CAS CS 350
Lec 21

Spark (cont.)

1. Overview
    a. RDD → represents dataset (collection of records that is physically distributed in class of machines)
    b. RDDs → support operations
        i. Passed into data operator (as inputs and outputs)
        ii. RDDs are mutable data structures → generates fault tolerance
    c. Spark & MapReduce
        i. Spark supports multiple inputs
        ii. Spark gives more control on intermediate data (specify as the user to keep the data on disk or memory unlike MapReduce), MapReduce stores data in local disk on mapper
        iii. Programming model is different
            1. MapReduce → define map and reduce functions (singular) and may have to implement many different versions
            2. Spark → more intuitive

    d.
    e. All the nodes are data file operators
    f. Data file operator → each instance applies the logic to the joint partitions
    g. Task parallelism → applying different task on the same data
        i. ex) Multi Tread → different tasks on the same data (concurrent task)
    h. Data parallelism → same function, different data
    i. In reality, system breaks down into series of tasks that look like Map and reduce functions
    j. Workers always communicate with the driver/master (so that it can give the output of one worker to another worker as input)
    k. Transformations (map, join, filter, reduce by key) → take as inputs as RDD and produce RDDs and pass down the output to the next operator in the graph
    l. Pipeline allows data to flow and make data progress in parallel
        i. Makes better use of resources
    m. Do we have pipeline in map and reduce?
        i. No, we wait for the mappers to finish and start the reduce phase

n. Transactions/actions → transformations pipeline the output to the next operator and actions output/materialize

o.

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

   i. Point → RDD
   ii. map → transformation
   iii. Gradient → RDD
   iv. For loop → Anonymous function that defines logic of the map
   v. Reduce → action that materializes gradient (because we need to subtract gradient)

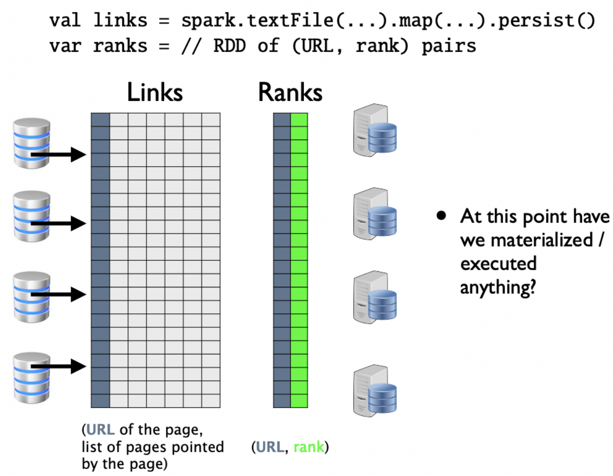
2. PageRank

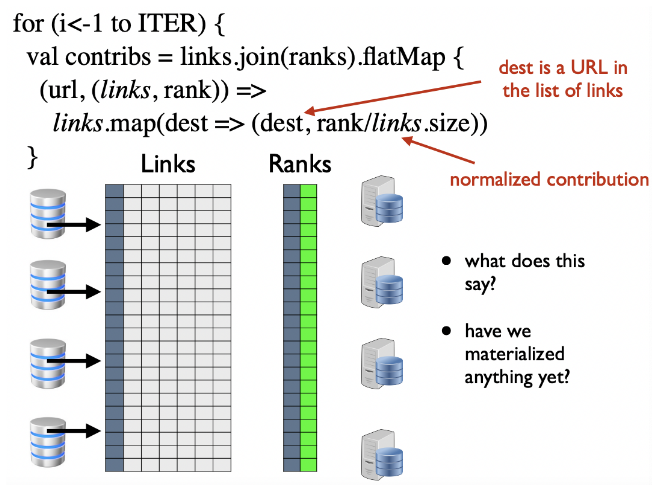
a.

```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
             .mapValues(sum => a/N + (1-a)*sum)
}
```

