- Assignment 1: Due Sep 28 at 11am
- Assignment 2: ML will be released Sep 28th evening

- REMINDER: Submit your discussions
  - Within 24 hrs after end of class (3pm next day)
  - Each student needs to submit

- Course project details: Next week
MOTIVATION: PROGRAMMABILITY

Most real applications require multiple MR steps
  – Google indexing pipeline: 21 steps
  – Analytics queries (e.g. sessions, top K): 2-5 steps
  – Iterative algorithms (e.g. PageRank): 10’s of steps

Multi-step jobs create spaghetti code
  – 21 MR steps → 21 mapper and reducer classes
MOTIVATION: PERFORMANCE

MR only provides one pass of computation
  – Must write out data to file system in-between

Expensive for apps that need to *reuse* data
  – Multi-step algorithms (e.g. PageRank)
  – Interactive data mining
#include "mapreduce/mapreduce.h"

// User's map function
class SplitWords: public Mapper {
public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while (i < n && isspace(text[i])) i++;
      // Find word end
      int start = i;
      while (i < n && !isspace(text[i])) i++;
      if (start < i)
        Emit(text.substr(start, i-start), "1");
    }
  }
};

REGISTER_MAPPER(SplitWords);

// User's reduce function
class Sum: public Reducer {
public:
  virtual void Reduce(ReduceInput* input) {
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }
    // Emit sum for input->key() Emit(IntToString(value));
  }
};

REGISTER_REDUCER(Sum);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);
  // Specify the output files
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Sum");
  // Do partial sums within map
  out->set_combiner_class("Sum");
  // Tuning parameters
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);
  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();
  return 0;
}
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)

counts.save("out.txt")
APACHE SPARK

Programmability: clean, functional API
- Parallel transformations on collections
- 5-10x less code than MR
- Available in Scala, Java, Python and R

Performance
- In-memory computing primitives
- Optimization across operators
Resilient distributed datasets (RDDs)
  – Immutable, partitioned collections of objects
  – May be cached in memory for fast reuse

Operations on RDDs
  – *Transformations* (build RDDs)
  – *Actions* (compute results)

Restricted shared variables
  – Broadcast, accumulators


```scala
Example: Log Mining

Find error messages present in log files interactively
(Example: HTTP server logs)

``` lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('t')(2))
messages.cache()
messages.filter(_.contains("foo")).count
EXAMPLE: LOG MINING

Find error messages present in log files interactively
(Example: HTTP server logs)

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split(\'\t\')(2))
messages.cache()
messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
...
```

**Result:** search 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)
Fault Recovery

messages = textFile(...).filter(_.startsWith("ERROR"))
  .map(_.split('t')(2))
## OTHER RDD OPERATIONS

### Transformations
(define a new RDD)

- map
- filter
- sample
- `groupByKey`
- `reduceByKey`
- cogroup

### Actions
(output a result)

- collect
- `reduce`
- take
- fold

- flatMap
- union
- join
- cross
- mapValues
- ...

- count
- `saveAsTextFile`
- `saveAsHadoopFile`
- ...

---
DEPENDENCIES

Narrow Dependencies:
- map, filter
- union

Wide Dependencies:
- groupByKey
- join with inputs not co-partitioned
- join with inputs co-partitioned
Captures RDD dependency graph

Pipelines functions into “stages”

Stage boundaries are shuffle operations
Cache-aware for data reuse, locality

Partitioning-aware to avoid shuffles
SUMMARY

Spark: Generalize MR programming model

Support in-memory computations with RDDs

Job Scheduler: Pipelining, locality-aware
DISCUSSION

https://forms.gle/zBs67MZqFBFSgVxt9
for (i <- 1 to numIters) {
    val modelBC = sc.broadcast(model)
    val grad = data.mapPartitions(iter => gradient(iter, modelBC.value))
    val aggGrad = grad.reduce(case(x, y) => add(x, y))
    model = computeUpdate(aggGrad, model)
}

Binary Reduction Tree

var g = grad
var numPartitions = g.partitions.size
while (numPartitions > 1) {
    numPartitions = numPartitions / 2
    val part = new HashPartitioner(numPartitions)
    g = g.mapPartitionsWithIndex { case (partId, itr) =>
        Iterator.single(partId / 2, itr.next)
    }.reduceByKey(part, reduceFunc).values
}
When would reduction trees be better than using `reduce` in Spark?
When would reduction trees not be a good idea?
NEXT STEPS

Next week: Resource Management
- Mesos
- DRF

Assignment 1 is due soon!