Some content adapted from Matei’s NSDI talk.
Licensed for use under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.
Some material taken/derived from MIT 6.824 by Robert Morris, Franz Kaashoek, and Nickolai Zeldovich.
Problems with Map-Reduce

• Scaled analytics to thousands of machines
• Eliminated fault-tolerance as a concern

• Not very expressive
  • Iterative algorithms
    (PageRank, Logistic Regression, Transitive Closure)
  • Interactive and ad-hoc queries
    (Interactive Log Debugging)

• Lots of specialized frameworks
  • Pregel, GraphLab, PowerGraph, DryadLINQ, HaLoop...
Sharing Data between Iterations/Ops

- Only way to share data between iterations/phases is through shared storage
- Allow operations to feed data to one another
  - Ideally, through memory instead of disk-based storage
- Need the ”chain” of operations to be exposed to make this work
- Also, does this break the MR fault-tolerance scheme?
  - Retry any Map or Reduce task since idempotent
Examples

Input → HDFS read → iter. 1 → HDFS write → iter. 2 → HDFS read → HDFS write → ...

Input → HDFS read → query 1 → result 1
Input → HDFS read → query 2 → result 2
Input → HDFS read → query 3 → result 3

Slow due to replication and disk I/O, but necessary for fault tolerance
Goal: In-memory Data Sharing

10-100× faster than network/disk, but how to get FT?
Challenges

• Want distributed memory abstraction that is both fault-tolerant and efficient

• Existing storage allow fine-grained mutation to state
  • In-memory Key-value stores
  • But, they require costly on-the-fly replication for mutations

• Insight: leverage similar coarse-grained approach that transforms whole data set per op, like MR
Resilient Distributed Datasets (RDDs)

- Restricted form of distributed shared memory
  - Immutable, partitioned collections of records
  - Can only be built through coarse-grained deterministic \textit{transformations}
  - Map, filter, join...

- Efficient fault recovery using \textit{lineage}
  - Log one operation to apply to many elements
  - Recompute lost partitions on failure
  - No cost if nothing fails
Spark Programming Interface

• Scala API, exposed within interpreter as well
• RDDs
• Transformations on RDDs (RDD → RDD)
• Actions on RDDs (RDD → output)
• Control over RDD partitioning (how items are split over nodes)
• Control over RDD persistence (in RAM, on disk, or recompute on loss)
Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(f : T ⇒ U)</code></td>
<td><code>RDD[T] ⇒ RDD[U]</code></td>
</tr>
<tr>
<td><code>filter(f : T ⇒ Bool)</code></td>
<td><code>RDD[T] ⇒ RDD[T]</code></td>
</tr>
<tr>
<td><code>flatMap(f : T ⇒ Seq[U])</code></td>
<td><code>RDD[T] ⇒ RDD[U]</code></td>
</tr>
<tr>
<td><code>sample(fraction : Float)</code></td>
<td><code>RDD[T] ⇒ RDD[T]</code>   (Deterministic sampling)</td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, Seq[V])]</code></td>
</tr>
<tr>
<td><code>reduceByKey(f : (V, V) ⇒ V)</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, V)]</code></td>
</tr>
<tr>
<td></td>
<td><code>union()</code></td>
</tr>
<tr>
<td></td>
<td><code>join()</code></td>
</tr>
<tr>
<td></td>
<td><code>cogroup()</code></td>
</tr>
<tr>
<td></td>
<td><code>crossProduct()</code></td>
</tr>
<tr>
<td><code>mapValues(f : V ⇒ W)</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, W)]</code>   (Preserves partitioning)</td>
</tr>
<tr>
<td><code>sort(c : Comparator[K])</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, V)]</code></td>
</tr>
<tr>
<td><code>partitionBy(p : Partitioner[K])</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, V)]</code></td>
</tr>
</tbody>
</table>

RDDs in terms of Scala types → Scala semantics at workers
Transformations are lazy “thunks”; cause no cluster action
### Actions

<table>
<thead>
<tr>
<th>Method</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td>RDD[T] ⇒ Long</td>
</tr>
<tr>
<td><code>collect()</code></td>
<td>RDD[T] ⇒ Seq[T]</td>
</tr>
<tr>
<td><code>reduce(f : (T,T) ⇒ T)</code></td>
<td>RDD[T] ⇒ T</td>
</tr>
<tr>
<td><code>lookup(k : K)</code></td>
<td>RDD[(K, V)] ⇒ Seq[V]</td>
</tr>
<tr>
<td><code>save(path : String)</code></td>
<td>Outputs RDD to a storage system, e.g., HDFS</td>
</tr>
</tbody>
</table>

**Consumes an RDD to produce output**

- either to storage (save) or to interpreter/Scala (count, collect, reduce)

**Causes RDD lineage chain to get executed on the cluster to produce the output**

(for any missing piece of the computation)
Interactive Debugging

```scala
lines = textFile("hdfs://foo.log")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()

errors.count()
errors.filter(_.contains("MySQL")).count()  # 32 count()
errors.filter(_.contains("HDFS"))
  .map(_.split("\t")(3))
  .collect()
```

```java
16 count()
```
**persist()?**

- Not an action and not a transformation
- A scheduler hint
- Tells which RDDs the Spark schedule should *materialize* and whether in memory or storage
- Gives the user control over reuse/recompute/recovery tradeoffs

