Name : Sushil Thasale

Project : PageRank in Scala

<u>Steps taken by Spark to execute source code:</u>

A) Job Initialization Step:
1. Initial we create a Spark configuration. Using this configuration we assign a name to our Spark application. We also set the master to local. This tell the Spark application that we are going to run on a local HDFS server.
2. Now Spark changes its data compression technique to lz4. This is done for optimization.
3. Now Spark changes its garbage collector to specified option.
4. Using the above configuration a new Spark Context is created.

B) Preprocessing:
1. Spark reads the input file and divides the lines amongst the available partitions.
2. Now, Spark selects each partition and perform the task mentioned in mapPartitions(). Inside mapPartitions(), Spark returns an Iterator over all the lines within that partition. Now, Spark creates and object of Bz2WikiParser class, dedicated for that partition and calls its parseLine function for each of line within that partition.
3. The Bz2 parser returns null string for bad pages (pages with name containng ~), while it returns a string "page-name : outlinks" for good pages. These strings returned by parser are assigned to corresponding partitions by Spark. Now Spark, applies filter on each line within each partition to eliminate bad pages.
4. Spark iterates over each string within each partition and processes the string to fetch (page,Array[outlinks]) .
5. Now, the RDD "parsedPages" is cached (persisted in memory) for optimization. Since, Spark is lazy, the above mentioned transformations will be performed only after cache action is called.
6. Spark performs an action "count" on RDD parsedPages, which returns the no. of elements spread over RDD. Using this page count, initial page-rank is computed.
7. Now Spark iterates over each element within each partition of parsedPages RDD and assigns an initial page rank to each of the element. The map transformation returns a new RDD which is stored as newPageList{page, (outlinks, page-rank)}.

C) Page Rank Iterations:
For each iteration of for loop, Spark performs following actions:

a. Sink Sum Computation:
Spark iterates over each element within each partition of newPageList RDD and filter out non-sink pages, emit page-ranks of such pages and compute sum using sum action. Since, Spark is lazy, newPageList will be created only when sum action is performed on it. Till that time newPageList will just have nominal existence in memory.

b. Compute Contributions:
Here, Spark iterates over each element with each partition of newPageList RDD and filter out sink pages. Later, for each outlink of the current element, Spark computes the element's contribution. This is done for all elements I.e pages. Now, Spark perform shuffling of data to group them by their keys.

Later, it computes the sum of contributions for each key. The intermediate output at this stage would be {page,unrefined-page-rank}. Now, for each of the elements in intermediate result, Spark applies computePageRank(), to refine page rank values.

c. Recover Outlinks for next iteration:
Here, Spark performs shuffling to join elements from parsedPages RDD to newPageRank RDD based on the keys. The resultant RDD is stored in newPageList. This RDD is later used in subsequent iteration.

D) Top-100:
1. Here, Spark sorts the elements inside newPageList RDD, in descending order of page rank values and select the top 100 elements. Now, the value returned is an Array of elements and not a RDD.
2. Now, Spark convert the array to a RDD by parallelizing it.
3. repartitionAndSortWithinPartitions causes Spark to shuffle the elements from top100 RDD and direct them to a single partition and sort them.
4. Later, Spark unpersists the cached parsedPages RDD, top100 RDD and writes the elements to a text file.

Scala code explanation and comparison with Hadoop MapReduce:

1.

```
val parsedPages = sc.textFile(args(0))
                    .mapPartitions{
                        lineIter =>
                        val bz: Bz2WikiParser = new Bz2WikiParser()
                        lineIter.map(line => bz.parseLine(line))
                        }
                    .filter(page => !(page == ""))
                    .map(line => line.split(" => "))
                    .map(fields => getPageData(fields))
                    .cache
```

Spark:
This snippet of Spark code reads the input file and parses it using Java's BZ2 parser. Now each partition would get some piece of input file. Here, for each partition, we create an object of Bz2WikiParser and call the parseLine() for each of the line within that partition. The parser returns a null string for bad pages, whereas it returns {page-name => outlinks} for good pages. Now, we remove bad pages by applying filter on the parser output. Later, we split the line on "=>" and process the output to get {page, outlinks}. We also cache the parsedPages RDD to improve efficiency.

Hadoop MapReduce:
The above mentioned functionality has been implemented in mapper of preprocessing job. The parser initializations have been performed in mapper setup, which ensures that parser initializations are performed only once per partition. The mapper does not cache the parser output, instead writes the output to local disk, so that it can be read by the next job.

