CS6200 Section 2　　Assignment 4　　Amogh Huilgol

Design Discussion

Describe the steps taken by Spark to execute your source code. In particular, for each method invocation of your Scala Spark program, give a brief high-level description of how Spark processes the data. (10 points)

The spark application performs following steps to compute page rank

1) Read the file using sc.textFile() function
2) Pass the data that is read from file to parser (written in Java) and then use filter to eliminate nulls
3) The output from the above step is split based on certain delimiter("######" in our case) to get List[Strings]. The output of this step is (String,List[String]) indicating pair (pageName, adjacencyList)
4) Adding the missing dangling nodes and combine the results
5) Flatten the RDD by converting the RDD to pair RDD
6) Run the reduce job for generating (page, adjacencyList) and saving it in memory for future iterations
7) Creating a default page Rank and then join it to the (page, adjacencyList)
8) Iterate over 10 iterations to calculate page rank for each node
9) Once the page ranks for all nodes is obtained ,we calculate topK nodes that have highest page rank . This is done by **.top(100)** function

Compare the Hadoop MapReduce and Spark implementations of PageRank. (10 points)

• For each line of your Scala Spark program, describe where and how the respective functionality is implemented in your Hadoop jobs.

There are three main processes to calculate Page Rank :

a) Pre-Processing
b) Page Rank Calculation
c) Top K Calculation

Pre-Processing

In Scala code , the following lines do the pre-processing

```
// reading path of input file through command line
val data = sc.textFile(args(0)+"/*.bz2").
  map(line => WikiParser.makeGraph(line)).
  filter(graphData => graphData != null).
  map(nodeName => nodeName.split("######")). //splitting based on a sequence
of text
  map(name2 => (name2(0),
  if (name2.size > 1) name2(1).split(", ").toList else List[String]())).
  map(node => List((node._1, node._2)) ++ node._2.
    map(adjNode => (adjNode, List[String]()))). // Create missing dangling nodes
  flatMap(node => node).
  reduceByKey((x, y) => (x ++ y)).
  persist()
```

In the above code, we read input from all the .bz2 files in the directory and pass the data to Parser file (Java code). The output of parser is then cleaned to remove null records. We then split each record based on the delimiter '######' which is used to generate page name and adjacency list. We then add the missing dangling nodes and combine the results . We then flatten the RDD and call normal reduce method to generate page-adjacency-list entries. The output is then persisted for future use

In Map reduce Job this is done by parseMapper.java and parseReducer.Java programs.

Page-Rank Computation

In Scala code, the page rank computation happens in the following lines of code

```
//getting value of |N|
val pageCount = data.count()


//assigning default page rank to each node . i.e 1/|N|
val ranks = data.map(node => (node._1, 1.0 / pageCount))


var graph = data.join(ranks);

for (i <- 1 to 10) {

  //logic to compute delta
  val delta = graph.filter(node => node._2._1.length == 0).
    reduce((a, b) => (a._1, (a._2._1, a._2._2 + b._2._2)))._2._2

  // adding page rank contribution to each node
  val page_ranks = graph.values
    .map(adjNodes => (adjNodes._1.map(node => (node, adjNodes._2 /
adjNodes._1.size)))).
    flatMap(node => node).
    reduceByKey((x, y) => x + y)

  //adding delta factors to the nodes using left join and handling nodes with no
inlinks
  graph = data.leftOuterJoin(page_ranks).
    map(n1 => {
      (n1._1, (n1._2._1, n1._2._2 match {
        case None => (alpha / pageCount) + ((alpha_rem) * delta / pageCount)
        case Some(x: Double) => (alpha / pageCount) + (alpha_rem * ((delta /
pageCount) + x))
      }))
    })
}
```

In the above code, we initially calculate default page rank in the following line

```
val ranks = data.map(node => (node._1, 1.0 / pageCount))
```

After the initial page rank, we run 10 iterations of page rank iterations . We compute the delta contribution in the following line

```
 //logic to compute delta
 val delta = graph.filter(node => node._2._1.length == 0).
   reduce((a, b) => (a._1, (a._2._1, a._2._2 + b._2._2)))._2._2
```