- Q: If persist() asks for the materialization of an RDD why isn’t it an action?
Lineage Graph of RDDs

```
lines filter(_.startsWith("ERROR"))

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

HDFS errors map(_.split("\t")(3))

time fields
```
Physical Execution of Tasks over RDDs
Example: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each page’s rank to

\[ \sum_{i \in \text{neighbors}} \frac{\text{rank}_i}{|\text{neighbors}_i|} \]

```scala
// RDD of (url, (Seq[url], rank)) pairs
links = RDD[(URL, Seq[URL])]

// RDD of (url, rank) pairs
ranks = RDD[(URL, Rank)]

for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
    links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
}
```

Reduce to RDD[(URL, Rank)]

For each neighbor in links, emits (URL, RankContribution)
Join (⊗)

If partitioning doesn’t match, then need to reshuffle to match pairs.
Same problem in reduce() for Map-Reduce.
Optimizing Placement

Links (url, neighbors) → Ranks<sub>0</sub> (url, rank) → join → Contribs<sub>0</sub> → reduce → Ranks<sub>1</sub> → join → Contribs<sub>2</sub> → reduce → Ranks<sub>2</sub> → ...

links & ranks repeatedly joined
Can *co-partition* them (e.g. hash both on URL) to avoid shuffles
Can also use app knowledge, e.g., hash on DNS name

\[ \text{links} = \text{links}.\text{partitionBy}(\text{new URLPartitioner()}) \]

Q: Where might we have placed persist()?
Co-partitioning Example

<table>
<thead>
<tr>
<th>From join of “top” partitions</th>
<th>foo.com</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>foo.com</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>widget.com</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>From join of “bottom” partitions</th>
<th>foo.com</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bar.com</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>bad.com</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>foo.com</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>widget.com</td>
<td>3</td>
</tr>
</tbody>
</table>

- Can avoid shuffle on join
- But, fundamentally a shuffle on reduceByKey
- Optimization: custom partitioner on domain
PageRank Performance

Figure 10a: 30 machines on 54 GB of Wikipedia data computing PageRank
Fault Recovery

RDDs track the graph of transformations that built them (their *lineage*) to rebuild lost data.

E.g.: ```messages = textFile(...).filter(_.contains("error")).map(_.split('\t')(2))```
Fault Recovery

RDDs track the graph of transformations that built them (their *lineage*) to rebuild lost data

E.g.: messages = textFile(...).filter(_.contains("error")).map(_.split(‘\t’)(2))
Fault Recovery

100 GB K-means Fig 11, partial reconstruction in step 6 much less than cost to write back results at each step with MR
Narrow & Wide Dependencies

Narrow Dependencies:
- map, filter
- union

Wide Dependencies:
- join with inputs co-partitioned
- groupByKey
- join with inputs not co-partitioned

Wide: multiple child partitions depend on partition. Must stall for all parent data, loss of child requires whole parent RDD (not just a small # of partitions)

Narrow: can pipeline on one machine
Task Scheduler

Dryad-like DAGs

Pipelines functions within a stage

Locality & data reuse aware

Partitioning-aware to avoid shuffles

= cached data partition
RDD Implementation

- partitions(): set of partitions (ranges/hash range)
- dependencies(): set of parent RDDs
- closure for computing the transformation
- preferredLocations(p): returns a set of locations where partition p can access parent data locally
- partitioner(): metadata about RDD partitioning scheme
Volcano Model

parent.partitioner() == child.partition() and no shuffle, so narrow

Iterator accesses are nested/recursive to pipeline work and avoid materialization while avoiding communication

\[ i = \text{iterator}(p, \text{qIter}) \]

\[ \text{child.count()} \{ \text{return sum([row for row in i])} \} \]
Generality of RDDs

- RDDs can express many parallel algorithms
  - They already apply the same operation to many items

- Unifies many programming models
  - Data flow models: MapReduce, Dryad, SQL, ...
  - Specialized models for iterative apps: BSP/Pregel, iterative MapReduce (Haloop), bulk incremental, ..

- Supports new applications that these models don’t
Tradeoff Space

Granularity of Updates

Fine

Coarse

Write Throughput

Low

High

Network bandwidth

Memory bandwidth

K-V stores, databases, RAMCloud

Best for transactional workloads

HDFS

RDDs

Best for batch workloads
Programming Models as Libraries

RDDs can express many existing parallel models
  » MapReduce, DryadLINQ
  » Pregel graph processing [200 LOC]
  » Iterative MapReduce [200 LOC]
  » SQL: Hive on Spark (Shark) [in progress]

Enables apps to efficiently *intermix* these models

All are based on coarse-grained operations
Memory Reuse Impact

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.
Analysis of Speedup

Fig 9: 256 GB on a single machine
Eviction and Working Sets

Fig 12: 100 GB LR on 25 machines
Scalability

Logistic Regression

K-Means

100 GB datasets each
Evolution of Spark

• Want to write Spark programs in different languages
  • Not everyone loves Scala, or JVM-based languages
• Problem: RDD semantics are bound to JVM
• Move toward Dataframes
  • Effectively, DB relations
  • Can manipulate representation w/o ser/des
  • Can use bindings in any language
• Capturing closures in many languages? Mismatching language/dataset semantics...
Conclusions

• M-R expressivity and performance have been a central point of sadness
• Several attempts to make improvements
• Spark improves expressivity, which also improves performance since scheduler can “think” across the whole pipeline
• Still preserves a lot of M-R fault-tolerance
• Does force users to reason a bit about fault-tolerance, though through careful persist() calls
• Not dead yet: TensorFlow and many more ...