Class – CS6240 Fall-2018 Sec 2
HW – 1
Name- Mustafa Kapadia
Github - https://github.ccs.neu.edu/cs6240f18/mustafa8895/tree/master/HW2

Spark Combining

Reduce-By-Key
Input = Text file (edges.csv)
***val** counts = map (Line L from input => mapBasedOnFile (L))*
.reduceByKey ((x, y) = x+y)

mapBasedOnFile(args: String) : (String, Int)
***emit** (args.substring(args.indexOf(**","**)+1), 1)*

---

(40) ShuffledRDD[3] at reduceByKey at twitterFollowers.scala:27 []
+-(40) MapPartitionsRDD[2] at map at twitterFollowers.scala:26 []
| input MapPartitionsRDD[1] at textFile at twitterFollowers.scala:25 []
| input HadoopRDD[0] at textFile at twitterFollowers.scala:25 []

Aggregate-By-Key
Input = Text file (edges.csv)
***val** counts = map (Line L from input => mapBasedOnFile (L))*
.aggregateByKey (initial= 0 , combiningFunc=(_+_), reduceFunc=(_+_))

mapBasedOnFile(args: String) : (String, Int)
***emit** (args.substring(args.indexOf(**","**)+1), 1)*

---

(40) ShuffledRDD[3] at aggregateByKey at twitterFollowers.scala:27 []
+-(40) MapPartitionsRDD[2] at map at twitterFollowers.scala:26 []
| input MapPartitionsRDD[1] at textFile at twitterFollowers.scala:25 []
| input HadoopRDD[0] at textFile at twitterFollowers.scala:25 []

Fold-By-Key
Input = Text file (edges.csv)
***val** counts = map (Line L from input => mapBasedOnFile (L))*
.foldByKey (initial= 0 , reduceAndCombineFunc=(_+_))

mapBasedOnFile(args: String) : (String, Int)
***emit** (args.substring(args.indexOf(**","**)+1), 1)*

---

(40) ShuffledRDD[3] at foldByKey at twitterFollowers.scala:27 []
+-(40) MapPartitionsRDD[2] at map at twitterFollowers.scala:26 []
| input MapPartitionsRDD[1] at textFile at twitterFollowers.scala:25 []
| input HadoopRDD[0] at textFile at twitterFollowers.scala:25 []

Group-By-Key

Input = Text file (edges.csv)

```
val counts = map ( Line L from input => mapBasedOnFile ( L ) )
                 .groupByKey()
         .map(for key k, sum all values)

mapBasedOnFile(args: String) : (String, Int)
  emit (args.substring(args.indexOf(",")+1), 1)
```

```
(40) MapPartitionsRDD[4] at map at twitterFollowers.scala:28 []
 |  ShuffledRDD[3] at groupByKey at twitterFollowers.scala:27 []
 +-(40) MapPartitionsRDD[2] at map at twitterFollowers.scala:26 []
    |  input MapPartitionsRDD[1] at textFile at twitterFollowers.scala:25 []
    |  input HadoopRDD[0] at textFile at twitterFollowers.scala:25 []
```

Data-Set

Input = Text file (edges.csv)
Load input into dataset d with column names to and from
d.groupBy(to).count()

```
== Parsed Logical Plan ==
Aggregate [to#15], [to#15, count(1) AS count#31L]
+- AnalysisBarrier
      +- Project [_c0#10 AS from#14, _c1#11 AS to#15]
         +- Relation[_c0#10,_c1#11] csv

== Analyzed Logical Plan ==
to: string, count: bigint
Aggregate [to#15], [to#15, count(1) AS count#31L]
+- Project [_c0#10 AS from#14, _c1#11 AS to#15]
   +- Relation[_c0#10,_c1#11] csv

== Optimized Logical Plan ==
Aggregate [to#15], [to#15, count(1) AS count#31L]
+- Project [_c1#11 AS to#15]
   +- Relation[_c0#10,_c1#11] csv

== Physical Plan ==
*(2) HashAggregate(keys=[to#15], functions=[count(1)], output=[to#15, count#31L])
+- Exchange hashpartitioning(to#15, 200)
   +- *(1) HashAggregate(keys=[to#15], functions=[partial_count(1)], output=[to#15,
count#37L])
      +- *(1) Project [_c1#11 AS to#15]
```

+- *(1) FileScan csv [_c1#11] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/Users/mustafa/Desktop/PDP/cs6240f18/HW2/twitter-Spark-DataSet/input/edges..., PartitionFilters: [], PushedFilters: [], ReadSchema: struct<_c1:string>

