```
6.824 2014 Lecture 4: 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

Had TreadMarks since 1996, and Distributed Shared Memory is a very general abstraction. Why
use MapReduce? Or why even use TreadMarks?
Say looking through a log, why not implement it using the regular abstractions (sockets,
files etc?)
  Saves a lot of work:
    communication between nodes
    distribute code
    schedule work
    handle failures


The MapReduce paper had a lot of impact on big data analytics: simple and powerful.
But bit too rigid. Other systems proposed fixes:

Dryad (Microsoft 2007): any directed acyclic graph, edges are communication channels, can be
through disk or via TCP.
  + can implement multiple iterations
  + can pipeline through RAM, don't have to go to disk
  - very low level:
      doesn't deal with partitioning of data, want 100,000 mappers? add 100,000 nodes
      what happens if you run out of RAM? (brief mention of "downgrading" a TCP channel to a
disk file)
  - doesn't checkpoint/replicate, in the middle of the run (so failures can be expensive)

* Pig latin (Yahoo 2008): programming language that compiles to MapReduce. Adds "Database
style" operators, mainly Join
Join: dataset 1 (k1,v1), dataset 2 (k1, v2). ==> (k1, v1, v2), takes cartesian product (all
tuples of combinations of v1, v2 with same k1)
Example: dataset 1: all clicks on products on website, dataset 2: demographics (age of
users), want average age of customer per product.
  + allows multiple iterations
  + can express more
  - still has rigidness from MR (writes to disk after map, to replicated storage after
reduce, RAM)


Spark

A framework for large scale distributed computation.
  An expressive programming model (can express iteration and joins)
  Gives user control over trade off between fault tolerance with performance
     if user frequently perist w/REPLICATE, fast recovery, but slower execution
     if infrequently, fast execution but slow recovery

Relatively recent release, but used by (partial list) IBM, Groupon, Yahoo, Baidu..
Can get substantial performance gains when dataset (or a major part of it) can fit in memory,
so anticipated to get more traction.
MapReduce is simple

Abstraction of Resilient Distributed Datasets: an RDD is a collection of partitions of
records.
Two operations on RDDs:
  Transformations: compute a new RDD from existing RDDs (flatMap, reduceByKey)
    this just specifies a plan. runtime is lazy - doesn't have to materialize (compute), so
```

```
it doesn't
  Actions: where some effect is requested: result to be stored, get specific value, etc.
    causes RDDs to materialize.

Logistic regression (from paper):
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
}

* w is sent with the closure to the nodes
* materializes a new RDD in every loop iteration


PageRank (from paper):
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)
}
```

What is an RDD (table 3, S4)
  list of partitions
  list of (parent RDD, wide/narrow dependency)
  function to compute
  partitioning scheme
  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

Handling faults.
When Spark computes, by default it only generates one copy of the result, doesn't replicate.
Without replication, no matter if it's put in RAM or disk, if node fails, on permanent failure, data is gone.
When some partition is lost and needs to be recomputed, the scheduler needs to find a way to recompute it. (a fault can be detected by using a heartbeat)
  will need to compute all partitions it depends on, until a partition in RAM/disk, or in replicated storage.

  if wide dependency, will need all partitions of that dependency to recompute, if narrow just one that RDD

So two mechanisms enable recovery from faults: lineage, and policy of what partitions to persist (either to one node or replicated)
We talked about lineage before (Transformations)

The user can call persist on an RDD.
  With RELIABLE flag, will keep multiple copies (in RAM if possible, disk if RAM is full)
  With REPLICATE flag, will write to stable storage (HDFS)
  Without flags, will try to keep in RAM (will spill to disk when RAM is full)

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.

Checkpointing is like buying insurance: pay writing to stable storage so can recover faster in case of fault.
Depends on frequency of failure and on cost of slower recovery
An automatic checkpointing will take these into account, together with size of data (how much time it takes to write), and computation time.

So can handle a node failure by recomputing lineage up to partitions that can be read from RAM/Disk/replicated storage.
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)

Shouldn't throw away partitions in RAM that are required but hadn't been used.

degrades to "almost" MapReduce behavior
In figure 7, k-means on 100 Hadoop nodes takes 76-80 seconds
In figure 12, k-means on 25 Spark nodes (with no partitions allowed in memory) takes 68.8
Difference could be because MapReduce would use replicated storage after reduce, but Spark by default would only spill to local disk, no network latency and I/O load on replicas.
no architectural reason why Spark would be slower than MR

Spark assumes it runs on an isolated memory space (multiple schedulers don't share the memory pool well).

Can be solved using a "unified memory manager"
Note that when there is reuse of RDDs between jobs, they need to run on the same scheduler to benefit anyway.

(from [P09])
Why not just use parallel databases? Commercially available: "Teradata, Aster Data, Netezza, DATAl-
legro (and therefore soon Microsoft SQL Server via Project Madi-
son), Dataupia, Vertica, ParAccel, Neoview, Greenplum, DB2 (via
the Database Partitioning Feature), and Oracle (via Exadata)"

At the time, Parallel DBMS were
 * Some are expensive and Hard to set up right
 * SQL declarative (vs. procedural)
 * Required schema, indices etc (an advantages in some cases)
 * "Not made here"

Picollo [P10] uses snapshots of a distributed key-value store to handle fault tolerance.
- Computation is comprised of control functions and kernel functions.
- Control functions are responsible for setting up tables (also locality), launching kernels, synchronization (barriers that wait for all kernels to complete), and starting checkpoints
- Kernels use the key value store. There is a function to merge conflicting writes.
- Checkpoints using Chandy-Lamport
* all data has to fit in RAM
* to recover, all nodes need to revert (expensive)
* no way to mitigate stragglers, cannot just re-run a kernel without reverting to a snapshot

[P09] "A Comparison of Approaches to Large-Scale Data Analysis", Pavlo et al. SIGMOD'09
[P10] Piccolo: Building Fast, Distributed Programs with Partitioned Tables, Power and Li, OSDI'10