REPORT

HW4

CS6240-Sec 2

By:

Abhay Kasturia

TABLE OF CONTENTS

Contents

HEADER

Class Number: CS6240 – Sec 2

HW number: HW4

Name: Abhay Kasturia

Map-Reduce Source Code

Present in the zip file, with separate folders for each of the programs.

To run the codes unzip the codes folder and copy all the source files to the input folder of the specific program and make sure there is no output directory in the specific programs folder.

Run “make run” command inside the folder of the program needed to run and the output can then be checked in the output folder in the same directory.

Design Discussion

Q. 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 applies it to the data.

At a high level, the driver program that is part of the Spark framework is assigned to run our main function and executes various *parallel operations* on a cluster.

Each operation given to the Spark(shuffling/mapping/writing data) is divided by the framework into multiple stages and each stage has multiple tasks which can are run parallely on the JVM of the worker node.

Another feature provided by Spark is a Resilient Distributed Dataset(RDD), which is a collection of elements that are partitioned and are spread across the nodes of the cluster. These partitions can be worked upon as parallel tasks.

Whenever we are taking an input in this case our bz2 compressed file(which is a Hadoop supported file) the RDDs are created when we load it into our program. All map filter reduce operations on RDD are then run in parallel on RDD.

Also , most RDD operations are lazy operation in the sense that are RDD are kept as a description of a series of operations and not actual data. So as for our file file loading:

```
val inputPages = sc.textFile(args(0))
```

It does nothing but creates an RDD that says "we need to load this file". The file is actually not loaded until we have some use for InputPages later in the code.

But if we perform an RDD operation that require the actual data to work with , like InputPages.count , the data for the file is read and the count is returned. But if we call the count operation again the data will be read and counted again ; thus we need to store the data into the memory for further operations and we should not be reading the file again and again . For the same reason we are using the persist and unpersist functions in the code. Persist loads the data into memory once read and unpersist unloads when not required. This enables faster operations !

This is valid for all the operations performed on inputPages, which includes parsePage and processedPages operations. In the end we persist the data we have for processedPages as that will be required for joins, count and other operations later.For parse page we apply an operation called mapPartitions , which is supposed to perform a certain operation for each of the partition of the mapper , which considerably reduces the running time. In our case we create an object of the parser class and for each iterator value we parse that line and save it to parsePage. This reduces the overhead of creating an object for each mapper call(as pointed out in last assignment).

The dangsum (the sum of PR of all the dangling nodes is calculated in each iteration as the first task) , the data is not persisted as it is required just once. It is calculated using simple map filter and sum operations (more explained in code commenting).

The pagerank is calculated by getting all the non dangling nodes and dividing their page rank to their outlinks using flatmap , which converts different array of elements to a single array of element. which is then reduced by the key (page which is the outlink) and all the page ranks for that particular page are accumulated and sum. and the final formula is applied using map. And the final output returned is (page , new page rank)

We unpersist the page val as it is supposed to be reset with the new page ranks and the old outlinks.

The outlinks are then get from processed pages via the key (the page name) , getting a key value pair of (page name , (outlinks , new page rank) , this is persisted as it is required in all the calculations for the next iteration.

And then the iteration continues.

The top 100 outcomes are achieved by sorting the result via page rank , getting top 100 out of that and then finally mapping it to a printable format.

Data Sources : StackOverflow , Spark documentation and several youtube videos.

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

The preprocessing job in Hadoop MapReduce , was performed by one mapper which was reading one line at a time and using the SAX parser to get the outlinks of the particular page.

Also a job counter was kept to track the total number of documents.

The SAX parser remains the same for Scala as well and the rest of the mapping is done as follows :

the parsePage contains data after being parsed by SAX , in the following format as a single string:

page_name:outlink1,outlink2,.....

```
val processedPages = parsePage // removing documents which are null
                        .filter(doc => !(doc == ""))
                        .map(doc => {
```

```
                    val page = doc.substring(0,doc.indexOf(":"))
                    (page , doc)})
                    .map(x => {
                      val (page , doc) = x
                      if (doc.length > (page.length+1)) {
                      /// getting the outlinks from the second part of the document
                      val outlinks = doc.substring(doc.indexOf(":")+1)
                      (page, outlinks.split(","))
                   } else
                        // when outlinks are null ,  it is a sink node!
                              (page, Array(""))
          }).persist()
```

The total number of documents is counted after this step , by a simple count operation on processedPages.

