Haoran Li, Yang Jin haoranli830 457929, 458848

Homework9

Problem 1: Transform Data using SPARK (15%)

(a) Using the command below to create a RDD holding html records.

```
>>> data2 = sc.textFile('/loudacre/weblogs').filter(lambda line: line.find('html
') != -1)
```

Transform the data into the required form.

```
>>> html = data2.map(lambda line: line.split()).map(lambda line: line[0] + '/'+l
ine[2])
```

Then display top 10 of them.

```
>>> for i in html.take(10):
...     print(i)
...
142.19.184.108/187
66.72.212.11/91
194.91.6.192/18641
252.232.52.250/67809
61.211.36.7/123
8.135.236.171/10022
138.229.167.20/34
153.100.5.159/71
46.250.3.164/70
16.52.46.209/1055
```

(b) There are 1079891 records in the weblogs data.

```
>>> data = sc.textFile("/loudacre/weblogs")
>>> data.count()
1079891
```

Among them, there are 474360 HTML requests in it.

We perform filter on the whole dataset and filter out those have ‘html’ in it.

```
>>> filtered = sc.textFile("/loudacre/weblogs").filter(lambda line: line.find('h
tml') != -1)
>>> filtered.count()
474360
```

Problem 2: Joining Datasets with SPARK (40%)

(a) There are 1079891 records in the weblogs data.

```
>>> data = sc.textFile("/loudacre/weblogs")
>>> data.count()
1079891
```

Use filter to find those data contains ‘html’. There are 474360 HTML requests

Command:

filtered = sc.textFile("/loudacre/weblogs").filter(lambda line: line.find('html') != -1)

```
>>> filtered = sc.textFile("/loudacre/weblogs").filter(lambda line: line.find('h
tml') != -1)
>>> filtered.count()
474360
```

(b) Among those data contains ‘html’, we split them by ‘ ’ and map them into pairs of the form (userid, 1). Then perform reduce to get the sum of each user ID.

Command:

filtered = sc.textFile("/loudacre/weblogs").filter(lambda line: line.find('html') != -1)

pair = filtered.map(lambda line: line.split()).map(lambda line: (line[2], 1))

sum = pair.reduceByKey(lambda v1, v2: v1 + v2)

```
>>> for a in sum.take(10):
...     print(a)
...
(u'127716', 4)
(u'59557', 3)
(u'38059', 1)
(u'99252', 1)
(u'108342', 2)
(u'43270', 3)
(u'87287', 4)
(u'57149', 30)
(u'63499', 2)
(u'101407', 27)
```

(c) Perform filter on the RDD we just got. There are 5831 users visited once, 1460 users visited 7 times and 635 users visited 12 times.

Command:

```
>>> one = sum.filter(lambda a : a[1] == 1)
>>> one.count()
5831
>>> seven = sum.filter(lambda a : a[1] == 7)
>>> seven.count()
1460
>>> twelve = sum.filter(lambda a : a[1] == 12)
>>> twelve.count()
635 _
```

(d) We use map to get the form we want:

Command:

```
>>> data = sc.textFile("/loudacre/accounts")
>>> data.count()
129761
```

```
>>> pair = data.map(lambda a : a.split(",")).map(lambda a: (a[0], a))
>>> pair.take(5)
[(u'1', [u'1', u'2008-10-23 16:05:05.0', u'\\N', u'Donald', u'Becton', u'2275 Wa
shburn Street', u'Oakland', u'CA', u'94660', u'5100032418', u'2014-03-18 13:29:4
7.0', u'2014-03-18 13:29:47.0']), (u'2', [u'2', u'2008-11-12 03:00:01.0', u'\\N'
, u'Donna', u'Jones', u'3885 Elliott Street', u'San Francisco', u'CA', u'94171',
 u'4150835799', u'2014-03-18 13:29:47.0', u'2014-03-18 13:29:47.0']), (u'3', [u'
3', u'2008-12-21 09:19:50.0', u'\\N', u'Dorthy', u'Chalmers', u'4073 Whaley Lane
', u'San Mateo', u'CA', u'94479', u'6506877757', u'2014-03-18 13:29:47.0', u'201
4-03-18 13:29:47.0']), (u'4', [u'4', u'2008-11-28 00:08:09.0', u'\\N', u'Leila',
 u'Spencer', u'1447 Ross Street', u'San Mateo', u'CA', u'94444', u'6503198619',
u'2014-03-18 13:29:47.0', u'2014-03-18 13:29:47.0']), (u'5', [u'5', u'2008-11-15
 23:06:06.0', u'\\N', u'Anita', u'Laughlin', u'2767 Hill Street', u'Richmond', u
'CA', u'94872', u'5107754354', u'2014-03-18 13:29:47.0', u'2014-03-18 13:29:47.0
']))]
```

