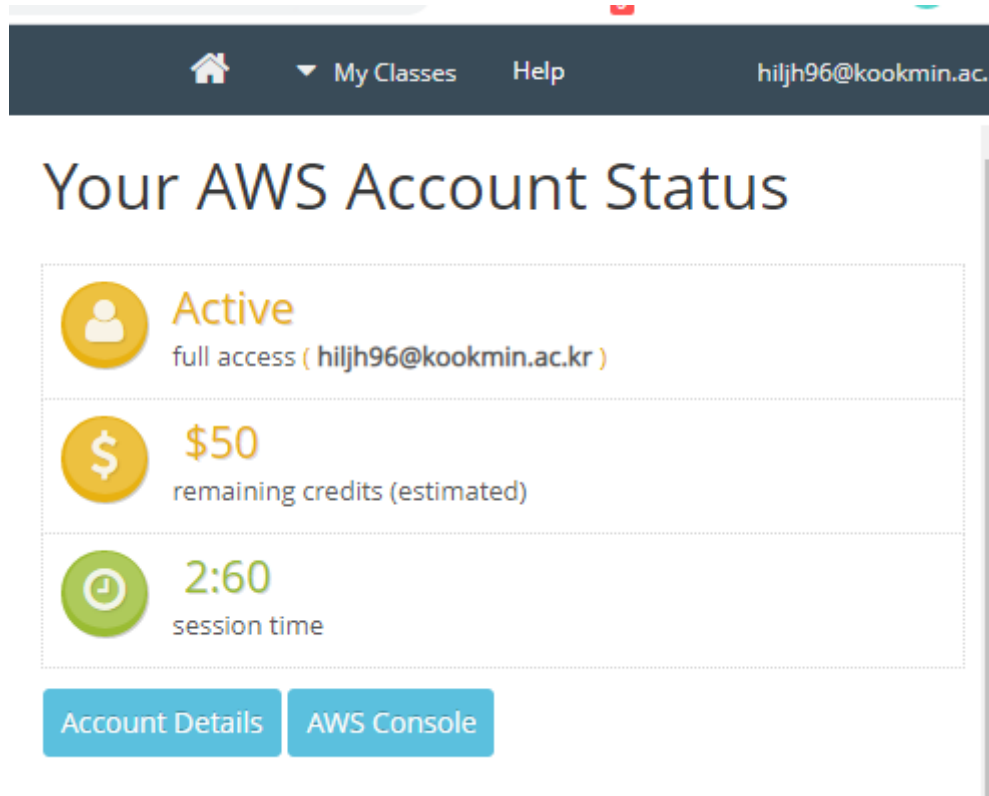


20152850 이재호 exam

주의 사항 및 HADOOP 설정

주의사항



The screenshot shows the AWS Account Status page for a user with email hiljh96@kookmin.ac.kr. The page has a dark blue header with a home icon, 'My Classes', 'Help', and the email address. The main content area is titled 'Your AWS Account Status' and contains three status cards: 'Active' (full access), '\$50' (remaining credits), and '2:60' (session time). Below these cards are two buttons: 'Account Details' and 'AWS Console'.

Status	Value	Description
Active	full access	(hiljh96@kookmin.ac.kr)
\$50	remaining credits	(estimated)
2:60	session time	

EMR 설정 및 Hadoop

ec2 key pair 및 test.pem chmod 설정

```
student@DESKTOP-T57KLS0:/mnt/d/workspace/bigfinal$ chmod 600 test.pem
student@DESKTOP-T57KLS0:/mnt/d/workspace/bigfinal$ ls -l
total 4
-rw----- 1 student student 1678 Dec  5 10:39 test.pem
student@DESKTOP-T57KLS0:/mnt/d/workspace/bigfinal$ |
```

