What is wrong with MapReduce?

- Certain applications require extensive data reuse between computations
  - Iterative machine learning
  - Interactive data mining
- MapReduce must write intermediate results to stable storage
  - Overhead from serializability, consistent replication, disk I/O
Ye Olde Way (Distributed Shared Memory)

- Fine grained operations on shared memory in between computations
- Serialized reads and writes
- Expensive systems for consistency and fault recovery
The New Way (Resilient Distributed Datasets)

- Coarse-grained transformations on RDDs
- Trivial consistency
- Efficient fault-tolerance
Resilient Distributed Datasets (RDDs)

- Constructed from deterministic transformations (lineage)
- User-controlled memory persistence
- Limited to coarse-grained transformations
- Immutable
Partitioning

RDD

Partition

Partition

Partition
RDD Creation

Lines = spark.textFile("...")

Errors = lines.filter(_.startsWith("ERROR"))

errors.persist()

errors.filter(_.contains("HDFS"))

.map(_.split(\t\')(3))

.collect()
RDD Recovery

(Persistent in memory)
## Other Great Features of RDDs

<table>
<thead>
<tr>
<th>Aspect</th>
<th>RDDs</th>
<th>Distrib. Shared Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
<td>Coarse- or fine-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Writes</td>
<td>Coarse-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Consistency</td>
<td>Trivial (immutable)</td>
<td>Up to app / runtime</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Fine-grained and low-overhead using lineage</td>
<td>Requires checkpoints and program rollback</td>
</tr>
<tr>
<td>Straggler mitigation</td>
<td>Possible using backup tasks</td>
<td>Difficult</td>
</tr>
<tr>
<td>Work placement</td>
<td>Automatic based on data locality</td>
<td>Up to app (runtimes aim for transparency)</td>
</tr>
<tr>
<td>Behavior if not enough RAM</td>
<td>Similar to existing data flow systems</td>
<td>Poor performance (swapping?)</td>
</tr>
</tbody>
</table>
Spark

- API that provides RDD abstraction
- User provides driver to connect workers
- Driver defines a set of RDDs
- Workers can store RDDs in local RAM
Use Case: Logistic Regression

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
}
Use Case: PageRank
# Representing RDDs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>partitions()</td>
<td>Return a list of Partition objects</td>
</tr>
<tr>
<td>preferredLocations(p)</td>
<td>List nodes where partition (p) can be accessed faster due to data locality</td>
</tr>
<tr>
<td>dependencies()</td>
<td>Return a list of dependencies</td>
</tr>
<tr>
<td>iterator(p, parentIters)</td>
<td>Compute the elements of partition (p) given iterators for its parent partitions</td>
</tr>
<tr>
<td>partitioner()</td>
<td>Return metadata specifying whether the RDD is hash/range partitioned</td>
</tr>
</tbody>
</table>
Parental Dependencies

Narrow Dependencies:
- map, filter
- union
- join with inputs co-partitioned

Wide Dependencies:
- groupByKey
- join with inputs not co-partitioned
Computing RDDs
Implementation Details
Scheduling

- Uses delay scheduling:
  - Trades fairness for data locality:
    - When it's time for a task to be scheduled, wait for a bit until nodes with the input partitions are freed up
    - Otherwise, send it to preferred locations
  - Failure Handling:
    - Uses re-execution
      - Necessary partitions are recomputed in parallel, and then computation is re-executed at another node
    - Scheduler itself cannot fail!
Eviction

- LRU policy for memory management:
  - Evicts partition from least recently accessed RDD
  - Might be better to evict the least recently used partition, regardless of RDD
Check Pointing

- Certain algorithms, for example Page Rank, have lineage graphs that grow linearly
  - This makes checkpointing an attractive option when cost of recomputing RDD > cost of replicating RDD
Evaluation

- Evaluated through benchmarks on EC2 m1.xlarge nodes, with 15GB RAM. EC2 currently offers 1,952GB with x1.32xlarge nodes.
- Would be interesting to see how improvement scales with RAM
ML Performance

- Up to 20x improvement over Hadoop due to not having to serialize writes
  - Most improvement comes from later iterations, after RDD already in memory
- Larger improvement for LR because k-means is more compute intensive
  - Based off their pseudocode, they are using batch gradient descent as their optimization algorithm
  - Would they still have large improvements if they were using stochastic gradient descent?

Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem, and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

Figure 8: Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.
Understanding Speedup

- HDFS overhead:
  - ~2 seconds
- Text->Binary overhead:
  - ~7 seconds
- Binary->Java objects:
  - ~3 seconds
- LR computation itself
  - ~3 seconds
PageRank Performance

- Ran 10 iterations of PageRank to process ~4 million articles from a 54GB wikipedia dump
- In memory storage provides 2.4x speedup
- Finer grained control over partitioning improves speedup to 7.4x
- Improvement scales almost linearly
Fault Recovery

- K-means with 10 iterations on 75 node cluster, 400 tasks working on 100GB of data (but how many data points?)
- Subsequent iterations twice as fast
- Single failure in the 6th iteration
  - reconstructing RDDs with lineage takes about 25 seconds
- Lineage graphs all<10KB in size
  - Superior to replicating the 100GB working set
Performance as a function of RAM

Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.
Other Applications

- Traffic modeling: essentially used ML algorithm that uses EM for optimization or 600,000 points. EM was implemented with two “map” and “reduceByKey” operations repeatedly
  - Linear improvement with # of machines
- Spam classification: used LR over 50GB dataset with 250,000 URLs and with $10^7$ features
  - Scaling not quite linear

![Graphs showing iteration time vs. number of machines for Traffic modeling and Spam classification.](image)
Interactive Querying

- Ran queries over 1TB of Wikipedia data, each query scanned entire data set.
- Same queries with 1TB file on disk took 170s.
  - Not sure why they make this comparison.

---

![Query response time graph](chart.png)
Discussion

• Programming interface exposed by Spark is powerful; can express the cluster programming models used by a number of separate frameworks, including:
  ○ MapReduce, DryadLINQ, SQL, Pregel, Iterative MapReduce, Batched Stream Processing

• Another bonus:
  ○ RDDs recomputability from lineage facilitates debugging
  ○ By logging lineage, can re-run tasks and reconstruct RDDs for later interactive querying
  ○ Don’t have to capture events across multiple nodes, only have to log lineage graph
Questions?