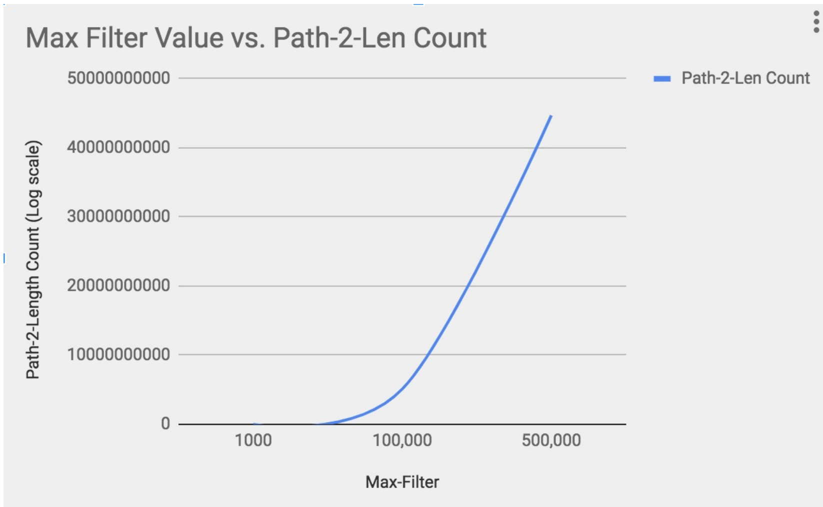
Aggregator	Time Taken	Data Shuffled
Reduce-By-Key	48 sec	116.5 MB
Aggregate-By-Key	72 sec	116.5 MB
Fold-By-Key	78 sec	116.5 MB
Group-By-Key	52 sec	347.6 MB
Data-Set	72 sec	118.2 MB

The numbers above make it clear that all **but** GroupBy perform aggregation before shuffling. GroupBy has the largest amount of data shuffled as no combining takes place whereas the other 4 have a considerably smaller amount of data shuffled.

	RS join input	**RS join shuffled**	**RS join output**	**Rep join input**	**Rep join file cache**	**Rep join output**
	Total cardinality and volume of input	Total cardinality and volume of data sent from Mappers to Reducers	Total cardinality and volume of output	Total cardinality and volume of input	Total cardinality and volume of data broadcast to all machines	Total cardinality and volume of output
Step 1	Cardinality = 85331845 Volume = 1319453657 (edges.csv)	Cardinality = 170663690 Volume = 2047964280 (from logs)	No. of paths= 953138453592 Hence cardinality = 953138453592 Volume= 11,437,661,443,104 (~12 bytes per record)	Cardinality= 85331845 Volume= 1319453657	Cardinality= 3,413,273,800 (Records in edges.csv X 40 mappers)	(Estimated) Cardinality <= 953138453592 (Upper Limit) Number of triangles cannot be more than path 2's Volume <= 11,437,661,443,104
Step 2	Cardinality=953,223,785,437 (edges.csv+output of step 1) Volume = 11,438,980,896,761	Cardinality = 953,223,785,437 Volume = 11,438,980,896,761	Cardinality <= 953138453592 (Upper Limit) Number of triangles cannot be more than path 2's Volume <= 11,437,661,443,104	Merged with step 1	Merged with step 1	Merged with step 1

- Path 2 for reduce side works for the whole input file and gives exact values
- For Replicated it fails due to going over the 5GB limit(Memory)
- Values for path 2 for replicated can be estimated based on those of reduce side
- Replicated was run for 3 max filter sizes
    - 1000 – Triangles = 400494
    - 100000 – Triangles = 5094483211
    - 500000 – Triangles = 44696295329
- It fails for higher values as the 5 GB memory cannot store the hashmap

- It would require disk i/o code to succeed



Pseudo code for cardinality path 2 (Used Reduce Side Join)

---

1. Mapper emits 2 records for each each input
2. Partitioner ensures that data is partitioned based on the node value of StringInt
3. Grouping Comparator ensures data is grouped based on node value of StringInt
4. Key Comparator creates a secondary sort such that data is sorted based on Node and then on Direction
5. Reducer gets an input such that all the to values come before the from values
6. numOfTos keeps a count of the number of To nodes
7. When the iterator reaches the from values it updates the global counter for each from value
8. The global counter contains the number of length 2 paths.

StringInt = (Dir: String, Node: int) // custom Data Type

Map(from, to)
Emit ((“from”, from)), (“to”, to))
Emit((“to”, to), (“from”, from))

Partitioner(key: StringInt value : StringInt numPartitions: int)
Return partition based on Node(key)

GroupingComparatior(o1: StringInt, o2: StringInt)
Return Node(o1) == Node(o2)

Reduce(mid : StringInt, values: Iterable(StringInt))
numOfTos=0
For val in values

```
		If Dir(val) = "to"
			numOfTos++
	else
		GlobalCounter.add(numOfTos)
```

PsuedoCode for Mapreduce Triangles Reduce Side

1. Job 1 uses secondary sort similar to the algorithm above but emits length 2 paths instead of a count
2. Job 2 ensures that completed triangles go to a single reducer which counts them and updates the global counter

---

StringInt = (Dir: String, Node: int) // custom Data Type
JOB : 1
Map1(from , to)

```
	If from and to <= max //max filter
	Emit (("from", from)), ("to", to))
	Emit(("to", to), ("from", from))
```

Partitioner(key: StringInt value : StringInt numPartitions: int)

```
	Return Partition based on Node(Key)
```

