CS 6240 – Parallel Data Processing with Map Reduce
Section-01, HW-4, Biyanta Vipulbhai Shah

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

The data is read and referenced from (http://spark.apache.org/docs/latest/programming-guide.html)

SparkContext object coordinates the sets of Spark processes running on a cluster in your main program. On a particular cluster, the SparkContext connects to several types of cluster managers (in our case STANDALONE or YARN), to allocate resources across applications. Once connected to these clusters, Spark obtains the executors on nodes in the cluster, these are processes that run the computations and store data. It next, sends the application code (JAR files in our case) to the executors. Finally, SparkContext sends tasks to the executor to run.

Method Invocation	**High level description of Spark processing**
map	Returns a new RDD, by implementing a function to all the elements of the RDD.
filter	Returns a new RDD which contains only elements that satisfies the condition.
flatMap	Return a new RDD which flattens all the elements of this RDD.
keyBy	Creates tuples of the given RDD, by applying the specified function. When called on RDD[(K,V)] it converts into tuple, with the given function. Then K and V can be accessed as tuple._1 and tuple._2.
join	When called on RDD[(K,V)] and RDD[(K,W)], returns a RDD[(K,(V,W)] pairs with all pairs of elements for each key
reduceByKey	When called on a RDD[(K, V)] pairs, returns an RDD[(K, V)] where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V
subtractByKey	Returns an RDD with the pairs from an RDD, whose keys are not present in the other RDD.
union	Returns a new RDD, which contains elements in the source RDD and the one given in the argument.

mapValues	Returns a new RDD which performs function on only the values of the RDD. (map performs the function on the keys as well as the values in the RDD)
takeOrdered	Returns the first n elements of the RDD using either their natural order or a custom comparator.
reverse	Reverses the string column and returns it as a new string column

Compare the Hadoop MapReduce and Spark implementations of PageRank

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

<table>
<tr><th>Spark Execution</th><th>Hadoop Execution</th></tr>
<tr><td>

```
var pagesWithoutTilde = sc.textFile(inputFile,
sc.defaultParallelism)
.map(record => Bz2WikiParser.parseXML(record))
.filter(record => !record.contains("tilde record"))
 .map(record => record.split("BIYANTA")) .map(record =>
if(record.length == 1){
                     (record(0),List())
              }
              else {
           (record(0), record(1).split("~").toList)
              })
pagesWithoutTilde.persist()

var pagesWithDanglingNodes = pagesWithoutTilde.values
        .flatMap (page => page)
        .keyBy(page => page)
        .reduceByKey((outlink1,outlink2) =>
outlink1.++(outlink2))
  .map(record => (record._1,List[String]()))

var finalPages =
pagesWithoutTilde.union(pagesWithDanglingNodes)
  .reduceByKey((outlink1,outlink2) =>
outlink1.++(outlink2))
```

</td><td>The file where this functionality is implemented in Hadoop is PageRankPreProcess.java

Map: Gets the data from the Bz2 parser. Returns a custom object which contains the node and its out-links. If the node does not have any out-links (dangling nodes), it returns an empty list.

Reduce: Converts out-links into strings, and adds page rank = -1.0 (which shows that page rank still has not been calculated)</td></tr>
<tr><td>

```
val totalRecords = finalPages.count()

val initialPageRank: Double = (1.0 / totalRecords)

val alpha: Double = 0.15

var finalPagesWithRanks = finalPages.keys.map(record =>
(record,initialPageRank))
```

</td><td>The file where this functionality is implemented in Hadoop is PageRankCalculate.java
and PageRankFinalDeltaCalc.java

It gets its input from PageRankPreProcess.java output.</td></tr>
</table>

<table>
<tr>
<td>

```
for (i <- 1 to 10) {
  try{
    var danglingMass = sc.accumulator(0.0)
    var pageRankCalc =
finalPages.join(finalPagesWithRanks).values

      .flatMap {
        case (outLinks, pageRank) => {

          val size = outLinks.size
          if (size == 0) {
            danglingMass += pageRank
            List()
          }
          else {
            outLinks.map(link => (link, pageRank / size))

          }
        }
      }.reduceByKey(_+_)

    pageRankCalc.first()
    val delta : Double = danglingMass.value

    var one =
finalPagesWithRanks.subtractByKey(pageRankCalc)
    var two = one.map(page =>
(page._1,0.0)).union(pageRankCalc)
    finalPagesWithRanks =
two.mapValues[Double](accumulatePageRank => (alpha /
totalRecords
      + (1- alpha) * (delta / totalRecords +
accumulatePageRank)))
```

