Why MapReduce?

CMPT 732, Fall 2022

Why did MapReduce get created the way it was?

Why is the Hadoop cluster infrastructure (YARN, HDFS) structured the way it is?

This is a good time for a little context...

MapReduce History

The original publication: MapReduce: Simplified Data Processing on Large Clusters, 2004.

Highlights:

- "Many real world tasks are expressible in this model, as shown in the paper."
- "Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning..., scheduling..., failures..., communication."

- "Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computation involved applying a *map* operation... and then applying a *reduce* operation..."
- "map: (k1,v1) → list(k2,v2)
  reduce: (k2,list(v2)) → list(v2)"
- "A cluster consists of hundreds or thousands of machines, and therefore machine failures are common."

The Hadoop implementation of MapReduce (that we used) was inspired by this paper.

... and it seems to be a fairly faithful implementation.

MapReduce lets do computation in parallel in a *very* scalable way. Compared to other options at the time, it was very easy and flexible.

But now it seems very restrictive.

How would you implement an iterative algorithm (gradient descent, logistic regression, PageRank, etc) with MapReduce?

Probably something like:

```
bool converged = False;
int iter = 0;
while ( !converged && iter < iter_limit ) {
  Job job = Job.getInstance(conf, "gradient descent");
  ⋮
  TextInputFormat.addInputPath(job, new Path('temp' + (iter-1)));
  TextOutputFormat.setOutputPath(job, new Path('temp' + iter));
  job.waitForCompletion(true);
  converged = …;
  iter++;
}
```

Fault Tolerance

Computers fail. In a cluster, *something* failing starts to be more and more likely...

If each machine fails with probability 0.1% (in one day/week/month/year), then...

Machines in Cluster	Failure Probability
1	0.1%
10	1.0%
100	9.5%
1000	63.2%
10,000	99.996%
n	$1-(1-0.001)^n$

Lesson: In a cluster, failure is a real possibilty we have to deal with.

What happens if a cluster node disappears?

HDFS handles faults by keeping multiple copies of each block. Number of copies is controled by the `dfs.replication` configuration: default 3.

This also makes it easier to schedule jobs local to data: there are more options.

In Hadoop 3, HDFS fault tolerance can also be done with erasure coding.

This decreases the storage overhead of fault tolerance, but requires reading several nodes to get the data. Better for infrequently-accessed data.

HDFS Erasure Coding

If one of the nodes holding a particular block fails, HDFS can declare the node missing and restore the desired replication.

If *all of them* disappear, that *block* will be inaccessible, but other filles (available on ≥1 node) will keep working.

For YARN, what happens if some work is happening and the node dies? The task tight have done nothing. It might have written incomplete output (to a file/database/wherever).

Short answer: it will do its best to recover.

Remember that our jobs are broken up into smaller pieces: map or reduce tasks, Spark tasks on one partition.

If the task writing **part-00005** fails, then it should be safe to delete it and start that task on another node (because creating it is *idempotent*).

But what if our output is a database or something else? What if getting the prerequisites of **part-00005** is a huge amount of work?

These depend on the details: you should occasionally ask yourself "what if part of this fails?"

In Spark, you can cache to disk, or checkpoint contents of a DataFrame (or RDD) so that it can be read and computation resumed if a task fails.

```
from pyspark.storagelevel import StorageLevel
step1 = horrible_long_computation()
step1 = step1.persist(MEMORY_AND_DISK)
step2 = step1.…
```

```
sc.setCheckpointDir('./spark-checkpoints')
step1 = horrible_long_computation()
step1 = step1.cache().checkpoint()
step2 = step1.…
```

The difference: checkpointing completely disconnects the execution plan from the ancestors; caching/persisting doesn't. Checkpointing can also help break up a big execution plan from a big calculation.

For a DataFrame, `.cache()` is a shorthand for `.persist(MEMORY_AND_DISK)`. For an RDD, it is shorthand for `.persist(MEMORY_ONLY)`.

Where Your Data Might Be

We have seen several places our data might be stored (permanently or temporarily) while we're working on it.

There are massive speed differences between them: having some (approximate) idea of the tradeoffs is important.

Tech	Size	Latency (cycles)	Throughput (B/s)
Registers	few kB	1	
L1 cache	<100 kB	4 *	250 G–1 T **
RAM	10s GB	50–100 **	20 G–50 G *
SSD	100s GB	10^6 *	500M (per disk) *
Spinning HD	TB	10^7 *	100 M (per disk) *
Network	PB?	10^7 + disk	120 M (shared?) *

(Good consumer-grade hardware, ~2018. All values approximate.)

MapReduce's habit of putting map output onto disk seems like an obvious bottleneck.

Keeping data in memory should be much faster (if it fits). Spark gives you that option. You can use it to make your logic run faster, but you may either run out of memory and fail, or spill data to disk anyway.