2.

```
val pageCount = parsedPages.count
val initialPageRank = 1.0 / pageCount
```

Spark:
Here, pageCount stores the total no. of pages and compute the initial page rank as 1/pageCount

Hadoop MapReduce:
The above mentioned functionality has been implemented in mapper of preprocessing job using a global counter. In this mapper we simply count the total pages received to that particular map task and increment the global counter by that value. Once the preprocessing job is completed, we retrieve the total page count and compute initial page rank.

3.

```
var newPageList = parsedPages.map(page => (page._1, (page._2, initialPageRank)))
```

Spark:
parsedPages would contain {page-name : outlinks}. Here, for each of the page, we assign 1/pageCount as initial page rank.

Hadoop MapReduce:
The driver program adds initial page rank to context of 2nd preprocessing job. The reducer of 2nd preprocessing job retrieves initial page rank and assign it to each page.

4.
```
var sinkSum = newPageList.filter{ case (page, (outlinks, pageRank)) => (outlinks(0) == "")}
                        .map{ case (page, (outlinks, pageRank)) => pageRank}
                        .sum()
```

Spark:
First, we filter all the sink pages from pasredPages and compute the sum of page rank of all the sink pages.

Hadoop Mapreduce:
The above mentioned functionality has been implemented in reducer of the page-rank job. For each page that comes to reducer, it check whether it is a sink page. If yes, the reducer add the corresponding page-rank to sinkSum. In the cleanup, this sinkSum is passed to the driver program. The driver program adds this sinkSum to context of next job, where is used to refine page ranks.

5.
```
var newPageRanks = newPageList
                        .filter { case (page, (outlinks, pageRank)) => !(outlinks(0) == "") }
                        .flatMap { case (page, (outlinks, pageRank)) => outlinks.map(out => (out,
pageRank/outlinks.size)) }
                        .reduceByKey((accum, y) => accum + y)
                        .mapValues(v => computePageRank(v, pageCount, sinkSum))
```

Spark:
newPageList contains {page, (outlinks, page-rank)}. First, we filter out the sink pages. Then, for each of the non-sink pages we compute its contribution towards the page rank of its outlinks. Later, for each of the pages we aggregate the contribution using reduceByKey. Then, for each unique pages we call computePageRank function. This function computes the refined page rank considering the sinkSum and teleportation factor.

Hadoop MapReduce:
The mapper of page-rank job computes the contribution of input page towards the page rank of its outlink and emits (outlink, contribution). The reducer of page-rank job aggregates all the contributions for a particular page and computes the refined page rank adding teleportation factor and sinkSum.

6.
```
newPageList = parsedPages.join(newPageRanks)
```

Spark:
newPagerank would contain (page-name, page-rank). But, we need a page's outlinks for next iteration of page-rank loop. Therefore, we join parsedPages which contains outlinks to newPageRanks.

Hadoop MapReduce:
The reducer of page-rank job writes (page, page-rank, outlinks) to local disk, which are read by mapper of next page-rank job.

7.

```
val sortedPageRanks = newPageList.sortBy(_._2._2, false)
                                 .take(100)

val top100 = sc.parallelize(sortedPageRanks)
                .map(page => (page._2._2*(-1), page._1))
                .cache

val top100_Final = top100.repartitionAndSortWithinPartitions(rPartitioner)
                         .map(page => (page._2, page._1*(-1)))

top100_Final.saveAsTextFile(args(1))
```

Spark:
Here, we sort the pages in descending order of page-rank and take only first 100 pages. If we write the page as it is, we could get multiple files corresponding to the partitions inside the top100 RDD. To direct the output into a single file we use repartitionAndSortWithinPartitions(). Here, we could have used rapartition(1) and later sorted it again in descending order of page-rank. But, it would be inefficient as it would result in additional shuffle.

Hadoop MapReduce:
In mapper of final map reduce job, we emit (page-rank, page) to take advantage of hadoop's sorting and shuffling process. To sort them in descending order we define a custom comparator. Later, in reducer we just emit first 100 pages received.

Spark vs Hadoop MapReduce:

1. Performance(memory and disk data footprint):
Our Spark program processes data in-memory while Hadoop MapReduce persists the full dataset to HDFS after running each job. So Spark program is fast, but requires lot of memory for performing shuffling and computations. If Spark runs on Hadoop alongside some other resource-demanding application, or if the data is too big to fit entirely into the memory, then there could be major performance degradations for Spark. MapReduce does much better in these situations as it immediately removes data from memory when its not required also required data is written to disk.
Since, Spark stores the data in-memory, it performs better for iterative computations that need to pass over the same data many times. In such cases, Hadoop MapReduce needs to read from disk for each iteration, which increases runing time. In conclusion, Spark performs better when all the data fits in the memory; Hadoop MapReduce is designed for data that doesn’t fit in the memory and it can run well alongside other services.

