6.824 2020 Lecture 15: Spark

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for
In-Memory Cluster Computing Zaharia et al., NSDI 2012

why are we looking at Spark?
  widely-used for datacenter computations
  generalizes MapReduce into dataflow
  supports iterative applications better than MapReduce
  successful research: ACM doctoral thesis award

three main topics:
  programming model
  execution strategy
  fault tolerance

let's look at page-rank
  here's SparkPageRank.scala from the Spark source repository
  like the code in Section 3.2.2, with more detail

```
val lines = spark.read.textFile("in").rdd
val links1 = lines.map{ s =>
  val parts = s.split("\\s+")
  (parts(0), parts(1))
}
val links2 = links1.distinct()
val links3 = links2.groupByKey()
val links4 = links3.cache()
var ranks = links4.mapValues(v => 1.0)

for (i <- 1 to 10) {
  val jj = links4.join(ranks)
  val contribs = jj.values.flatMap{
    case (urls, rank) =>
      urls.map(url => (url, rank / urls.size))
  }
  ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
}

val output = ranks.collect()
output.foreach(tup => println(s"${tup._1} has rank:  ${tup._2} ."))
```

page-rank input has one line per link, extracted from a big web crawl
  from-url to-url
  the input is vast!

page-rank output is the "importance" of each page
  based on whether other important pages point to it
  really models estimated probability that someone will visit each page
  user model:
    85% chance of following a link from current page
    15% chance of visiting a random page

page-rank algorithm
  iterative, essentially simulates multiple rounds of users clicking links
  ranks (probabilities) gradually converge
  page-rank would be awkward and slow in MapReduce

my example input -- file "in":

```
u1 u3
u1 u1
u2 u3
u2 u2
u3 u1
```

I'll run page-rank in Spark (local machine, not a cluster):

```
./bin/run-example SparkPageRank in 10
u2 has rank:  0.2610116705534049 .
u3 has rank:  0.9999999999999998 .
u1 has rank:  1.7389883294465944 .
```

apparently u1 is the most important page.

let's run some of the page-rank code in the Scala interpreter

```
  ./bin/spark-shell

  val lines = spark.read.textFile("in").rdd
    -- what is lines? does it contain the content of file "in"?
  lines.collect()
    -- lines yields a list of strings, one per line of input
    -- if we run lines.collect() again, it re-reads file "in"
  val links1 = lines.map{ s => val parts = s.split("\\s+"); (parts(0), parts(1)) }
  links1.collect()
    -- map, split, tuple -- acts on each line in turn
    -- parses each string "x y" into tuple ( "x", "y" )
  val links2 = links1.distinct()
    -- distinct() sorts or hashes to bring duplicates together
  val links3 = links2.groupByKey()
    -- groupByKey() sorts or hashes to bring instances of each key together
  val links4 = links3.cache()
    -- cache() == persist in memory
  var ranks = links4.mapValues(v => 1.0)

  -- now for first loop iteration
  val jj = links4.join(ranks)
    -- the join brings each page's link list and current rank together
  val contribs = jj.values.flatMap{ case (urls, rank) => urls.map(url => (url, rank / urls.size)) }
    -- for each link, the "from" page's rank divided by number of its links
  ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
    -- sum up the links that lead to each page

  -- second loop iteration
  val jj2 = links4.join(ranks)
    -- join() brings together equal keys; must sort or hash
  val contribs2 = jj2.values.flatMap{ case (urls, rank) => urls.map(url => (url, rank / urls.size)) }
  ranks = contribs2.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
    -- reduceByKey() brings together equal keys

  -- the loop &c just creates a lineage graph.
  -- it does not do any real work.

  val output = ranks.collect()
    -- collect() is an action.
    -- it causes the whole computation to execute!
  output.foreach(tup => println(s"${tup._1} has rank:  ${tup._2} ."))
```

until the final collect(), this code just creates a lineage graph
  it does not process the data

what does the lineage graph look like?
  Figure 3
  it's a graph of transform stages -- a data-flow graph
  it's a complete recipe for the computation
  note that the loop added to the graph -- there is not actually a cycle
    there's a *new* ranks/contribs for each loop iteration

for multi-step computation, this programming model is more convenient than MapReduce

the Scala code runs in the "driver" machine of Figure 2
  the driver constructs a lineage graph
  the driver compiles Java bytecodes and sends them to worker machines
  the driver then manages execution and data movement

what does the execution look like?
  [diagram: driver, partitioned input file, workers]
  * input in HDFS (like GFS)
  * input data files are already "partitioned" over many storage servers
    first 1,000,000 lines in one partition, next lines in another, &c.
  * more partitions than machines, for load balance
  * each worker machine takes a partition, applies lineage graph in order
  * when computation on different partitions is independent ("narrow"):
    no inter-machine communication required after first read
    a worker applies series of transformations to input stream

this is already more efficient than MapReduce
  data is forwarded directly from one transformation to the next
  MR would need multiple Map+Reduces
    with expensive store to GFS, then re-read, between each

what about distinct()? groupByKey()? join()? reduceByKey()?
  these need to look at data from *all* partitions, not just one
  because all records with a given key must be considered together
  these are the paper's "wide" dependencies (as opposed to "narrow")

how are wide dependencies implemented?
  [diagram]
  a lot like Map intermediate output in MapReduce
  the driver knows where the wide dependencies are
    e.g. between the map() and the distinct() in page-rank
    upstream transformation, downstream transformation
  the data must be "shuffled" into new partitions
    e.g. bring all of a given key together
  after the upstream transformation:
    split output up by shuffle criterion (typically some key)
    arrange into buckets in memory, one per downstream partition
  before the downstream transformation:
    (wait until upstream transformation completes -- driver manages this)
    each worker fetches its bucket from each upstream worker
    now the data is partitioned in a different way
  wide is expensive!
    all data is moved across the network
    it's a barrier -- all workers must wait until all are done

what if data is re-used?
  e.g. links4 in our page-rank
  by default, must be re-computed, e.g. re-read from input file
  persist() and cache() cause links to be saved in memory for re-use

re-using persisted data is another big advantage over MapReduce

Spark can optimized based on its view of the whole lineage graph
  stream records, one at a time, though sequence of narrow transformations
    increases locality, good for CPU data caches
    avoids having to store entire partition of records in memory
  notice when shuffles aren't needed b/c inputs already partitioned in the same way
    e.g. links4.join(ranks)

what about fault tolerance?
  what if one machine crashes?
  its memory and computation state are lost
  driver re-runs transformations on crashed machine's partitions on other machines
    usually each machine is responsible for many partitions
    so load can be spread
    thus re-computation is pretty fast
  for narrow dependencies, only lost partitions have to be re-executed

what about failures when there are wide dependencies?
  re-computing one failed partition requires information from *all* partitions
  so *all* partitions may need to re-execute from the start!
    even though they didn't fail
  Spark supports checkpoints to HDFS (like GFS) to cope with this
    driver only has to recompute along lineage from latest checkpoint
  for page-rank, perhaps checkpoint ranks every 10th iteration

limitations?
  geared up for batch processing of bulk data
  all records treated the same way
  transformations are "functional" -- turn input into output
    no notion of modifying data in place

summary
  Spark improves expressivity and performance vs MapReduce
  giving the framework a view of the complete dataflow is helpful
    performance optimizations
    failure recovery
  what were the keys to performance?
    leave data in memory between transformations, vs write to GFS then read
    re-use of data in memory (e.g. links in page-rank)
  Spark very successful, widely used