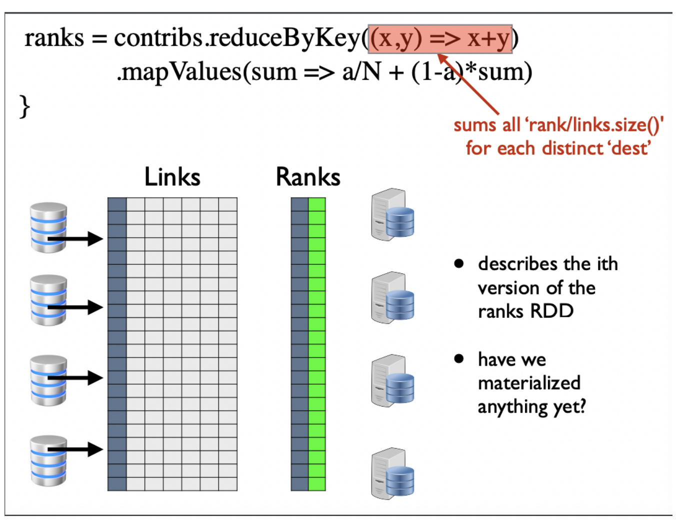
initial rank

b. Links, ranks → RDDs
c. links.join(ranks) → graph with two inputs (links and ranks) that go as input in join(transformation) that generates a new RDD that goes input into flatMap, which also produces an RDD
d. flatMap will apply the logic inside and will generate the records
e. Input format of flatMap → (url, and tuple of (links, rank)), and therefore, the links.join(ranks) should return such format
f. For each record in the collection, apply logic (desk → (dest, rank/links.size))

g. 

h. 

i. Join will join the two tables on the common attribute (URL) and will produce a RDD

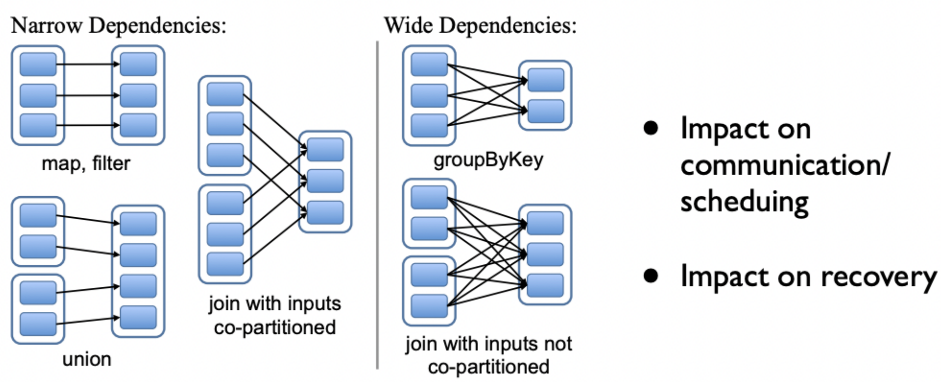
j. 

k. Output of the flatmap is inserted as input to reduceByKey (transformation)

3. What an RDD is composed of
    a.

Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p, parentIters*)	Compute the elements of partition *p* given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

    b. RDD dependencies → which RDD depends on which RDD
    c. This means that if something fails, we know the RDDs that depend on, so we can handle fault tolerance
    d. Dependencies store parent RDDs, and data in case of failure in order to reconstruct the partition
    e.



    f. Dependencies look like the figure above
    g. Wide dependencies in mapReduce
4. Why does an RDD carry metadata on its partitioning?
    a. Each RDD also has some information about the way it is partitioned
    b. Transformations that depend on multiple RDDs know whether they need to shuffle data (wide dependency) or not (narrow) and allows users control locality and reduce shuffles
5. Handling Faults
    a. No replication by default - even when persisted (because it is expensive)
        i. Ram or disk node failure could mean permanent loss of data
    b. Assume heartbeats used to detect lost worker

    i. If worker lost need a way to recompute it’s partitions - will need all dependent partitions
    ii. Recompute if they can’t be found in RAM or on disk
    iii. If wide dependency will need all partitions of dependent RDD if narrow then only one partition
c. So two mechanisms enable recovery from faults: lineage and policy of what partitions to persist (driver and actual replication and partition strategy)