EMR 연결


```
scala> val dataRdd = input.map(line => line.split("\t"))
dataRdd: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[2] at map at <console>:25

scala> dataRdd.take(15)
res1: Array[Array[String]] = Array(Array(0, 1), Array(0, 2), Array(0, 3), Array(0, 4), Array(0, 5),
, Array(0, 6), Array(0, 7), Array(0, 8), Array(0, 9), Array(0, 10), Array(0, 11), Array(0, 12), Ar
ray(0, 13), Array(0, 14), Array(0, 15))
```

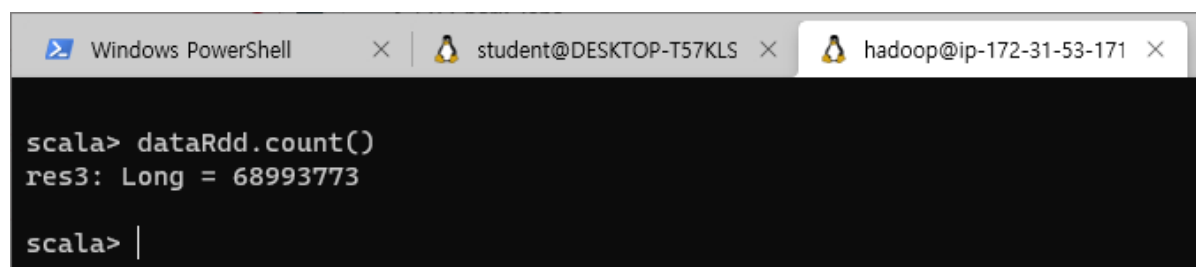
테스트 데이터 만들기

```
student@DESKTOP-T57KLS0:/mnt/d/workspace/bigfinal$ head -n 1000 bigdata-input.txt > out1.txt
student@DESKTOP-T57KLS0:/mnt/d/workspace/bigfinal$ tail -n 1000 bigdata-input.txt > out2.txt
student@DESKTOP-T57KLS0:/mnt/d/workspace/bigfinal$ cat out1.txt out2.txt > data.txt
student@DESKTOP-T57KLS0:/mnt/d/workspace/bigfinal$ ls -l
total 1055364
-rw-r--r-- 1 student student 1080597849 Dec  3 12:16 bigdata-input.txt
-rw-r--r-- 1 student student    26507 Dec  5 20:07 data.txt
-rw-r--r-- 1 student student    9604 Dec  5 12:26 out1.txt
-rw-r--r-- 1 student student   16903 Dec  5 20:06 out2.txt
-rw-r--r-- 1 student student    26507 Dec  5 12:27 output.txt
-rw----- 1 student student    1678 Dec  5 10:39 test.pem
```

bigdata-input의 상위 1000개 값과 하위 1000개 값을 사용했다.

스파크 프로그래밍

1. 임의의 출발지에서 목적지로 향하는 전체 기록의 횟수



```
scala> dataRdd.count()
res3: Long = 68993773

scala> |
```

dataRdd는 입력한 파일이다.

2. 출발지의 총 id 갯수를 세는 스파크 프로그램

```
val startLoc = dataRDD.map( loc => loc(0) )
val distStartLoc = startLoc.distinct()
distStartLoc.collect().foreach(println)
distStartLoc.count()
```

distinct()를 사용 했다.

```
Windows PowerShell | student@DESKTOP-T57KLS | hadoop@ip-172-31-53-171 | + | - | □ | ×
res0: Array[String] = Array(0 1)

scala> val dataRdd = input.map(line => line.split("\t"))
dataRdd: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[2] at map at <console>:25

scala> dataRdd.take(15)
res1: Array[Array[String]] = Array(Array(0, 1), Array(0, 2), Array(0, 3), Array(0, 4), Array(0, 5),
Array(0, 6), Array(0, 7), Array(0, 8), Array(0, 9), Array(0, 10), Array(0, 11), Array(0, 12), Ar
ray(0, 13), Array(0, 14), Array(0, 15))

scala> val startLoc = dataRdd.map(loc => loc(0))
startLoc: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at map at <console>:25

scala> val distStartLoc = startLoc.distinct()
distStartLoc: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[6] at distinct at <console>:25

scala> distStartLoc.count()
res2: Long = 4308452
```

