Spark: Cluster Computing with Working Sets
Why?
Mesos
Resilient Distributed Dataset
Spark & Scala
Examples
Uses
Why?

- MapReduce deficiencies:
  - Standard Dataflows are Acyclic
    - Prevents Iterative Jobs
    - Not for Applications that reuse a *working set*
    - Machine Learning, Graph Applications
  - Interactive Analytics
    - Ad hoc questions answered by basic SQL queries
    - Have to wait for reads from disk, or deserialization
  - Multiple Queries
    - Processed, but ephemeral
    - Each query is individual, even if they all rely on similar base data
a) Dryad

b) Spark

Figure 4: Data flow of a logistic regression job in Dryad vs. Spark. Solid lines show data flow within the framework. Dashed lines show reads from a distributed file system. Spark reuses in-memory data across iterations to improve efficiency.
Why? : Iterative Problems

● MapReduce
● What is meant by "reuse a working set"?
  ○ Same data is reused across iterations
  ○ like Virtual Memory
● Example Algorithms
  ○ k-means -
    ■ data points to be classified
  ○ Logistic Regression
    ■ data points to be classified
  ○ Expectation Maximization
    ■ Observed Data
  ○ Alternating Least Squares
    ■ Feature Vectors for each side
How?

- Caching
- Avoid reading from files and deserializing java objects
- Main Hadoop speedup was by caching the files in memory (an OS level thing), then by caching serialized data
  - These still both need to read data in!
- Avoid even reading the data and keep it around as standard java objects
Mesos

- Resource isolation and sharing across distributed applications
- Manages pools of compute instances
  - distribution of files, work, memory
  - network communications
- Allow heterogeneous and incompatible systems to coexist within a cluster.
- Give each job the resources it needs to ensure throughput
  - Don't starve anyone
  - But make sure to utilize all available resources
- Manages different types of systems in a cluster
  - Spark, Dryad, Hadoop, MPI
- Allow multiple datasets for multiple groups to process, all using their own data.
Figure 2: Mesos architecture diagram, showing two running frameworks (Hadoop and MPI).
Resilient Distributed Dataset

- Read-only
- Partitioned across multiple machines
- Can be rebuilt if Partition goes down
- Lazily computed
- Constructed from
  - files, from a shared file system such as HDFS
  - parallel operations
  - transform on existing RDD
  - materialization
    - cache - LRU
    - save
- Not replicated in memory (possibly in a future version)
Resilient Distributed Dataset

- Transformations
  - Produce a new RDD as a result
  - Parallel Computations
    - map
    - sample
    - join
- Actions
  - Take an RDD and produce a result
  - Example
    - collect
    - reduce
    - count
    - save
# RDD Operations

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

<table>
<thead>
<tr>
<th>Actions</th>
<th>Function</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>count()</td>
<td>RDD[T] \rightarrow Long</td>
<td></td>
</tr>
<tr>
<td>collect()</td>
<td>RDD[T] \rightarrow Seq[T]</td>
<td></td>
</tr>
<tr>
<td>reduce(f : (T, T) \rightarrow T)</td>
<td>RDD[T] \rightarrow T</td>
<td></td>
</tr>
<tr>
<td>lookup(k : K)</td>
<td>RDD[(K, V)] \rightarrow Seq[V] (On hash/range partitioned RDDs)</td>
<td></td>
</tr>
<tr>
<td>save(path : String)</td>
<td>Outputs RDD to a storage system, e.g., HDFS</td>
<td></td>
</tr>
</tbody>
</table>
RDD Resiliency

- RDD always contains enough information in lineage to compute the answer
- Update vs Checkpointing
  - Lineage carries update information
  - RDDs offer checkpointing, if lineage graphs get large
    - Done manually, hoping to implement auto
Figure 11: Iteration times during k-means computation in presence of single node failure. One machine was killed at the beginning of the 6th iteration, which resulted in partial RDD reconstruction using lineage information.
 RDD FAQs

● If we can only create one from another RDD, where do we start?
  ○ Hint: What do we already have distributed in a redundant manner?
  ○ The files!
  ○ All RDDs start as a read from a file.

● What happens if I don't have enough memory to cache?
  ○ Graceful degradation
  ○ Cache as much as possible, on next pass through data, start with cached first.
  ○ Scheduler takes care of this
Limited Cache - Graceful Degradation

Figure 12: Spark performance for logistic regression using 100 GB data on 25 machines for varying size of RDD cache.
Spark Interpreter

- Wanted a "Parallel Calculator"
- Scala's Functional properties fit well with RDDs
- Running on JVM makes it compatible with Hadoop
- Standard interactive interpreter, like Lisp, Python, Matlab
- Modified Scala interpreter, not Scala compiler
- Must ensure cloud computers have bytecode that makes sense in context.
  - Serves scala bytecode to cluster
  - Modified Code Generation
    - So that all referenced variables get sent too
- Special Shared Variables
  - Broadcast - Each node is sent the data only once
  - Accumulator - Support only add operation, driver reads
Spark Interpreter

Line 1:
var query = "hello"