Once the delta is calculated, the page rank contribution of each node is calculated in the following lines

```
 // adding page rank contribution to each node
 val page_ranks = graph.values
   .map(adjNodes => (adjNodes._1.map(node => (node, adjNodes._2 /
adjNodes._1.size)))).
   flatMap(node => node).
   reduceByKey((x, y) => x + y)
```

The next step is to handle nodes that do not contain any inlinks and hence do not receive any computations. We need to preserve these nodes for future iterations. Hence we do a left join to get those nodes. This is done in following lines

```
 graph = data.leftOuterJoin(page_ranks).
   map(n1 => {
     (n1._1, (n1._2._1, n1._2._2 match {
       case None => (alpha / pageCount) + ((alpha_rem) * delta / pageCount)
       case Some(x: Double) => (alpha / pageCount) + (alpha_rem * ((delta /
pageCount) + x)) }))
```

In Map Reduce the Page Rank Compuation is done by calling the modules PageRankMapper.java and PageRankReducer.java for 10 iterations. The intermediate data is stored back in Hdfs

Top K Computation

In Scala , the Top-K computation is done by following lines of code

```
val output = graph.map(v => {
  (v._2._2, v._1)
}).
  top(100)
```

In Map reduce, the Top-K computation is done by TopKMapper.java and TopKReducer.java modules where we use TreeMap to compute Page ranks

Discuss the advantages and shortcomings of the different approaches. This could include, but is not limited to, expressiveness and flexibility of API, applicability to PageRank, available optimizations, memory and disk data footprint, and source code verbosity

Advantages of using Spark

1) The amount of code needed to be written Is less in scala. In Scala implementation of Page Rank is achieved in 80 lines. However the Java version of the implementation in Hadoop needs more than 500 lines of code
2) Since we write less code , maintainability of code increases
3) Dangling node and delta node can be computed before reduce jobs without any additional computation
4) Spark keeps a JVM running on each node which helps in saving time to spawn new JVM's whereas in Map reduce , time is spent in spawning JVM's every time

5) Spark caches data in-memory which saves lot of I/O time whereas in Map reduce all intermediate data is saved in HDFS and significant time is required to read data from HDFS and writing data to HDFS
6) Spark uses DAG based computation which provides options for optimizations

Performance Comparison

The performance Comparison for 6 machines is as follows

Map Reduce : 57 minutes
Spark : 52 minutes

The performance comparison for 11 machines is as follows

Map Reduce : 30 minutes
Spark : 26 minutes

Discuss which system is faster and briefly explain what could be the main reason for this performance difference

As seen from above statistics, Spark is faster. This is due to

a) Spark caches data in-memory whereas map-reduce stores intermediate data in HDFS. Hence HDFS takes lot of time to read and write data to HDFS
b) Spark uses DAG based processing engine which allows for lots of optimization such as avoiding redundant reduce tasks
c) Spark keeps JVM running in node. hence time is not wasted in spawning of JVM's whereas Map reduce needs to spawn a JVM everytime.

Report the top-100 Wikipedia pages with the highest PageRanks, along with their PageRank values, sorted from highest to lowest, for both the simple and full datasets, from both the Spark and MapReduce execution. Are the results the same? If not, try to find possible explanations. (4 points)

The results of the page rank are the same . However in Spark , we are comnputing the delta value before the reduce phase since doing so doesn't result in loss of performance. Hence we see a slight change in page rank values and pages are updated with the correct page rank values in spark implementation