3. 목적지의 총 id 갯수를 세는 스파크 프로그램

```
val destLoc = dataRDD.map(loc => loc(1))
val distDestLoc = destLoc.distinct()
distStartLoc.collect().foreach(println)
distStartLoc.count()
```

```
Windows PowerShell | student@DESKTOP-T57KLS | hadoop@ip-172-31-53-171 | + | - | □ | ×
^

scala> val destLoc = dataRdd.map(loc => loc(1))
destLoc: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[7] at map at <console>:25

scala> distDestLoc = destLoc.distinct()
<console>:27: error: not found: value distDestLoc
val $ires6 = distDestLoc
              ^

<console>:25: error: not found: value distDestLoc
    distDestLoc = destLoc.distinct()
    ^

scala> val distDestLoc = destLoc.distinct()
distDestLoc: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[10] at distinct at <console>:25

scala> distDestLoc.count()
res4: Long = 4489240

scala> |
```

4. 출발지 중에서 목적지로 향하는 횟수가 가장 많은 출발지 ID와 횟수 출력

```
val startOne = startLoc.map(s => (s,1))
startOne.take(10).foreach(println)
val startCounter = startOne.reduceByKey((x,y) => (x+y))
startCounter.takeOrdered(1)(Ordering[Long].reverse.on(x=>x._2))
```

reduceByKey를 사용 했다.

```
Windows PowerShell | student@DESKTOP-T57KLS | hadoop@ip-172-31-53-171 |
scala> val startOne = startLoc.map(s=>(s,1))
startOne: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[11] at map at <console>:25

scala> startOne.take(1)
res5: Array[(String, Int)] = Array((0,1))

scala> val startCounter = startOne.reduceByKey((x,y)=>(x+y))
startCounter: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[12] at reduceByKey at <console>:25

scala> startCounter.takeOrdered(1)(Ordering[Long].reverse.on(x=>x._2))
res6: Array[(String, Int)] = Array((10009,20293))

scala> |
```

5. 목적지로 들어오는 출발지 아이디어가 가장 많은 목적지 아이디어 및 해당 목적지로 도착하는 출발지 아이디어의 갯수

```
val destOne = destLoc.map(d => (d,1))
destOne.take(10)
val destCounter = destOne.reduceByKey((x,y) => (x+y))
destCounter.takeOrdered(3)(Ordering[Long].reverse.on(x=>x._2))
```

```
Windows PowerShell | student@DESKTOP-T57KLS | hadoop@ip-172-31-53-171 |
scala> val destOne = destLoc.map(d=>(d,1))
destOne: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[14] at map at <console>:25

scala> val destCounter = destOne.reduceByKey((x,y) => (x+y))
destCounter: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[15] at reduceByKey at <console>:25

scala> destCounter.takeOrdered(1)(Ordering[Long].reverse.on(x=>x._2))
res7: Array[(String, Int)] = Array((10029,13906))

scala> |
```