Line 2:
rdd.filter(_.contains(query)).count()

closure1
  line1: 
    eval(s): { return s.contains(line1.query) }

a) Lines typed by user
b) Resulting object graph

Figure 7: Example showing how the Spark interpreter translates two lines entered by the user into Java objects.
Logistic Regression

- Logistic regression is a way of describing the relationship between one or more independent variables and a binary response variable, expressed as a probability, that has only two values, such as spam, or not spam.
- Classify points in a multidimensional feature space into one of two sets.
- Uses a logistic function, which is a common sigmoid curve.
- Maps all inputs to values between 0 and 1.
- Update through gradient descent.
- This makes it functionally equivalent to backpropagation and thus a single layer artificial neural network.
Logistic Regression Example

- Points used a broadcast variable
  
  // Real points from a text file and cache them
  val points = spark.textFile(...)
    .map(parsePoint).cache()
  
  // Initialize w to random D-dimensional vector
  var w = Vector.random(D)

  // Run multiple iterations to update w
  for(i <- 1 to ITERATIONS) {
    val grad = spark.accumulator(new Vector(D))
    for (p<- points) { // parallel foreach
      val s = (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
      grad += s * p.x
    }
    w -= grad.value
  }
Logistic Regression Example

- No mention of broadcast variables or accumulators

```scala
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.random(D) // current separating plane
for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}
println("Final separating plane: "+ w)
```
Figure 9: Length of the first and later iterations for Hadoop, HadoopBinMem, and Spark for logistic regression and k-means using 100 GB data on a 100-node cluster.
Figure 8: Running times for iterations after the first one in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB data on an increasing number of machines.
Figure 9: Hadoop and Spark logistic regression running times.
## Logistic Regression Results

<table>
<thead>
<tr>
<th></th>
<th>In-memory HDFS File</th>
<th>In-memory Local File</th>
<th>Cached RDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Input</td>
<td>15.38 (0.26)</td>
<td>13.13 (0.26)</td>
<td>2.93 (0.31)</td>
</tr>
<tr>
<td>Binary Input</td>
<td>8.38 (0.10)</td>
<td>6.86 (0.02)</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Iteration times for logistic regression using 256 MB data on a single machine for different forms of input. Numbers in parentheses represent standard deviations.
Interactive Example: Console Log Mining

```scala
lines = spark.textFile("hdfs://huegLogFile")
  // load our giant file

errors = lines.filter(_.startsWith("ERROR"))
  // find all the errors
  // NOTE: errors is not computed yet. LAZY!

errors.cache() // when materialized, cache

errors.count() // returns the # of errors
  // errors is materialized

errors.filter(_.contains("MySQL")).count()
  // MySQL errors
```
errors.filter(_.contains("HDFS"))
  .map(_split('\t')(3))
  .collect()

// Return the time fields of errors mentioning
// HDFS as an array (assuming time is field
// number 3 in a tab-separated format)
Figure 1: Lineage chain for the distributed dataset objects defined in the example in Section 4.
Interactive Example: Wikipidia

- Find total views of
  - all pages
  - pages with titles exactly matching
  - pages with titles partially matching
- Initial load into memory took 170 seconds

Figure 14: Response times for different interactive queries on increasingly larger input datasets.
Use: Mobile Millennium Project

- Traffic Reporting and Prediction
  - Data from Taxis and GPS-enabled Mobile Phones
- Uses Expectation Maximization
  - Alternates between two different map/reduce steps
- Originally Python & Hadoop with a PostgreSQL Server
- Moved to Spark, with several optimizations
  - Each bringing 2-3x speed improvement
  - End result was a better algorithm, at faster than real time processing
Figure 2: An example output of traffic estimates on the Mobile Millennium visualizer
Broadcast road network and load observations

Draw samples for each observations and weigh them by their likelihood (E step)

Collect samples by link (Shuffle step)

Compute maximum likelihood estimate of parameters based on samples (M step)

Broadcast link distribution parameters and iterate

**Figure 4:** Data flow in the importance sampling EM algorithm we employed. The algorithm iterates through the E, shuffle and M steps until it converges.
Figure 7: Running time experiments on different clusters. See section 5 for details.
Conclusion

- Caching helps, even if you can't hold every thing in memory
- RDDs are a good abstraction in the MapReduce world
Extra Slides
EC2 Information

Amazon Elastic Compute Cloud
Computing capacity in the cloud.

Extra Large Instance

15 GB memory
8 EC2 Compute Units (4 virtual cores with 2 EC2 Compute Units each)
1,690 GB instance storage
64-bit platform
I/O Performance: High
API name: m1.xlarge

High-Memory Quadruple Extra Large Instance

68.4 GB of memory
26 EC2 Compute Units (8 virtual cores with 3.25 EC2 Compute Units each)
1690 GB of instance storage
64-bit platform
I/O Performance: High
API name: m2.4xlarge
Quick Spark

- modification of scala, which is on the jvm
- interactive toplevel (from scala interpreter)
- abstraction called a Resilient Distributed Dataset (RDD)
  - read only collection, partitioned across machines that can be rebuilt if a portion is lost.
- Outperforms Hadoop in iterative machine learning applications
- built on mesos
  - provides underlying data