Data Science **November 10 , 2017**

Big Data

Exercises sheet n° : algorithms and programming in MapReduce

Exercise 1. Design a variant of the WordCount algorithm where the Map function performs local aggregation. The idea is to have a Map that emits couples of the kind (w, k) where k>=1 is the number of w occurrences in the input value.

Solution :

https://www.dropbox.com/sh/srxng13y2ddxkyw/AACH9oGiTZscDBDPvL56jlQOa?dl=0

Do you see any problem with this approach?

The problem is that local aggregation is performed only at the line level, and not at the split level.

Is the combiner still needed?

Yes.

Can Reduce be considered as a combiner as it is?

Yes, counting is an associative operation.

Observe the result of the Job, do you see any problems due to data cleaning issues?

We have several characters (e.g., punctuation) that make two same words almost the same.

So the idea is to remove from word everything that is not an alpha-numeric character.

This is implemented here

https://www.dropbox.com/sh/xnug48n26ewbc5b/AAASlbLzajx2Wu6Hgb7qTpI-a?dl=0

Also, local aggregation is here made at the split level, hence the combiner is not yet needed. But the dictionary may exceed the available memory for the Mapper, so the approach is not ensured to scale. In order to solve this problem, the code needs to be modified in order to emit the current content of the dictionary periodically (i.e., when a counter exceeds a given threshold).

Exercise 2. Design a MapReduce job without Combine that takes as input a file where each line contains a string indicating a Web URL and a natural number indicating the amount of time (in seconds) a user has spent on that Web page during a visit session. Of course you can have multiple lines for the same URL. The job is expected to return the list of unique URLs together with the average visit time.

Solution

https://www.dropbox.com/sh/o3787vkj55ophij/AACrJffWk94cNEuetCvm-JEVa?dl=0

Can the Reduce be considered as a Combiner?

No, as the average is not associative.

If not, indicate the necessary modifications at the Map and Reduce level in order to have a Combine, and provide a definition of this last one.

Solution

https://www.dropbox.com/sh/gl6gx5o7119fl60/AADyOxvRm8KGMEPrPAhzmIExa?dl=0

The main idea is to have a Map that emits a pair (url, (1,1)) and a combiner that emits (url, (s,c)) where s is the local sum of times for url, while c is the number of times url has been met.

Exercise 3. Provide a Python encoding of the following algorithms dealt with in classes: WordCount+ (Exercise 1), WordCount solving data cleaning issues, and Average calculation (Exercice2). To this end, rely on Hadoop streaming seen in the class.

Solutions are provided above in Python.

Exercise 4. Encoding SQL queries in MapReduce. Consider the following simple relational schema containing informations about clients and orders they made.

Customer(cid, startDate, name)

Order(#cid, total)

Note that, for the sake of simplicity, we do not have any primary key for Order. Also assume that all the fields are mandatory.

Provide the Python MapReduce encoding of the following SQL queries.

- SELECT name FROM Customer WHERE month(startDate)=7

Solution:

https://www.dropbox.com/sh/xpzfulx9zwuqedi/AAAw7PzVSL2DxhhCh6nKQDUWa?dl=0

- SELECT DISTINCT name FROM Customer WHERE month(startDate)=7

Solution:

https://www.dropbox.com/sh/kphvikyanjm1pk6/AAAHXSN3yus_N8talXnSy6zFa?dl=0

- SELECT O.cid, SUM(total), COUNT(DISTINCT total) FROM Order O GROUP BY O.cid

Solution:

https://www.dropbox.com/sh/qygt7wp93alcy7k/AACasI-2IsplU8CiRL3c1o3Xa?dl=0

- SELECT C.cid, O.total FROM Customer C, Order O
  WHERE month(startDate)=7 and C.cid=O.cid

Solution:

https://www.dropbox.com/sh/53432t19etg2pyh/AAA-SIeewUqiWz1O1je0w_Wca?dl=0

- SELECT C.cid, O.total
  FROM Customer C LEFT OUTER JOIN ON Order O ON C.cid=O.cid
  WHERE month(startDate)=7

Solution:

https://www.dropbox.com/sh/ddta97xcwngeh0s/AAAur56wdedrg5I06thK3U75a?dl=0

For testing the jobs use the following files

https://www.dropbox.com/s/tmt6u80mkrwfjkv/Customer.txt

https://www.dropbox.com/s/8n5cbmufqhzs4r3/Order.txt

Exercice 5. Graph analytics. Design and implement algorithms in Spark for the following two analytics problems on directed graphs. These problems are at the core of several analytics problems on graphs. Assume that a graph is a set of edge pairs (v,w) indicating that an edge exists from node v to node w.

1. **Universal sinks.** Given a directed graph, find the set of nodes having an incoming edge from all the remaining nodes, and having no outgoing edges.