The results for top 100 pages are as follows

```
(0.002895612340097201,United_States_09d4)
(0.0025837662658429223,2006)
(0.0013756641642122783,United_Kingdom_5ad7)
(0.0011887559248941935,2005)
(9.331545921987524E-4,Biography)
(9.001438659452175E-4,Canada)
(8.946269909634951E-4,England)
(8.819707744050026E-4,France)
(8.274616141252904E-4,2004)
(7.562886831203115E-4,Germany)
(7.36458560886575E-4,Australia)
(7.160204348160967E-4,Geographic_coordinate_system)
(6.664083680801914E-4,2003)
(6.491299090540317E-4,India)
(6.337472702455914E-4,Japan)
(5.378696331119126E-4,Italy)
(5.361487027053221E-4,2001)
(5.288554115340109E-4,2002)
(5.261088552781753E-4,Internet_Movie_Database_7ea7)
(5.05338627655328E-4,Europe)
(5.015343994400534E-4,2000)
(4.8157318146452115E-4,World_War_II_d045)
(4.677556736359891E-4,London)
(4.479541246273308E-4,Population_density)
```

```
(4.4584862738815027E-4,Record_label)
(4.429924746820317 6E-4,1999)
(4.3990978261179264E-4,Spain)
(4.395693315386666E-4,English_language)
(4.1461816515566937E-4,Race_(United_States_Census)_a07d)
(4.136495368823274E-4,Russia)
(4.0666077969001176E-4,Wiktionary)
(3.8530050980075347E-4,Wikimedia_Commons_7b57)
(3.827812361076351E-4,1998)
(3.751212766256925E-4,Music_genre)
(3.6565269841030875E-4,1997)
(3.6102616020661433E-4,Scotland)
(3.6045735876055523E-4,New_York_City_1428)
(3.4384331363404146E-4,Football_(soccer))
(3.4312474248817553E-4,1996)
(3.3844560924704674E-4,Television)
(3.380003935101963E-4,Sweden)
(3.269634115661369E-4,Square_mile)
(3.262717267436496E-4,Census)
(3.229966918588742E-4,1995)
(3.2165967950471857E-4,California)
(3.1649468075906236E-4,China)
(3.1133725323703027E-4,Netherlands)
(3.107236747236988E-4,New_Zealand_2311)
(3.084883710911745E-4,1994)
(2.93671860855416E-4,1991)
(2.9131985809121593E-4,1993)
(2.89386728132904E-4,1990)
(2.8901262991085184E-4,New_York_3da4)
(2.884412303906937E-4,Public_domain)
(2.7906163399038774E-4,1992)
(2.787788273348962E-4,United_States_Census_Bureau_2c85)
(2.778208315541957E-4,Film)
(2.7595879051725433E-4,Scientific_classification)
(2.754066917352692E-4,Actor)
(2.7257320340277853E-4,Norway)
(2.7167995761518906E-4,Ireland)
(2.649319762745634E-4,Population)
(2.6177712980001057E-4,1989)
(2.557897929509218E-4,1980)
(2.555902889567623E-4,Marriage)
(2.5503294304771244E-4,January_1)
(2.542917856358834E-4,Brazil)
(2.529464259863486E-4,Mexico)
(2.5191426125792903E-4,Latin)
```

```
(2.4902485950715277E-4,1986)
(2.469965868947966E-4,Politician)
(2.4277841622087566E-4,1985)
(2.424000463321689E-4,1979)
(2.4189489896314498E-4,1982)
(2.4173203771582025E-4,French_language)
(2.415703542578 94E-4,1981)
(2.4117545225081595E-4,Per_capita_income)
(2.3980906233639755E-4,Poland)
(2.3948752284725367E-4,1974)
(2.393706993395123E-4,Album)
(2.3779856609833328E-4,South_Africa_1287)
(2.372284536614088E-4,Switzerland)
(2.3714836890764663E-4,1984)
(2.3714758540815123E-4,1987)
(2.369819628698466E-4,1983)
(2.3576252382941846E-4,Record_producer)
(2.331483660262823E-4,1970)
(2.317663016182171E-4,1988)
(2.304074207449563E-4,1976)
(2.291697977121181E-4,Km²)
(2.2749905467795415E-4,1975)
(2.2478568148544845E-4,1969)
(2.2454623105876834E-4,Paris)
(2.234838328185196E-4,Greece)
(2.2324046519028555E-4,1972)
(2.22457357515466E-4,Personal_name)
(2.219781710852829E-4,1945)
(2.2135127507557303E-4,1977)
(2.204035099755662E-4,Poverty_line)
(2.2034742559349087E-4,1978)
```