</td>
<td>
Map: Assigns initial page rank in the first iteration. Emits each node and its adjacency list and page rank. For dangling nodes emits the “dummy” key and the page rank. Also emits the P(m)/C(m) calculation for each page in the outlinks. The Map also updates the page ranks by adding the delta values for iteration i in iteration i+1.

Reduce: Calculates the delta, by the page ranks of the dangling nodes and does an intermediate calculation of page ranks without the dangling node mass value.

Since the page ranks are updated in iteration i+1 for iteration i, we will need an extra map job to just compute the final updated page ranks for iteration i. This computation is done in PageRankFinalDeltaCalc.java

The Driver program is from where the PageRankCalculate.java is iterated over 10 times.
</td>
</tr>
<tr>
<td>

```
var sorted =
finalPagesWithRanks.takeOrdered(100)(Ordering[Doub
le].reverse.on { line => line._2 })

var finalTopK =
sc.parallelize(sorted).saveAsTextFile(outputFile)
```

</td>
<td>
The file where this functionality is implemented in Hadoop is PageRankTopK.java

It gets its input data from PageRankFinalDeltaCalc.java output.

Map: Computes the local top 100 for page ranks.

Reduce: Combines all these local top 100 records and generates
</td>
</tr>
</table>

	global top-100 page ranks with its pages.

- **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.**

Comparison Measures	Spark	Hadoop
Memory and disk data footprint	Spark processes data-in-memory. Thus Spark needs a lot of memory. If data is too big to fit into memory, then there would be performance degradation. Hence **a lot of memory is used** but since we do not keep writing to the disk, **disk data footprint is less**.	Hadoop MapReduce persists back to the disk after a map or reduce action. It kills a job as soon as it is done. Thus **less memory is used**. However we write to and fro to HDFS for iterative computations, thus **increasing the disk data footprint.**
Expressiveness and flexibility of API	Spark has comfortable APIs for Java, Python and Scala, hence **having increased flexibility for API than Hadoop.** Spark also has an interactive mode for running commands making it **more expressive** as well.	Hadoop MapReduce is written in Java and is infamous for being very **difficult to program.** Although it has tools to make it easier. Map Reduce does not have an interactive command line, being **less expressive** for its users.
Source Code Verbosity	Spark code is very **concise**.	Hadoop MapReduce code will be **high in verbosity**
Cost	Since the Spark needs data to fit into the memory, if the memory in the Spark cluster is at least as large as the data we need to process, then it'll be the cheaper option.	If the cluster isn't as big as the data is , then Hadoop MapReduce would be the cheaper option since hard disk is quite cheaper than memory space.
Applicability to Page Rank	Spark is **applicable** to Page Rank	Hadoop as well is **applicable** to Page Rank. The difference between the two would be the trade-offs explained in point 1, 2 and 3. The programmer can take those points into consideration and choose a framework.
Fault Tolerance	Spark has several retries per task, however since it relies on memory, when Spark program fails, it needs to start processing from the start.	Hadoop MapReduce also has several retries per task, but MapReduce relies on hard disk, so if a process fails then it could start it from where it failed, thus saving time. **Hadoop would thus be more fault tolerant than Spark.**

Performance Comparison

Run #	Spark Execution Time	Hadoop Execution Time
Run - 1 (5 worker machines)	6454 seconds	3868 seconds
Run - 2 (10 worker machines)	2722 seconds	2287 seconds

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

According to my understanding and readings done from the modules, since Scala is better for iterative computation due to less writes on HDFS, it should be faster. However, as you can see from the above comparison, such is not the case.
One reason could be that I am not able to efficiently understand how RDD's are persisted and thus am not making correct design decisions, also partitioning, reduce and map methods may be used incorrectly, hence slowing down the process.
Also another plausible explanation for this could be, when we are parsing the compressed bz2 file; Spark, by default might not be parallely processing the parser written in Java, since pre-processing the input file is a major chunk of the data load, the same could be causing delays in the total execution time. I have tried to optimize the code and run it to the best of my abilities and understanding.