*Please before reading the solution, read the exercise text until the end. **

We first load the data into an RDD, by assume that the graph file is on the local file system (hence the *file:* prefix below)

```
 g = sc.textFile(  “file:/home/dario.colazzo/data/graphs/graph.txt")
```

We then perform the preprocessing to retain in each label only the couples (v,w) such that there is an edge from v to w (here we are not interested on edge information).

```
graph = g.map(lambda x: (   x.split('\t')[0].encode('utf-8'),
x.split('\t')[2].encode('utf-8')   ) )
```

Note the use of .encode(‘utf-8’) changing the string encoding, and so eliminating the u’ prefixes (indicating Unicode encoding) of strings extracted from the initial file.

IMPORTANT: assume in the initial graph file we have multiple edges from v to w with, of course, different labels (for instance : “2 5 7 7” and “2 9 7 7” meaning from node “2” we have two outgoing edges whose labels are “5” and “7”, respectively). The preprocessing above will drop information about edges “5” and “9”, so it will create an RDD where the string (2, 7) is repeated twice.

So we eliminate potential duplicates from graph:

```
graph_unique = graph.distinct()
```

We need to count how many nodes we have in graph_unique:

```
nodes = graph_unique.flatMap(lambda x: [x[0]]+
[x[1]]).distinct().count()
```

Now, as discussed in class we eliminate from graph_unique all the pairs (v,w) such that (w,u) exists in graph_unique, as in this case w has not to be considered as a potential sink (recall that a sink can not emit any edges). We first need to compute the inverse of graph_unique

```
graph_unique_inv = graph_unique.map(lambda x : (x[1], x[0]))
```

And then we compute the graph gss on which we will look for sinks

```
gss = graph_unique_inv.subtractByKey(graph_unique).map(lambda x : (x[1],
x[0]))
```

Concerning the previous example on (v,w), it is first inverted, so to obtain (w,v), and then it is eliminated from graph_unique_inv by subtractByKey since (w,u) is in graph_unique. Observe that, after subtraction we need to inverse again to obtain gss, the part of graph_unique that is needed to look for sinks.

```
pre_sinks = gss.map(lambda x: (x[1], 1)).reduceByKey(lambda x,y :
x+y).filter(lambda x : x[1] == nodes-1)

sinks = pre_sinks.map(lambda x: x[0])
```

As you will see by means of sinks.collect() the graph contains no sink.

Replay now all the sequence of RDD manipulations, by making the following RDD creation for graph.

```
graph = sc.parallelize([('1','4'), ('2','4'), ('3','4'), ('3','2') ])
```

Here there is one sink, the node ‘4’.

2. **Triangles enumerations.** Given a directed graph, enumerate all directed triangles, with no duplicates.

Here is the solution:

```
>>> g = sc.parallelize([(1,4), (4,2), (2,1), (3,2), (2,5), (5,3)])
>>> g.collect()
[(1, 4), (4, 2), (2, 1), (3, 2), (2, 5), (5, 3)]

>>> gInverse = g.map(lambda x: (x[1],x[0]))

>>> gInverse.collect()
[(4, 1), (2, 4), (1, 2), (2, 3), (5, 2), (3, 5)]

>>> opentriangles=g.join(gInverse)

>>> opentriangles.collect()
[(1, (4, 2)), (2, (1, 4)), (2, (1, 3)), (2, (5, 4)), (2, (5, 3)), (3,
(2, 5)), (4, (2, 1)), (5, (3, 2))]


>>> trianglesToClose = opentriangles.map(lambda x : ( (x[1] , x)))

>>> trianglesToClose.collect()
[((4, 2), (1, (4, 2))), ((1, 4), (2, (1, 4))), ((1, 3), (2, (1, 3))),
((5, 4), (2, (5, 4))), ((5, 3), (2, (5, 3))), ((2, 5), (3, (2, 5))),
((2, 1), (4, (2, 1))), ((3, 2), (5, (3, 2)))]

>>> triangles = trianglesToClose.join(g.map(lambda x : (x, 1)))

>>> triangles.collect()
```

```
[((4, 2), ((1, (4, 2)), 1)), ((2, 5), ((3, (2, 5)), 1)), ((3, 2), ((5,
(3, 2)), 1)), ((1, 4), ((2, (1, 4)), 1)), ((5, 3), ((2, (5, 3)), 1)),
((2, 1), ((4, (2, 1)), 1))]
```

```
>>> triangles.count()

6
```

These two problems are fundamental problems in many application tasks related to graph analysis, and find applications to Internet and social network analysis, just to mention a few.

Pay attention to the fact that each of these problems may require at least two jobs in order to be solved. The second job takes as input the output of the first job.

For both exercices, in order to test your Spark implementation first manually build very simple input graphs in the form of text files where each line models an edge (v, w) and contains a string for v followed by a \t , in turn followed by a string for w.

Then consider the text file graph.txt in

https://www.dropbox.com/s/3bre2e1jbsysxej/graph.txt

containing a bigger graph. Concerning the **Universal sink** exercice, in the case where there is no universal sink, find the nodes n having maximum number of (unique) nodes m connected via an (m,n) edge.

Pay attention to the fact each line the graph.txt file (obtained by a variant of the graph generator GTgraph) is of the form "v l w w" where v,l,w are integer values and l is the label for the an edge from v to w; w is repeated repeated twice. Just perform pre-processing to retain only (v,w) couples.