6. 4,5번을 다른 방식으로 구현하기

4,5번과 달리 자료형이 Int로 저장되어서 Long 대신 Int로 Sort 했다.

```
val startGroup = startOne.groupByKey().mapValues(_.toList)
val stgr = startGr.map(args => (args._1, args._2.sum))
stgr.takeOrdered(1)(Ordering[Int].reverse.on(x=>x._2))
```

```
Windows PowerShell | student@DESKTOP-T57KLS | hadoop@ip-172-31-53-171 |
scala> val startGroup = startOne.groupByKey().mapValues(_.toList)
startGroup: org.apache.spark.rdd.RDD[(String, List[Int])] = MapPartitionsRDD[18] at mapValues at <console>:25

scala> val stctr = startGr.map(args => (args._1, args._2.sum))
<console>:23: error: not found: value startGr
      val stctr = startGr.map(args => (args._1, args._2.sum))
                    ^

scala> val stctr = startGroup.map(args => (args._1, args._2.sum))
stctr: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[19] at map at <console>:25

scala> stctr.takeOrdered(1)(Ordering[Int].reverse.on(x=>x._2))
res8: Array[(String, Int)] = Array((10009,20293))

scala> |
```

```
val destGroup = destOne.groupByKey().mapValues(_.toList)
val destctr = destGroup.map(args => (args._1, args._2.sum))
destctr.takeOrdered(1)(Ordering[Int].reverse.on(x=>x._2))
```

```
Windows PowerShell | student@DESKTOP-T57KLS | hadoop@ip-172-31-53-171 |
scala> stctr.takeOrdered(1)(Ordering[Int].reverse.on(x=>x._2))
res8: Array[(String, Int)] = Array((10009,20293))

scala> val destGroup = destOne.groupByKey().mapValues(_.toList)
<console>:1: error: illegal start of simple expression
val destGroup = destOne.groupByKey().mapValues(_.toList)
                                         ^

scala> val destGroup = destOne.groupByKey().mapValues(_.toList)
destGroup: org.apache.spark.rdd.RDD[(String, List[Int])] = MapPartitionsRDD[22] at mapValues at <console>:25

scala> val destctr = destGroup.map(args => (args._1, args._2.sum))
destctr: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[23] at map at <console>:25

scala> destctr.takeOrdered(1)(Ordering[Int].reverse.on(x=>x._2))
res9: Array[(String, Int)] = Array((10029,13906))

scala> |
```

4, 5번은 reduceByKey()를 사용했다. 파티션 내의 reduceByKey()를 적용하고 셔플되기 때문에 큰 데이터에서 stack overflow나, 데이터 트래픽이 요구되는 정도가 적을 것이다. 반면 6번은 groupByKey로 생성을했다. Iterator[Int]로 생성되는 값들의 크기가 매우 크기 때문에 4,5번 방법에 비해 메모리가 많이 요구되며 이 값이 메모리보다 클 경우 메모리가 overflow되게 된다. 또한 개인적으로 Iterator를 이용한 계산이 익숙하지 않아서 이를 List로 바꾸고 계산했다. 따라서 reduceByKey 방법을 사용하는게 6번 방법보다 적절할 것 같다.

7. S3를 이용해 1번 문제를 풀어 보기


```
Windows PowerShell
student@DESKTOP-T57KLS0: /i
hadoop@ip-172-31-53-171:~

scala> val lines = sc.textFile("s3://hiljh96-bigdata-input/data.txt")
lines: org.apache.spark.rdd.RDD[String] = s3://hiljh96-bigdata-input/data.txt MapPartitionsRDD[1] at textFile at <console>:24

scala> lines.take(3)
res0: Array[String] = Array(0 1, 0 2, 0 3)

scala> val iter = 3
iter: Int = 3

scala> |
```

데이터는 테스트 데이터셋을 만들어서 사용했다. 이를 s3에 업로드해서 가져왔다.

```
Windows PowerShell
student@DESKTOP-T57KLS0: /i
hadoop@ip-172-31-53-171:~

iter: Int = 3

scala> val roads = lines.map(s => {
  | val splited = s.split("\t")
  | (splited(0), splited(1))
  | })
roads: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD[2] at map at <console>:25

scala> roads.take(10)
res1: Array[(String, String)] = Array((0,1), (0,2), (0,3), (0,4), (0,5), (0,6), (0,7), (0,8), (0,9), (0,10))

scala> |
```

roads 변수에 lines로 읽어온 데이터를 넣는다.