2. Ease of use and source code verbosity:
Page-rank algorithm is easier to code in Spark. Hadoop MapReduce is comparatively difficult to program. In Hadoop MapReduce, to create a job we need to write separate mapper and reducer classes. But in Spark, we can implement the same functionality just by using map and reduceByKey() transformation. As a result, Spark code is more concise and readable.

Cost:
In case of Spark, the available memory should be as large as the amount of data you need to process, because the data has to fit into the memory for optimal performance. So, if you need to process really Big Data, Hadoop will definitely be the cheaper option since hard disk space comes at a much lower cost than memory space.

Failure Tolerance:
In case of MapReduce, if a process crashes in the middle of execution, it could continue where it left off (since data is written to disk), whereas Spark will have to start processing from the beginning (data in memory would be lost).

Applicability to PageRank:
Page-rank algorithm is computation intensive and iterative in nature. As a result, Spark would be a good choice, since it keep the intermediate result in memory and passes it to next iteration. Whereas, MapReduce would require transferring the intermediate result over network, writing it to disk and reading it back for next iteration. However, if the available memory is small, then page-rank using Spark could be a bad choice.

Available optimizations:
In Spark, some of the features available for optimization are as follows:
1. For shuffle intensive job, we can allocate more memory for shuffling task:
```
sparkConf.set("spark.storage.memoryFraction", "0.02")
sparkConf.set("spark.shuffle.memoryFraction", "0.8")
```

2. For computation intensive job, we can allocate more memory for computation and less memory for shuffling:
```
sparkConf.set("spark.storage.memoryFraction", "0.02")
sparkConf.set("spark.shuffle.memoryFraction", "0.6")
```

3. We can use suitable compresion technique e.g. lz4, for best possible compression and decrease the amount of data transferred over network while shuffling.
for e.g. conf.set("spark.io.compression.codec", "lz4")

4. We can also use a suitable garbage collector for better performance.
for e.g. conf.set("spark.executor.extraJavaOptions", "-XX:+UseG1GC")

Performance Comparison :

	Execution time	
Approach	6 m4-large	11 m4-large
Spark	2548	780
Hadoop MapReduce	1549	924

In case of six m4-large machines, the performance of Spark is really poor as compared to that of Hadoop MapReduce. This might be because, only 5 worker machines are used and the input data is too large to fit in memory of 5 machines combined. Also, this program is running on Hadoop YARN, alongside some other resource-demanding application. As a result, the fraction of memory available for Spark job is too small to process entire input in memory. This can be the possible reason for Spark's performance degradation.

In case of 11 m4-large machines, Spark performs better as compared to Hadoop MapReduce. Here we have 10 worker machines, each having 5GB of memory. Even after considering the resource-demanding applications running alongside it, the memory available for Spark job is almost double as compared to previous case. This memory might be sufficient for processing entire input. This can be the possible reason for improvement in performance of Spark.

Output Comparison of both approaches:

Top-100 Wikipedia pages with the highest PageRanks, for each version of the program have been reported under "All output and log files" folder.

The result is similar, but not exactly the same. These variation might be because of following reason:-
The input data is skewed; for some pages, the outlinks contain dead pages. By dead pages, we mean pages that are present as outlink of some page, but do not exist as a vertex in graph.
For example, consider the following graph:
A : {B, C, D}
B : {A, C}
C : {A, B}
Here, D is a dead page.

In mapper of page-rank job, we compute current page's contribution towards the page rank of each of its outlinks. Later, in reducer, for each received page we compute its page-rank and write to disk. These pages include the dead pages. Since dead pages do not have outlinks, they contribute to sinkSum (sum of page-rank of sink nodes), which intern affects the final page rank of all pages.

In Spark, I have followed a slightly different approach. Here, I eliminate dead pages i.e. do not consider them for computing sinkSum. parsedPages would contain only legal pages from input file, while newPageRanks would contain legal pages as well as dead pages. By performing an equi-join on parsedPages and newPageRanks, we get only legal pages, which are later considered for computing sinkSum.

References:
1. **https://www.xplenty.com/blog/2014/11/apache-spark-vs-hadoop-mapreduce/**
2. **http://fdahms.com/2015/10/04/writing-efficient-spark-jobs/**
3. **https://spark.apache.org/docs/latest/tuning.html**
4. **https://aws.amazon.com/blogs/aws/new-apache-spark-on-amazon-emr/**