume.choosing.policy
1       dfs.datanode.data.dir
1       dfs.namenode.rpc
```

Explanation:

This command extracts strings that match a regular expression from input files and counts how many times they occurred, sorts matching strings by their frequency and writes the output in output file.

The argument input identifies input directory path and output identifies output directory path. Argument 'dfs[a-z.]+' is a regular expression that matches the character sequence ‘dfs’ (case sensitive) followed by any characters in the range between a and z and ‘.’ one or more times.

The program runs map and reduce jobs in sequence. Map job takes a line as input and matches the regex against the line, with output as (matching string, 1). Reducer job takes the output from mapper and sums the frequency of each matching string and the output is (matching string, frequency).

2.2 Spark Setup

Command:

```
>>> rdd=sc.parallelize(range(10000000))
>>> rdd.takeSample(False,10)
[Stage 0:>                                                  (0 + 0) / 4]17/04/03 19:03:12 WARN TaskSetManager: Stage 0 contains a t
ask of very large size (12130 KB). The maximum recommended task size is 100 KB.
17/04/03 19:03:13 WARN TaskSetManager: Stage 1 contains a task of very large size (12130 KB). The maximum recommended task size is 100 KB.
[8643435, 6704656, 2380813, 6520879, 7053948, 4182820, 7147427, 7767040, 1886445, 6780719]
>>>
```

Explanation:

sc means SparkContext, which tells Spark how and where to access a cluster. We should use SparkContext to create RDDs. We don’t have to initialize it in pyspark shell because it automatically create SparkContext for us.

3. MapReduce Programming for Finding Degree Distribution in Graphs

Undirected graph

3.1

The mapper takes each line (nodeX -- nodeY) as input and produces output as (nodeX, 1) and (nodeY, 1). The reducer takes the output from mapper and sums the frequency based on key. The output would be (node, frequency).

3.2

Mapper output:

(1, 1) (2, 1)
(1, 1) (3, 1)
(3, 1) (4, 1)
(1, 1) (4, 1)
(9, 1) (7, 1)
(5, 1) (6, 1)
(7, 1) (5, 1)
(7, 1) (6, 1)

(1, 1) (9, 1)
(8, 1) (9, 1)
Reducer output:
(1,4)
(2,1)
(3,2)
(4,2)
(5,2)
(6,2)
(7,3)
(8,1)
(9,3)

Directed graph
3.3
The mapper takes each line (nodeX → nodeY) as input and produces output as (nodeY, 1). The reducer takes (nodeY, 1) as input and sums the frequency based on nodeY, the output is (nodeY, frequency).

3.4
Mapper output:
(2,1)
(3,1)
(4,1)
(4,1)
(7,1)
(6,1)
(5,1)
(6,1)
(9,1)
(9,1)

Reducer output:
(2,1)
(3,1)
(4,2)
(5,1)
(6,2)
(7,1)
(9,2)

4. Computing Degree Distribution in Hadoop
4.1

```
[hadoop@ip-172-31-24-108 hadoop]$ hdfs dfs -moveFromLocal /localinput/com-friendster.ungraph.txt input
[hadoop@ip-172-31-24-108 hadoop]$ hdfs dfs -ls /input
ls: `/input': No such file or directory
[hadoop@ip-172-31-24-108 hadoop]$ hdfs dfs -ls input
Found 1 items
-rw-r--r--   2 hadoop hadoop 32364651652 2017-04-04 13:43 input/com-friendster.ungraph.txt
[hadoop@ip-172-31-24-108 hadoop]$
```

4.2

Both moveFromLocal and put commands copy files from local file system to HDFS. The difference is that put command copies the file and keeps the original one while moveFromLocal command moves the file to HDFS and deletes the original one.

4.3

See code at BlackBoard.

4.4

It sorted the id of nodes (key) in ascending order.

4.5

See code at BlackBoard.

4.6

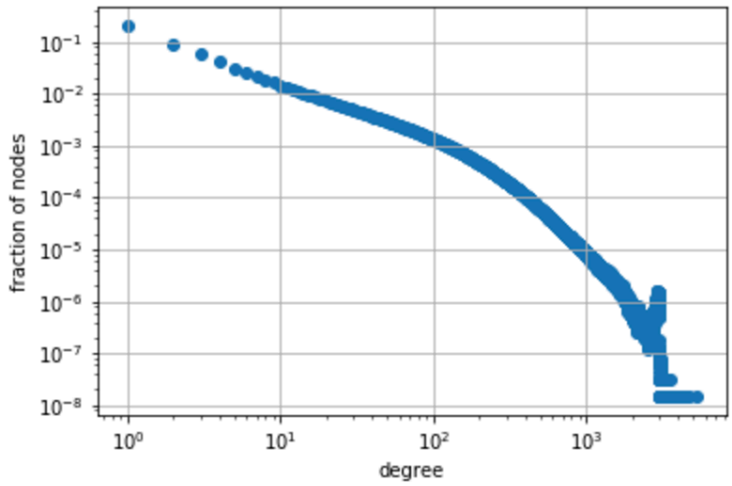
```
71768986,5214
54083823,4667
54118654,4488
89750075,4406
54085211,4333
54118345,4214
7688909,4006
19713658,3897
54086114,3891
10278129,3829
106212071,3722
7123984,3707
45360334,3570
6879541,3562
20366275,3561
45032137,3553
25849922,3553
34557675,3544
33529861,3525
59858810,3507
41937494,3458
76883100,3440
57394902,3430
13701967,3426
81272200,3423
28596706,3392
52736481,3377
59033707,3346
52736401,3336
113966763,3316
44632351,3316
49787738,3306
63715953,3289
6821999,3286
3653464,3286
31742320,3280
```