roads에 데이터를 넣을 때 val splited라는 변수에 map을 이용해 s(0) 시작 주소 s(1) 도착 주소로 split("\t")로 나뉜다. 이를 roads의 (0)과 (1)에 할당한다.

```
scala> val distinctRoads = roads.distinct()
distinctRoads: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD[5] at distinct at <console>:25

scala> distinctRoads.take(10)
res2: Array[(String, String)] = Array((2,1866608), (10,275866), (1,82854), (4847391,4847391), (4847359,4847363), (4847481,4846944), (4847368,4847542), (2,996011), (6,173959), (4847289,4846094))
```

distinct로 roads의 unique 값을 구한다. 1번문제와 동일한 구조이다.

```
Windows PowerShell
student@DESKTOP-T57KLS0: /i
hadoop@ip-172-31-53-171:~

scala> val groupedRoads = distinctRoads.groupByKey()
groupedRoads: org.apache.spark.rdd.RDD[(String, Iterable[String])] = ShuffledRDD[6] at groupByKey at <console>:25

scala> groupedRoads.take(10)
res3: Array[(String, Iterable[String])] = Array((4847528,CompactBuffer(4847219)), (4847324,CompactBuffer(4846204)), (4847454,CompactBuffer(4847462, 4847465, 4846893, 4847466, 4847468, 4847469, 4847464, 4847453, 4847463, 4847461, 4847455, 4847456, 4847467, 4847458, 4847460, 4847459, 4847457)), (4847515,CompactBuffer(4847104, 763407)), (4847287,CompactBuffer(4846035)), (4847397,CompactBuffer(4846544)), (4847432,CompactBuffer(4846733)), (4847283,CompactBuffer(550575, 4846016, 4847283, 258767, 285234, 88340, 4847282, 367146, 438377)), (4847467,CompactBuffer(4847469, 4847468, 4847462, 4847455, 4847453, 4847460, 4847454, 4847457, 4847464, 4847458, 4847463, 4847471, 4847456, 4847466, 4847465, 4846893, 4847459, 4847461)), (4847261,CompactBuffer(4847261, 4845886, 1043521, 1405366)))

scala> |
```

GroupByKey로 key ID 값(출발점) Value(도착지들)이 들어가게 된다.

key값은 출발점이며, Value는 도착지들의 Iterable[String] 즉 여러개이다.

```
scala> var results = roads.mapValues(v=>1.0)
results: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[7] at mapValues at <console>:25

scala> results.take(10)
res4: Array[(String, Double)] = Array((0,1.0), (0,1.0), (0,1.0), (0,1.0), (0,1.0), (0,1.0), (0,1.0), (0,1.0), (0,1.0), (0,1.0))
```

mapValue를 이용해 key 값은 ID 뒤는 1.0으로 설정한다.

```
scala> for (i <- 1 to iters) {
  | val startTime = System.currentTimeMillis()
  | val weights = groupedRoads.join(results).values.flatMap { case(a,b) =>
  | a.map(x=>(x,b / a.size))
  | }
  | results = weights.reduceByKey((a,b) => a+b).mapValues(y=>0.1+0.9*y)
  | val interimResult = results.take(10)
  | interimResult.foreach(record => println(record._1 + " : " + record._2))
  | val endTime = System.currentTimeMillis()
  | println(i + " th iteration took " + (endTime - startTime)/1000 + " seconds")
  | }
```

위의 startTime은 현재 시간을 System에서 가져온다

val weights는 groupedRoads 즉 id, value(도착지들) 에 1.0을 합쳐 string iterable[string] double(1.0)으로 만든다. 그예시는 아래와 같다.

```
val wt0 = groupedRoads.join(results)
wt0.take(10)
```

▶ (1) Spark Jobs

```
wt0: org.apache.spark.rdd.RDD[(String, (Iterable[String], Double))] = MapPartitionsRDD[901] at flatMap at command-4410902002533013:1
res20: Array[(String, (Iterable[String], Double))] = Array((4847469,0.570441687873609), (4847468,0.570441687873609), (4847462,0.570441687873609), (4847455,0.570441687873609), (4847453,0.570441687873609), (4847460,0.570441687873609), (4847454,0.570441687873609), (4847457,0.570441687873609), (4847464,0.570441687873609), (4847458,0.570441687873609))
```

