6.824 2018 Lecture 12: Spark Case Study

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing
Zaharia, Chowdhury, Das, Dave, Ma, McCauley, Franklin, Shenker, Stoica
NSDI 2012

Today: more distributed computations
- Case study: Spark
- Why are we reading Spark?
  - Widely-used for datacenter computations
  - popular open-source project, hot startup (Databricks)
  - Support iterative applications better than MapReduce
  - Interesting fault tolerance story
  - ACM doctoral thesis award

MapReduce make life of programmers easy
- It handles:
  - Communication between nodes
  - Distribute code
  - Schedule work
  - Handle failures
- But restricted programming model
  - Some apps don't fit well with MapReduce

Many algorithms are iterative
- Page-rank is the classic example
  - compute a rank for each document, based on how many docs point to it
  - the rank is used to determine the place of the doc in the search results
- on each iteration, each document:
  - sends a contribution of r/n to its neighbors
    - where r is its rank and n is its number of neighbors.
  - update rank to `alpha/N + (1 - alpha)*Sum(c_i)`,
    - where the sum is over the contributions it received
    - N is the total number of documents.
- big computation:
  - runs over all web pages in the world
  - even with many machines it takes long time

MapReduce and iterative algorithms
- MapReduce would be good at one iteration of the algorithms
  - Each map does part of the documents
  - Reduce for update the rank of a particular doc
- But what to do for the next iteration?
  - Write results to storage
  - Start a new MapReduce job for the next iteration
  - Expensive
  - But fault tolerant

Challenges
- Better programming model for iterative computations
- Good fault tolerance story

One solution: use DSM
- Good for iterative programming
  - ranks can be in shared memory
  - workers can update and read
- Bad for fault tolerance
  - typical plan: checkpoint state of memory
    - make a checkpoint every hour of memory
  - expensive in two ways:
    - write shared memory to storage during computation
    - redo all work since last checkpoint after failure
- Spark more MapReduce flavor
  - Restricted programming model, but more powerful than MapReduce
  - Good fault tolerance plan

Better solution: keep data in memory
- Pregel, Dryad, Spark, etc.
- In Spark
  - Data is stored in data sets (RDDs)
  - "Persist" the RDD in memory
  - Next iteration can refer to the RDD

Other opportunities
- Interactive data exploration
  - Run queries over the persisted RDDs

```
  Like to have something SQL-like
    A join operator over RDDs

Core idea in Spark: RDDs
  RDDs are immutable --- you cannot update them
  RDDs support transformations and actions
  Transformations:  compute a new RDD from existing RDDs
    map, reduceByKey, filter, join, ..
    transformations are lazy: don't compute result immediately
    just a description of the computation
  Actions: for when results are needed
    counts result, collect results, get a specific value

Example use:
  lines = spark.textFile("hdfs://...")
  errors = lines.filter(_.startsWith("ERROR"))    // lazy!
  errors.persist()    // no work yet
  errors.count()      // an action that computes a result
  // now errors is materialized in memory
  // partitioned across many nodes
  // Spark, will try to keep in RAM (will spill to disk when RAM is full)

Reuse of an RDD
  errors.filter(_.contains("MySQL")).count()
  // this will be fast because reuses results computed by previous fragment
  // Spark will schedule jobs across machines that hold partition of errors

Another reuse of RDD
  errors.filter(_.contains("HDFS")).map(_.split('\t')(3)).collect()

RDD lineage
  Spark creates a lineage graph on an action
  Graphs describe the computation using transformations
    lines -> filter w ERROR -> errors -> filter w. HDFS -> map -> timed fields
  Spark uses the lineage to schedule job
    Transformation on the same partition form a stage
      Joins, for example, are a stage boundary
      Need to reshuffle data
    A job runs a single stage
      pipeline transformation within a stage
    Schedule job where the RDD partition is

Lineage and fault tolerance
  Great opportunity for *efficient* fault tolerance
    Let's say one machine fails
    Want to recompute *only* its state
    The lineage tells us what to recompute
      Follow the lineage to identify all partitions needed
      Recompute them
  For last example, identify partitions of lines missing
    Trace back from child to parents in lineage
    All dependencies are "narrow"
      each partition is dependent on one parent partition
    Need to read the missing partition of lines
      recompute the transformations

RDD implementation
  list of partitions
  list of (parent RDD, wide/narrow dependency)
    narrow: depends on one parent partition  (e.g., map)
    wide: depends on several parent partitions (e.g., join)
  function to compute (e.g., map, join)
  partitioning scheme (e.g., for file by block)
  computation placement hint

Each transformation takes (one or more) RDDs, and outputs the transformed RDD.

Q: Why does an RDD carry metadata on its partitioning?  A: so transformations
  that depend on multiple RDDs know whether they need to shuffle data (wide
  dependency) or not (narrow). Allows users control over locality and reduces
  shuffles.

Q: Why the distinction between narrow and wide dependencies?  A: In case of
  failure.  Narrow dependency only depends on a few partitions that need to be
  recomputed.  Wide dependency might require an entire RDD

Example: PageRank (from paper):
```

```
  // Load graph as an RDD of (URL, outlinks) pairs
  val links = spark.textFile(...).map(...).persist() // (URL, outlinks)
  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)
  }
```