```
29364051,3279
25722722,3247
85554090,3235
37384478,3225
24605639,3224
79188771,3223
20943522,3219
46191989,3217
35364707,3214
12628503,3208
32892161,3206
8046960,3197
35393926,3196
27830575,3192
44848860,3189
25265623,3183
23382015,3180
11304056,3176
12132865,3176
68344538,3171
18419444,3171
37491512,3170
36291316,3166
73976135,3164
37872400,3159
52840255,3156
14008201,3150
24307738,3141
64782691,3141
36542244,3132
62168417,3129
16324321,3127
76056267,3127
37748346,3124
63915702,3121
33961320,3120

30139434,3119
38906012,3117
94826493,3116
35018189,3115
76358513,3111
61429407,3111
46598206,3109
36947798,3105
90608504,3103
27089370,3102
26388641,3102
16209345,3099
49203749,3097
35816339,3096
39171857,3094
76461293,3094
45338224,3094
43571592,3094
37883265,3093
40203014,3092
65925160,3091
55704705,3090
44014655,3089
13085298,3088
66911524,3084
117227284,3084
21594966,3084
6261933,3081
```

4.7

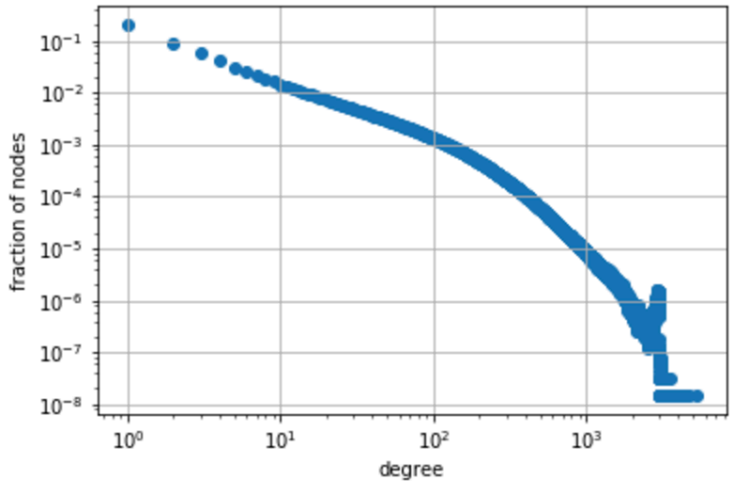
Plot:



5. Computing Degree Distribution in Spark

See code at BlackBoard.

Plot:



6. About Hadoop and Spark

6.1 Explain the concept of lazy evaluation employed by Apache Spark. How does it allow Spark to optimize the execution of code?

Lazy evaluation means Spark remembers the operation we tell it to do on the dataset, but it absolutely evaluates nothing until we take an action operation.

This allows Spark to optimize the memory and disk usage. Spark will understand the entire instruction chain as a whole, figure out the final answer it needs to evaluate and how much work it has to do, and "does the right thing" based on the information. However, if Spark does evaluation step by step explicitly as the instructions we give, it will hold a lot of intermediate datasets and results in memory which will increase GC costs.

6.2 Explain how you can pin an RDD in main memory in Spark using an RDD function. Describe a simple programming use case where such explicit pinning of an RDD in main memory will improve runtime execution.

We can use rdd.cache() to store the computed result of rdd in memory thus we don't need to re-compute it every time. For example, if we want to do sentiment analysis where the label of dataset is either positive or negative. Then after we load the dataset using sc.textFile() and split the line by flatMap(), we are going to count the number of positive instances and that of negative instances. If we do not cache, Spark will reload the data for both posCount and negCount operation. However, if we do cache after RDD is created in flatMap step, the processing done previously will be stored and reused.

6.3 During a Spark program execution, one of the slaves died and partitions of RDDs stored on the slave were lost. Will Spark re-compute the entire RDD or only the partitions? Explain briefly how Spark will carry out this RDD recovery.

Spark will re-compute only the partitions. RDDs are designed to recover from worker failures as they have a long lineage graph and can regain their state by re-computing lost blocks using the prior information based on lineage.

Also, in case of slave failure, if the executor fails it will be re-launched automatically; if the receiver fails it will re-started by the worker and actually the data received by receiver is replicated on another slave.

6.4 Give a simple example of a summary statistic that is difficult to compute using Apache Hadoop MapReduce programming but can be easily computed using Apache Spark. (Hint: MapReduce programming is limited to map and reduce functions, while Spark incorporates many other functions that are calculated over all elements in the RDD.)

For example, we have a sentiment dataset with twitter, sentiment, hashtag as attributes and we want to calculate the difference between the number of positive instances and that of negative ones on each hashtag.

References:

http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/
http://stackoverflow.com/questions/28981359/why-do-we-need-to-call-cache-or-persist-on-a-rdd
https://www.sigmoid.com/fault-tolerant-stream-processing/