The Page Rank calculation in Hadoop was done using two sets of Mappers and Reducers , first to calculate the initial page rank and the sink sum which was the sum of the dangling nodes and the other pair was used for iteratively running on top of this data and received a key value pair of (page_name,(outlinks and page_rank)). The same is achieved here with just 3 complex operations :

```
val initPR = 1.0 / totalDocs
var pages = processedPages.map(page => (page._1, (page._2, initPR))).persist() // initial graph
```

In iteration :

```
{
var dangSum = pages.filter(x => {
                        val (page, (outlinks, pr)) = x
                        (outlinks(0) == "")})
                .map(x => { val (page, (outlinks, pr)) = x
                         pr})
                .sum() // tracks sum of pr of angling nodes , much more efficient than counters and
double to long conversions

    // filter non dangling nodes
    // divide the page rank to each of the outlinks
    // sum based on the outlink(page) as the key,
    // use sum , dangsum and the formula to get the final answer
    var pageRank = pages.filter(x => {
                      val (page, (outlinks, pr)) = x
                      !(outlinks(0) == "")})
                .flatMap(x => {
                      val (page, (outlinks, pr)) = x
                      outlinks.map(outlink => (outlink, pr / outlinks.size)) })
                .reduceByKey((accum, one_pr) => accum + one_pr)
```

```
                .map(x => {
                    val (k,v) = x
                    (k, (((1-lambda) / totalDocs) + (lambda*v) +
(lambda*dangSum/totalDocs)))}).persist()

    pages.unpersist()

    // join to get the outlinks back for next iteration
    pages = processedPages.join(pageRank).persist()

}
```

And finally the top k job was implemented in Hadoop using a mapper and reducer , where the mapper emitted top 100 records for each of it's call and the reducer then took the top 100 from the cumulative top 100 of the mappers. But in scala we do it in a 2 step process where the data is sorted , take 100 and then mapped to printable format , parallely:

```
val sortPR = pages.sortBy(_._2._2, false)
                .take(100)
                .map( x => {
                  val (page, (outlinks, pr)) = x
                  page + "\t" + pr})

   val top100 = sc.parallelize(sortPR)

   top100.saveAsTextFile(args(1))
```

Q. 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.

There are several differences between the two approaches .

Spark manages shuffling better than hadoop as Spark loads all the data into memory as compared to Hadoop where it has a threshold to loading data into memory. Spark relies on OS for such thresholds. Executing batch processing jobs in Spark is about 10 to 100 times faster than the Hadoop MapReduce framework just by merely cutting down on the number of reads and writes to the disc.

Spark API extends it's functionality in terms of the RDD which is very powerful in terms of parallel computation , is fault tolerance in terms of node failures and has overall a good efficiency.

Using Spark in Scala we can execute more logic using a fewer lines of code as compared to MapReduce in Java. and is thus less verbose.

Spark had several optimizations for Page Rank by providing RDD (which helped us parallely operate on the graph) and the lazy approach worked in favour of reducing the overall processing time. It also provided the option to keep the data which was used again and again in the memory , using persist option which fasten the execution.

Since Spark advantages are mostly related to loading the data into memory and fastening the overall process. Thus , the memory in the Spark cluster should be at least as large as the amount of data we need to process, because the data has to fit into the memory for optimal performance. So, if we need to process really Big Data, Hadoop will be a cheaper option since Hadoop uses hard disk more than the memory and hard disk space comes at a much lower price.

But Spark can be cost effective in the terms of hardware used. Since spark requires less computational time, it can be cheaper in a cloud based environment such as EMR, where compute power is paid per use.

Source :berkely.edu , quora and youtube videos.

Performance Comparison

Speed Up = Average Time taken by 6 machines / Average Time taken by 11 machines

In the ideal scenario since the worker machines are doubled, the speed up should be close to 2. But practically a speed up close to 1.5 is acceptable.

Jobs	6 m4.large	11 m4.large
Spark Application	31 minutes	12 minutes
MapReduce Application	36 minutes	30 minutes

Speed up for Spark Application = 31/12 = 2.583

Spark Execution seems faster. The reason for that can be the features spark provides on top of hadoop platform which was missing in the Hadoop implementations. Some of which were used in the code were :

- Spark takes MapReduce to the next level with less expensive shuffles in the data processing. With capabilities like in-memory data storage and near real-time processing, this was achieved using the persist calls for the data that was required again and again.

- Spark also supports lazy evaluation of big data queries, which helps with optimization of the steps in data processing workflows. Which means a statement is not executed where it is declared but when it's value is actually being used.

- As per the spark documentation spark holds intermediate results in memory rather than writing them to disk which is very useful especially when we need to work on the same dataset multiple

times. Also spark supports both in-memory and on-disk executions. the operators perform external operations when the data is larger than the memory.

- Spark attempts to store as much as data in memory as much possible and then saves the rest of the data to disk.

- And needless to say it provides a concise APIs and requires lot less of syntactic code , thanks to functional programming achieved via Scala. As Spark is written in Scala Programming Language and runs on Java Virtual Machine (JVM) environment.

Q. The top 100 links are attached with the report.

The output from both the execution seems identical and no a little variation in page rank scores can be because of the conversion less operations in spark compared to manual conversions done in map reduce to save the sink sum / dang sum in the job counters. The output is attached .