Lineage for PageRank
  See figure 3
  Each iteration creates two new RDDs:
    ranks0, ranks1, etc.
    contribs0, contribs1, etc.
  Long lineage graph!
    Risky for fault tolerance.
    One node fails, much recomputation
  Solution: user can replicate RDD
    Programmer pass "reliable" flag to persist()
     e.g., call ranks.persist(RELIABLE) every N iterations
    Replicates RDD in memory
    With REPLICATE flag, will write to stable storage (HDFS)
  Impact on performance
   if user frequently perist w/REPLICATE, fast recovery, but slower execution
   if infrequently, fast execution but slow recovery

Q: Is persist a transformation or an action? A: neither. It doesn't create a new RDD, and doesn't cause materialization. It's an instruction to the scheduler.

Q: By calling persist without flags, is it guaranteed that in case of fault that RDD wouldn't have to be recomputed? A: No. There is no replication, so a node holding a partition could fail. Replication (either in RAM or in stable storage) is necessary

Currently only manual checkpointing via calls to persist. Q: Why implement checkpointing? (it's expensive) A: Long lineage could cause large recovery time. Or when there are wide dependencies a single failure might require many partition re-computations.

Q: Can Spark handle network partitions? A: Nodes that cannot communicate with scheduler will appear dead. The part of the network that can be reached from scheduler can continue computation, as long as it has enough data to start the lineage from (if all replicas of a required partition cannot be reached, cluster cannot make progress)

What happens when there isn't enough memory?
  LRU (Least Recently Used) on partitions
     first on non-persisted
     then persisted (but they will be available on disk. makes sure user cannot overbook RAM)
  User can have control on order of eviction via "persistence priority"
  No reason not to discard non-persisted partitions (if they've already been used)

Performance
  Degrades to "almost" MapReduce behavior
  In figure 7, logistic regression on 100 Hadoop nodes takes 76-80 seconds
  In figure 12, logistic regression on 25 Spark nodes (with no partitions allowed in memory)
    takes 68.8
  Difference could be because MapReduce uses replicated storage after reduce, but Spark by default only spills to local disk
    no network latency and I/O load on replicas.
  no architectural reason why MR would be slower than Spark for non-iterative work
    or for iterative work that needs to go to disk
  no architectural reason why Spark would ever be slower than MR

Discussion
  Spark targets batch, iterative applications
  Spark can express other models
    MapReduce, Pregel
  Cannot incorporate new data as it comes in
    But see Streaming Spark
  Spark not good for building key/value store

Like MapReduce, and others
RDDs are immutable

References

http://spark.apache.org/
http://www.cs.princeton.edu/~chazelle/courses/BIB/pagerank.htm