이 value 값들을 map하게 되는데 x,b와 a.size 즉 도달한 사이트의 개수를 더한다 1.0은 초기에 설정된 값이며 이 반복문이 반복될때 마다 값이 변하며 이 코드의 가중치 역할을 한다. 이 값을 map으로 roads에 계속해서 붙여준다.

```
val wt = groupedRoads.join(results).values.flatMap { case(a, b) =>
  a.map(x=> (x, b / a.size))
}
wt.take(10)
```

▶ (1) Spark Jobs

```
wt: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[901] at flatMap at command-4410902002533012:1
res19: Array[(String, Double)] = Array((4847469,0.570441687873609), (4847468,0.570441687873609), (4847462,0.570441687873609), (4847455,0.570441687873609), (4847453,0.570441687873609), (4847460,0.570441687873609), (4847454,0.570441687873609), (4847457,0.570441687873609), (4847464,0.570441687873609), (4847458,0.570441687873609))
```

Command took 0.59 seconds -- by hiljh96@kookmin.ac.kr at 2020. 12. 5. 오후 9:12:43 on e

results 구문은 가중치를 계산하는 구문이다. 페이지에 가중치를 적용해 page 랭크를 설정한다.

```
results = weights.reduceByKey((a,b) => a+b).mapValues(y=>0.1+0.9*y)
```

```
Array[(String, Double)] = Array((4846893,15.805381004875022), (4847462,10.376088539619698), (4847469,0.570441687873609), (4847468,0.570441687873609), (4847462,0.570441687873609), (4847455,0.570441687873609), (4847453,0.570441687873609), (4847460,0.570441687873609), (4847454,0.570441687873609), (4847457,0.570441687873609), (4847464,0.570441687873609), (4847458,0.570441687873609))
```

아래 구문은 계산과정의 시간을 구하는 코드이다. 출력값은 아래와 같다.

```
val interimResult = results.take(10)
interimResult.foreach(record => println(record._1 + " : " + record._2))
val endTime = System.currentTimeMillis()
```

```
887243 : 0.9999999999999997
57226 : 0.9999999999999992
2514877 : 1.0000000000000001
1416271 : 0.9999999999999997
136130 : 0.99999999999999978
373791 : 0.9999999999999987
4845917 : 1.0
84671 : 0.9999999999999987
4847173 : 2.8000000000000003
4846666 : 1.0
1 th iteration took 13 seconds
```

10. 작성한 코드는 어떤 알고리즘, 목적

작성한 코드는 page rank에 쓰이는 코드이다. 관련된 알고리즘은 reverse web link graph이다. 웹페이지의 target, source를 입력으로 받아 각 타겟별로 소스 주소들을 연결한다. weight는 여기서 rank에 해당한다. 이 알고리즘은 PageRank 등수에 따라 정렬을 해놔 검색어가 포함된 페이지를 순위별로 나열하기 위해 쓰인다. 이 알고리즘은 재귀적인 특징을 가진다. 페이지 랭크는 더 중요한 페이지는 더 많은 다른 사이트로부터 링크를 받는다는 관찰에 기초한다. 페이지 랭크는 페이지간 페이지 랭크 값을 주고 받는 것을 반복하다 전체 웹페이지가 특정한 페이지 랭크 값을 수렴하는 것을 통해 각 페이지의 최종 페이지 랭크를 계산한다.

11. 위 프로그램 실행 장면 캡처

```

val interimResult = results.take(10)
interimResult.foreach(record => println(record._1 + " : " + record._2))
val endTime = System.currentTimeMillis()
println(i + " th iteration took " + (endTime - startTime)/1000 + " seconds")
}
887243 : 0.9999999999999987
57226 : 0.9999999999999992
2514877 : 1.0000000000000001
1416271 : 0.9999999999999997
136130 : 0.9999999999999978
373791 : 0.9999999999999987
4845917 : 1.0
84671 : 0.9999999999999987
4847173 : 2.8000000000000003
4846666 : 1.0
1 th iteration took 18 seconds
887243 : 0.1262499999999997
57226 : 0.14621621621621622
2514877 : 0.10947368421052632
1416271 : 0.1610714285714287
136130 : 0.11199999999999999
373791 : 0.12478260869565216
4846301 : 0.895
84671 : 0.12478260869565216
4847173 : 1.4500000000000002
4847362 : 12.543994505494508
2 th iteration took 0 seconds
887243 : 0.10296525135869565
57226 : 0.11742147552958364
2514877 : 0.10263409610983981
1416271 : 0.1097112566704575
136130 : 0.10135554347826087
373791 : 0.10394990548204158
4846301 : 0.86125
84671 : 0.10394990548204158
4847173 : 0.3475000000000003
4847362 : 8.647612178480857
3 th iteration took 0 seconds