(e) First we tried to join them directly, but it seems exceeds the memory limit of my VM. So I first perform groupByKey on the RDD pair to repartition it, then join with the sum pair. But after repartition, the list in pair becomes a ResultIterable object.
Command:

```
>>> joined = pair.groupByKey().join(sum)
>>> joined.take(5)
[(u'102667', (<pyspark.resultiterable.ResultIterable object at 0x29060d0>, 7)),
(u'41471', (<pyspark.resultiterable.ResultIterable object at 0x2906190>, 16)), (
u'28305', (<pyspark.resultiterable.ResultIterable object at 0x29066d0>, 4)), (u'
95082', (<pyspark.resultiterable.ResultIterable object at 0x2906710>, 12)), (u'4
3572', (<pyspark.resultiterable.ResultIterable object at 0x2906750>, 1))]
```

So, we performed following operations to convert it back to a list so that we can choose the column we want to output.
Command:

```
>>> newJoined = joined.map(lambda a : (a[0], (str(list(a[1][0])[0]).split(","),
a[1][1])))
>>> newJoined.take(3)
[(u'102667', (["[u'102667'", " u'2013-07-04 15:32:55.0'", " u'2014-01-14 13:40:4
4.0'", " u'Louise'", " u'Johnson'", " u'4545 Snyder Avenue'", " u'Kingman'", " u
'AZ'", " u'86472'", " u'9286040368'", " u'2014-03-18 13:33:01.0'", " u'2014-03-1
8 13:33:01.0']"], 7)), (u'41471', (["[u'41471'", " u'2012-07-14 08:43:03.0'", "
u'2013-11-22 03:42:18.0'", " u'Debbie'", " u'Patterson'", " u'4981 Ottis Street'
", " u'Bakersfield'", " u'CA'", " u'93390'", " u'6618428443'", " u'2014-03-18 13
:31:03.0'", " u'2014-03-18 13:31:03.0']"], 16)), (u'28305', (["[u'28305'", " u'2
011-12-25 15:25:25.0'", " u'2014-01-31 08:04:51.0'", " u'William'", " u'Hughes'"
, " u'254 Mercer Street'", " u'Fresno'", " u'CA'", " u'93775'", " u'5599181388'"
, " u'2014-03-18 13:30:38.0'", " u'2014-03-18 13:30:38.0']"], 4))]
```

The result is:

```
>>> result = newJoined.map(lambda a : a[0] + ' ' + str(a[1][1]) + ' ' + a[1][0][
3] + ' '  + a[1][0][4])
>>> for a in result.take(5):
...     print(a)
...
102667 7  u'Louise'  u'Johnson'
41471 16  u'Debbie'  u'Patterson'
28305 4  u'William'  u'Hughes'
95082 12  u'Dominique'  u'Pepper'
43572 1  u'Philip'  u'McHale'
```

Problem 3: Write a SPARK Program (30%)

(a) See Python codes under wustlkey haoranli830.

(b) Run SPARK application on the weblogs data locally.
Command:

```
$ spark-submit --master local CountJPGs.py file:/home/cloudera/training_materials/dev1/data/weblogs
```

The number of JPG requests：

```
('counts: ', 64978)
```

In local mode, the driver runs locally. The process is created inside a single JVM. The result is distributed across the executors until an action returns a value to driver. The result is not stored in HDFS.

(c) Run your SPARK application on the weblogs data on the cluster using client mode.
Command:

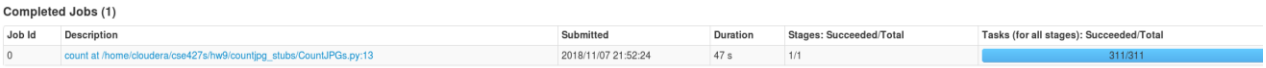
```
$ spark-submit --master yarn-client CountJPGs.py /loudacre/weblogs
```

The number of JPG requests：

```
('counts: ', 64978)
```

In client mode, the driver runs in the client process. Processing happens on the executors in the node managers of the YARN cluster. The result is serialized and sent to the executor backend, and then back to the driver as a status update message. The result is not stored in HDFS.

(d) Stages:1 Tasks:311



Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	count at /home/cloudera/cse427s/hw9/countjpg_stubs/CountJPGs.py:13	2018/11/07 21:52:24	47 s	1/1	311/311

(e) Run the job in cluster mode.
Command:

```
$ spark-submit --master yarn-cluster CountJPGs.py /loudacre/weblogs
```

The number of JPG requests：

```
('counts: ', 64978)
```

In cluster mode, the Spark driver runs inside an application master process which is managed by YARN on the cluster. The processing happens on executors on node managers on clusters. The result is serialized and sent to the executor backend, and then back to the driver as a status update message. The result is not stored in HDFS.

Problem 4: PageRank Data Representation (15%)

(a) Raw input data:
(a – a), (a – b), (a – c),
(b – a), (b – c),
(c – c), (c – b).
Links:
(a – <a, b, c>), (b – <a, c>), (c – <c, b>).

(b) The links representation is more efficient. For a general webgraph with n pages and m links, the raw input representation storage is 2*m. While the link representation storage is n+m. So for the case that that number of links is much larger than number of pages, links representation is more efficient. If there are fewer links than pages, the raw input representation is more efficient.