Spark Execution:

Output of the simple Wikipedia data set on local machine (standalone mode)

PAGE NAMES	PAGE RANK
United_States_09d4	0.005189009
Wikimedia_Commons_7b57	0.004806766
Country	0.003940285
England	0.002752481
Water	0.00268781
Animal	0.002554088
City	0.002510824
United_Kingdom_5ad7	0.002358647
Germany	0.002350402
Earth	0.002324735
France	0.002323608
Europe	0.002038097
Wiktionary	0.001753884
English_language	0.001749677
Government	0.001732345

Computer	0.00171684
India	0.001713171
Money	0.001667384
Japan	0.001551691
Plant	0.00152356
Italy	0.001507433
Canada	0.001481407
Spain	0.001471124
Food	0.001424687
Human	0.001412097
China	0.001396715
People	0.001382249
Australia	0.001329854
Asia	0.001284436
Capital_(city)	0.001274268
Television	0.001264997
Sun	0.00126021
Number	0.001243236
State	0.001240376
Sound	0.001235212
Science	0.001232543
Mathematics	0.001231057
Metal	0.001192305
Year	0.001177093
2004	0.001173357
Language	0.001150166
Russia	0.001146182
Wikipedia	0.00112333
Religion	0.001098567
19th_century	0.001096539
Music	0.001087431
Scotland	0.001054801
20th_century	0.001053705
Greece	0.001049223
Latin	0.001029861
London	0.001027355
Greek_language	0.001004357
Energy	9.99E-04
World	9.86E-04
Centuries	9.76E-04
Culture	9.45E-04
History	9.36E-04
Liquid	9.15E-04
Netherlands	9.06E-04
Planet	9.05E-04
Light	9.02E-04

Society	9.01E-04
Atom	8.90E-04
Wikimedia_Foundation_83d9	8.88E-04
Scientist	8.88E-04
Image	8.88E-04
Law	8.86E-04
Geography	8.79E-04
List_of_decades	8.79E-04
Uniform_Resource_Locator_1b4e	8.62E-04
Africa	8.61E-04
Turkey	8.45E-04
Inhabitant	8.30E-04
Capital_city	8.23E-04
Plural	8.22E-04
Electricity	8.14E-04
Poland	7.97E-04
Building	7.97E-04
Car	7.95E-04
Sweden	7.92E-04
Book	7.91E-04
Biology	7.87E-04
War	7.71E-04
Chemical_element	7.68E-04
God	7.61E-04
North_America_e7c4	7.56E-04
September_7	7.55E-04
Website	7.46E-04
Nation	7.43E-04
Politics	7.40E-04
2006	7.33E-04
Fish	7.32E-04
Species	7.31E-04
Mammal	7.22E-04
Island	7.18E-04
Portugal	7.17E-04
Gas	7.16E-04
River	7.12E-04
Switzerland	7.06E-04
World_War_II_d045	7.02E-04

Output of the full Wikipedia data set

PAGE NAMES	PAGE RANK
United_States_09d4	0.002622883
2006	0.001228497
United_Kingdom_5ad7	0.001203135

Biography	9.82E-04
2005	9.17E-04
England	8.80E-04
Canada	8.56E-04
Geographic_coordinate_system	7.72E-04
France	7.25E-04
2004	7.20E-04
Australia	6.80E-04
Germany	6.54E-04
2003	5.87E-04
India	5.83E-04
Japan	5.83E-04
Internet_Movie_Database_7ea7	5.34E-04
Europe	5.09E-04
Record_label	4.91E-04
2001	4.87E-04
2002	4.83E-04
World_War_II_d045	4.78E-04
Population_density	4.70E-04
Music_genre	4.67E-04
2000	4.65E-04
Italy	4.46E-04
Wiktionary	4.36E-04
Wikimedia_Commons_7b57	4.35E-04
London	4.35E-04
English_language	4.18E-04
1999	4.06E-04
Spain	3.63E-04
1998	3.56E-04
Russia	3.44E-04
1997	3.37E-04
Television	3.36E-04
New_York_City_1428	3.35E-04
Football_(soccer)	3.26E-04
1996	3.24E-04
Census	3.24E-04
Scotland	3.22E-04
1995	3.10E-04
China	3.09E-04
Population	3.04E-04
Square_mile	3.04E-04
Scientific_classification	3.04E-04
California	3.02E-04
1994	2.91E-04
Sweden	2.88E-04
Public_domain	2.87E-04