```

12. 위 코드 실행 후에 results 변수에 들어 있는 원소 중 큰 값을 가지는 상위 10개 아이디와 그 값을 출력

```
results.takeOrdered(10)(Ordering[Double].reverse.on(x=>x._2))
```

jar를 만들 때는 foreach.(println)을 사용했습니다.

```

scala> results.takeOrdered(10)(Ordering[Double].reverse.on(x=>x._2))
res9: Array[(String, Double)] = Array((4846893,15.805381004875022), (4847462,10.376088539619698), (4847466,10.376088539619698), (4847459,10.341648400523455), (4847454,10.341648400523454), (4847467,10.267950381724964), (4847460,10.233656868926296), (4847468,10.103731830964577), (4847457,9.663673367639978), (4847461,9.153425965603729))

```

기본으로 실행한 결과 다음과 같이 나타났다. 데이터는 2000개를 사용한 데이터셋을 사용했다. 위 코드의 실행 중 성능을 향상시키기 위해서는 Worker의 수를 늘릴 필요가 있다. 파티션을 더 많이 분할해서 사용하면 대규모의 데이터를 처리하는데 성능이 증가하기 때문이다. 이와 더불어 기존의 데이터의 iterations의 사이즈가 너무 크기 때문에 메모리에서 이를 인식하지 못해 실행이 안되는 것 같다. 이를 해결 하기 위해서는 할당된 메모리의 사이즈를 높여야 한다.

S3에서 스파크처럼 파일을 분할처리 하는 방법을 잘 모르겠다. 만약 S3에서 파일을 직접 core 개수 만큼 분할해서 s3에 업로드 안뒤 이를 각각 `sc.textFile("fname1~n")` n은 코어 노드 개수 만큼 나눠서 파일을 분할 저장 한 뒤 실행하면 task의 속도가 훨씬 빨라 질 것이다. 아래의 예시이면 데이터 분석에 쓰일 데이터를 4분할 한 뒤에 사용하면 될 것 이다.

Node type	Instance type	Instance count	Purchasing option
Master Master - 1	m4.large 2 vCore, 8 GiB memory, EBS only storage EBS Storage: 32 GiB Add configuration settings	1 Instances	<input checked="" type="radio"/> On-demand <input type="radio"/> Spot Use on-demand as max price
Core Core - 2	m4.large 2 vCore, 8 GiB memory, EBS only storage EBS Storage: 32 GiB Add configuration settings	4 Instances	<input checked="" type="radio"/> On-demand <input type="radio"/> Spot Use on-demand as max price
Task Task - 3	m5.xlarge 4 vCore, 16 GiB memory, EBS only storage EBS Storage: 64 GiB Add configuration settings	0 Instances	<input checked="" type="radio"/> On-demand <input type="radio"/> Spot Use on-demand as max price

[+ Add task instance group](#)

Total core and task units 4 Total units

Cluster scaling

Adjust the number of Amazon EC2 instances available to an EMR cluster via EMR-managed scaling or a custom automatic scaling policy. [Learn more](#)

Cluster scaling ☐ Enable Cluster Scaling

EBS Root Volume

Specify the root device volume size up to 100 GiB. This sizing applies to all instances in the cluster. [Learn more](#)