GroupingComparatior(o1: StringInt, o2: StringInt)

```
	Return Node(o1) == Node(o2)
```

Reduce1(mid : StringInt, values: Iterable(StringInt))

```
	ListOfTos=[]
	For val in values:
		If Dir(val) = "to"
			ListOfTos.add(Node(val))
	else
		for ToNode in ListOfTos:
			if(ToNode != Node(Val))
				Emit(Node(val), ToNode))
```

JOB 2 :

Map2(from, to) //From edges.csv

```
	If from and to <= max
	Emit ((to, from), "1")
```

Map3(from, to) //Output from Job 1

```
Emit((from, to), "2")

Reduce 2(Edge, values)
    For val in Values:
        If val is of type "1"
        M++
        If val is of type "2"
        N++
    GlobalCounter.Increment(M X N)
```

Pseudocode for map reduce triangles replicated join

---

1. Job 1 filters the edges.csv file based on the max value and broadcasts the result
2. Job 2 converts the broadcast to a hash table. It then counts the number of triangles and updates the global counter

---

```
Job 1:
Map (from, to) //max filter
    If from and to <= max
    Emit(from,to)

Broadcast output
```

```
Job 2:
Setup()
    Create hashmap(Adjacency List) <key, List of values>

Map(from, to)
    For a in adjacency list of to
        For b in adjacency list of a
            If b == from
                Triangles ++

Cleanup()
    Increment_Global_Counter(Triangles)
```

Pseudocode for Spark Reduce Side join with Dataset

1. Filter the input
2. Perform 2 joins to retrieve number of triangles

Convert input to data into dataset(from, to) ds filtered=ds.filter(**"from <=max and to <=max"**)

path2=join filtered.as(**"S1"**) with (filtered.as(**"S2"**),
    where **"S1.to"** = **$"S2.from"** and **"S1.from"** != **$"S2.to"**,
    .select(**"S1.from"**, **"S2.to"**)

triangles = join path2.as(**"S3"**) with (filtered.as(**"S4"**) where **"S3.to"** = **$"S4.from"**
    and **$"S4.to"** = **$"S3.from"**)

numTriangles = triangles.count/3

Pseudocode for Spark Replicated join with Dataset

1. Filter the input
2. Broadcast the filtered dataset
3. Perform 2 joins to retrieve number of triangles

Convert input to data into dataset(from, to) ds

filtered=ds.filter(**from and to <= max**)

broadcasted= broadcast filtered

path2=join filtered.as(**"S1"**) with (broadcasted.as(**"S2"**),
    where **"S1.to"** = **$"S2.from"** and **"S1.from"** != **$"S2.to"**,
    .select(**"S1.from"**, **"S2.to"**)

triangles = join path2.as(**"S3"**) with (broadcasted.as(**"S4"**) where **"S3.to"** = **$"S4.from"**
    and **$"S4.to"** = **$"S3.from"**)
numOfTriangles= triangles.count/3

Pseudocode for Spark Reduce Side join with RDD

1. Filter based on max
2. Self join to get path 2
3. Join again to get path 3 and filter to ensure triangle condition
4. Count triangles

RDD1 = filter(from and to <= max)
RDD2 = RDD1.map((from, to) => (to, from))
Path 2 = RDD2.join(RDD1)
triangles = Path2.join(RDD2)
    .filter((mid, (from,to)) => from==to)

numOfTriangles = triangles.count/3

Pseudocode for Spark Replicated Join with RDD

FilteredRDD = filter(from and to <= max)
Create Hashmap H<From, ListOfTos> from filteredRDD
Broadcast H
Count = FilteredRDD.mapPartitions(findTrio).sum/3

```
FindTrio(from, to)
        For ( path1 in H[to] )
                For( path2 in H[node1]
                        If path2 == from
                                Triangles++
        Return Triangles
```

Outputs Table:

CONFIGURATION	SMALL CLUSTER RESULT	LARGE CLUSTER RESULT
RS Join in MR Max=50000	Time = 45 minutes Triangles= 12029907	Time = 24 minutes Triangles= 12029907
Rep Join in MR Max=70000	Time = 26 minutes Triangles=28282537	Time = 25 minutes Triangles=28282537
RS Join in Spark RDD Max=40000	Time = 47 minutes Triangles = 4741564	Time=36 minutes Triangles= 4741564
RS Join in Spark Dataset Max=75000	Time = 35 minutes Triangles = 34193535	Time = 18 minutes Triangles = 34193535
Rep Join in Spark RDD Max=50000	Time=38 minutes Triangles=12029907	Time=26 minutes Triangles=12029907
Rep Join in Spark Dataset Max=150000	Time=24 minutes Triangles=60464480	Time=23 minutes Triangles=60464480

References:
http://spark.apache.org/docs/latest/sql-programming-guide.html#getting-started -sparksession
https://stackoverflow.com/questions/38111700/chaining-of-mapreduce-jobs - chaining jobs
https://buhrmann.github.io/hadoop-distributed-cache.html - distributed cache