Film	2.86E-04
Record_producer	2.84E-04
New_Zealand_2311	2.83E-04
New_York_3da4	2.79E-04
Netherlands	2.77E-04
Marriage	2.76E-04
1993	2.75E-04
United_States_Census_Bureau_2c85	2.75E-04
1991	2.72E-04
1990	2.68E-04
1992	2.66E-04
Politician	2.65E-04
Album	2.61E-04
Latin	2.60E-04
Actor	2.58E-04
Ireland	2.58E-04
Per_capita_income	2.56E-04
Studio_album	2.52E-04
Poverty_line	2.51E-04
Km²	2.50E-04
1989	2.47E-04
Norway	2.41E-04
Website	2.39E-04
1980	2.35E-04
Animal	2.29E-04
Area	2.29E-04
1986	2.27E-04
Personal_name	2.26E-04
Poland	2.26E-04
Brazil	2.26E-04
1985	2.24E-04
1987	2.23E-04
1983	2.22E-04
1982	2.21E-04
French_language	2.19E-04
1981	2.19E-04
1979	2.19E-04
1984	2.19E-04
World_War_I_9429	2.19E-04
1988	2.19E-04
Paris	2.18E-04
1974	2.18E-04
Mexico	2.16E-04
19th_century	2.12E-04
1970	2.11E-04
January_1	2.11E-04

USA_f75d	2.11E-04
1975	2.09E-04
1976	2.08E-04
Africa	2.08E-04
South_Africa_1287	0.000207360149838549

Hadoop Execution:

Output of the simple Wikipedia data set on local machine (standalone mode)

PAGE NAMES	PAGE RANK
United_States_09d4	0.005189009
Wikimedia_Commons_7b57	0.004806766
Country	0.003940285
England	0.002752481
Water	2.69E-03
Animal	2.55E-03
City	2.51E-03
United_Kingdom_5ad7	2.36E-03
Germany	2.35E-03
Earth	2.32E-03
France	2.32E-03
Europe	2.04E-03
Wiktionary	1.75E-03
English_language	1.75E-03
Government	1.73E-03
Computer	1.72E-03
India	1.71E-03
Money	1.67E-03
Japan	1.55E-03
Plant	1.52E-03
Italy	1.51E-03
Canada	1.48E-03
Spain	1.47E-03
Food	1.42E-03
Human	1.41E-03
China	1.40E-03
People	1.38E-03
Australia	1.33E-03
Asia	1.28E-03
Capital_(city)	1.27E-03
Television	1.26E-03
Sun	1.26E-03

Number	1.24E-03
State	1.24E-03
Sound	1.24E-03
Science	1.23E-03
Mathematics	1.23E-03
Metal	1.19E-03
Year	1.18E-03
2004	1.17E-03
Language	1.15E-03
Russia	1.15E-03
Wikipedia	1.12E-03
Religion	1.10E-03
19th_century	1.10E-03
Music	1.09E-03
Scotland	1.05E-03
20th_century	1.05E-03
Greece	1.05E-03
Latin	1.03E-03
London	1.03E-03
Greek_language	1.00E-03
Energy	9.99E-04
World	9.86E-04
Centuries	9.76E-04
Culture	9.45E-04
History	9.36E-04
Liquid	9.15E-04
Netherlands	9.06E-04
Planet	9.05E-04
Light	9.02E-04
Society	9.01E-04
Atom	8.90E-04
Wikimedia_Foundation_83d9	8.88E-04
Scientist	8.88E-04
Image	8.88E-04
Law	8.86E-04
Geography	8.79E-04
List_of_decades	8.79E-04
Uniform_Resource_Locator_1b4e	8.62E-04
Africa	8.61E-04
Turkey	8.45E-04
Inhabitant	8.30E-04
Capital_city	8.23E-04
Plural	8.22E-04
Electricity	8.14E-04
Poland	7.97E-04
Building	7.97E-04