Root device EBS volume size 64 GiB

core 노드와 ram을 증가 시켜 map reduce 성능을 높이는 시도를 했다. 시간이 약간 감소한 것이 확인 가능하다.

```
scala> for (i <- 1 to iters) {
  | val startTime = System.currentTimeMillis()
  | val weights = groupedRoads.join(results).values.flatMap { case(a,b) =>
  |   a.map(x=>(x,b / a.size))
  | }
  | results = weights.reduceByKey((a,b) => a+b).mapValues(y=>0.1+0.9*y)
  | val interimResult = results.take(10)
  | interimResult.foreach(record => println(record._1 + " : " + record._2))
  | val endTime = System.currentTimeMillis()
  | println(i + " th iteration took " + (endTime - startTime)/1000 + " seconds")
  | }
887243 : 0.9999999999999987
57226 : 0.9999999999999992
2514877 : 1.0000000000000001
1416271 : 0.9999999999999997
136130 : 0.9999999999999978
373791 : 0.9999999999999987
4845917 : 1.0
84671 : 0.9999999999999987
4847173 : 2.8000000000000003
4846666 : 1.0
1 th iteration took 13 seconds
887243 : 0.12624999999999997
57226 : 0.14621621621621622
2514877 : 0.10947368421052632
1416271 : 0.1610714285714287
136130 : 0.11199999999999999
373791 : 0.12478260869565216
4846301 : 0.895
84671 : 0.12478260869565216
4847173 : 1.4500000000000002
```

하둡에서의 시도 (기존 하둡에서는 실행이 되지 않았다.)

```
FSCK started by hadoop (auth:SIMPLE) from /172.31.57.228 for path /dataset/bigdata-input.txt at Sat Dec 05 13:
22:29 UTC 2020
Status: HEALTHY
Total size: 1080597849 B
Total dirs: 0
Total files: 1
Total symlinks: 0
Total blocks (validated): 9 (avg. block size 120066427 B)
Minimally replicated blocks: 9 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 0 (0.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 2
Average block replication: 2.0
Corrupt blocks: 0
Missing replicas: 0 (0.0 %)
Number of data-nodes: 4
Number of racks: 1
FSCK ended at Sat Dec 05 13:22:29 UTC 2020 in 12 milliseconds
```

하둡에서 블록의 수가 9개인 것이 확인 가능하다. 기존 실습보다 더 빠르게 진행될 것 같으나 너무 오래 걸렸다.

13. iters값 변화시 값의 변화

iters 값이 커지면 값이 점점 감소하다 1.0 근처로 수렴할 것이다. 왜냐하면 웹페이지에서 접근하려는 source, target을 기반으로 구축된 알고리즘 이기 때문이다. 또한 이 알고리즘은 수렴하게 구현되어 있다. 랭크 값을 주고 받다 보면 전체 웹페이지가 특정한 페이지 랭크 값으로 수렴하는 것을 찾기 위함이다. 이를 통해 최종 페이지의 랭크를 계산한다.

클라우드에서 스파크 프로그램 활용

14. spark-submit 사용하기

add step으로 jar 생성 추가하기

이전단계에서 생성했던 s3 폴더를 사용했다.

Step typeCustom JAR

Name*Custom JAR

JAR location*s3://hijh96-bigdata-input/

JAR location rclass in the cl

Clone

Terminate

AWS CLI export

Cluster: My cluster Waiting Cluster ready to run steps.

Summary

Application user interfaces

Monitoring

Hardware

Configurations

Events

Steps

E

Concurrency: 1 [Change](#)

After last step completes: Cluster waits

Add step

Clone step

Cancel step

View Jobs in the A

Filter: All steps

Filter steps ...

1 step (all loaded)

	ID	Name	Status	Start time (UTC+9)	Elapsed time
	s-1V8U1ZJAIDFZM	Custom JAR	Pending		--

add step으로 jar 생성 추가하기

hadoop에 vi 로 코드 만들기

java로 하둡을 쓰는게 익숙하지 않아 잘 되지 않았다.