Car	7.95E-04
Sweden	7.92E-04
Book	7.91E-04
Biology	7.87E-04
War	7.71E-04
Chemical_element	7.68E-04
God	7.61E-04
North_America_e7c4	7.56E-04
September_7	7.55E-04
Website	7.46E-04
Nation	7.43E-04
Politics	7.40E-04
2006	7.33E-04
Fish	7.32E-04
Species	7.31E-04
Mammal	7.22E-04
Island	7.18E-04
Portugal	7.17E-04
Gas	7.16E-04
River	7.12E-04
Switzerland	7.06E-04
World_War_II_d045	7.02E-04

Output of the full Wikipedia data set

PAGE NAMES	PAGE RANKS
United_States_09d4	0.002622934
2006	0.001228507
United_Kingdom_5ad7	0.001203149
Biography	9.82E-04
2005	9.17E-04
England	8.80E-04
Canada	8.56E-04
Geographic_coordinate_system	7.72E-04
France	7.25E-04
2004	7.20E-04
Australia	6.80E-04
Germany	6.54E-04
2003	5.87E-04
India	5.83E-04
Japan	5.83E-04
Internet_Movie_Database_7ea7	5.34E-04
Europe	5.09E-04
Record_label	4.91E-04
2001	4.87E-04

2002	4.83E-04
World_War_II_d045	4.78E-04
Population_density	4.70E-04
Music_genre	4.67E-04
2000	4.65E-04
Italy	4.46E-04
Wiktionary	4.36E-04
Wikimedia_Commons_7b57	4.35E-04
London	4.35E-04
English_language	4.18E-04
1999	4.06E-04
Spain	3.63E-04
1998	3.56E-04
Russia	3.44E-04
1997	3.37E-04
Television	3.36E-04
New_York_City_1428	3.35E-04
Football_(soccer)	3.26E-04
1996	3.24E-04
Census	3.24E-04
Scotland	3.22E-04
1995	3.10E-04
China	3.09E-04
Population	3.04E-04
Square_mile	3.04E-04
Scientific_classification	3.04E-04
California	3.02E-04
1994	2.91E-04
Sweden	2.88E-04
Public_domain	2.87E-04
Film	2.86E-04
Record_producer	2.84E-04
New_Zealand_2311	2.83E-04
New_York_3da4	2.79E-04
Netherlands	2.77E-04
Marriage	2.76E-04
1993	2.75E-04
United_States_Census_Bureau_2c85	2.75E-04
1991	2.72E-04
1990	2.68E-04
1992	2.66E-04
Politician	2.65E-04
Album	2.61E-04
Latin	2.60E-04
Actor	2.58E-04
Ireland	2.58E-04

Per_capita_income	2.56E-04
Studio_album	2.52E-04
Poverty_line	2.51E-04
Km²	2.50E-04
1989	2.47E-04
Norway	2.41E-04
Website	2.39E-04
1980	2.35E-04
Animal	2.29E-04
Area	2.29E-04
1986	2.27E-04
Personal_name	2.26E-04
Poland	2.26E-04
Brazil	2.26E-04
1985	2.24E-04
1987	2.23E-04
1983	2.22E-04
1982	2.21E-04
1981	2.19E-04
French_language	2.19E-04
1979	2.19E-04
1984	2.19E-04
World_War_I_9429	2.19E-04
1988	2.19E-04
Paris	2.18E-04
1974	2.18E-04
Mexico	2.16E-04
19th_century	2.12E-04
1970	2.11E-04
January_1	2.11E-04
USA_f75d	2.11E-04
1975	2.09E-04
1976	2.08E-04
Africa	2.08E-04
South_Africa_1287	2.07E-04

Are the results the same?

Yes, the results for Spark and Hadoop execution are the same.