Spark!

Game plan for today:

What's wrong with Map-Reduce?

Spark: Basic Concepts

Some examples

Spark on our cluster

Project proposals

What's wrong with Map-Reduce?

A different question: what's *right* with Map-Reduce?

- Scales wonderfully
- Nice HDFS abstraction
- Flexible formalism (in some ways)

So, what's the problem?

What's wrong with Map-Reduce?

- *Primarily* good for batch-processing...
- ... iterative algorithms need a lot of thought:
- Clunky API
- Inefficient for iterative algorithms (lots of data schlepping)

Fundamentally, Map-Reduce is a *low-level* programming abstraction.

Spark is a higher-level API for Hadoop programming:

Rather than explicitly creating discrete M-R jobs, one codes “as normal” using familiar functional programming constructs:

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val ones = errs.map(_ => 1)
val count = ones.reduce(_+_)
```

A note on language:

Spark is written in Scala:


```scala
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello, world!")
  }
}
```

```scala
object MatchTest2 extends App {
  def matchTest(x: Any): Any = x match {
    case 1 => "one"
    case "two" => 2
    case y: Int => "scala.Int"
  }
  println(matchTest("two"))
}
```

```scala
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val ones = errs.map(_ => 1)
val count = ones.reduce(_+_)
```

1. ssh

```scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SimpleApp {

  def main(args: Array[String]) {

    val nyt = "/data/nyt/nyt_eng.tok.txt"

    val conf = new SparkConf().setAppName("Scala Test")
    val sc = new SparkContext(conf)
    val nytData = sc.textFile(nyt)

    val numAs = nytData.filter(line => line.contains("a")).count()
    val numBs = nytData.filter(line => line.contains("b")).count()

    println("Lines with a: %s, Lines with b: %s".format(numAs, numBs))


  }

}
                                                          1,25          All
```

A note on language:

Spark is written in Scala:

... but there are APIs in Python, R, etc.

```python
"""SimpleApp.py"""
from pyspark import SparkContext, SparkConf

nyt = "/data/nyt/nyt_eng.tok.txt"

conf = SparkConf().setAppName("Simple Test App")

sc = SparkContext(conf=conf)
nytData = sc.textFile(nyt).cache()

numAs = nytData.filter(lambda s: 'a' in s).count()
numBs = nytData.filter(lambda s: 'b' in s).count()

print("Lines with a: %i, lines with b: %i" % (numAs, numBs))
```

Spark is built around “RDD”s:

Resilient...

... Distributed...

... Datasets.

The key idea: an RDD *looks* like a single object...

... but is *actually* distributed across the cluster.

(Using regular HDFS-esque partitioning)

Spark is built around "RDD"s:

Other key ideas:

RDDs are *lazily* constructed...

... "know" how they were created...

... and can be cached for future use.

RDDs, "under the hood," comprise:

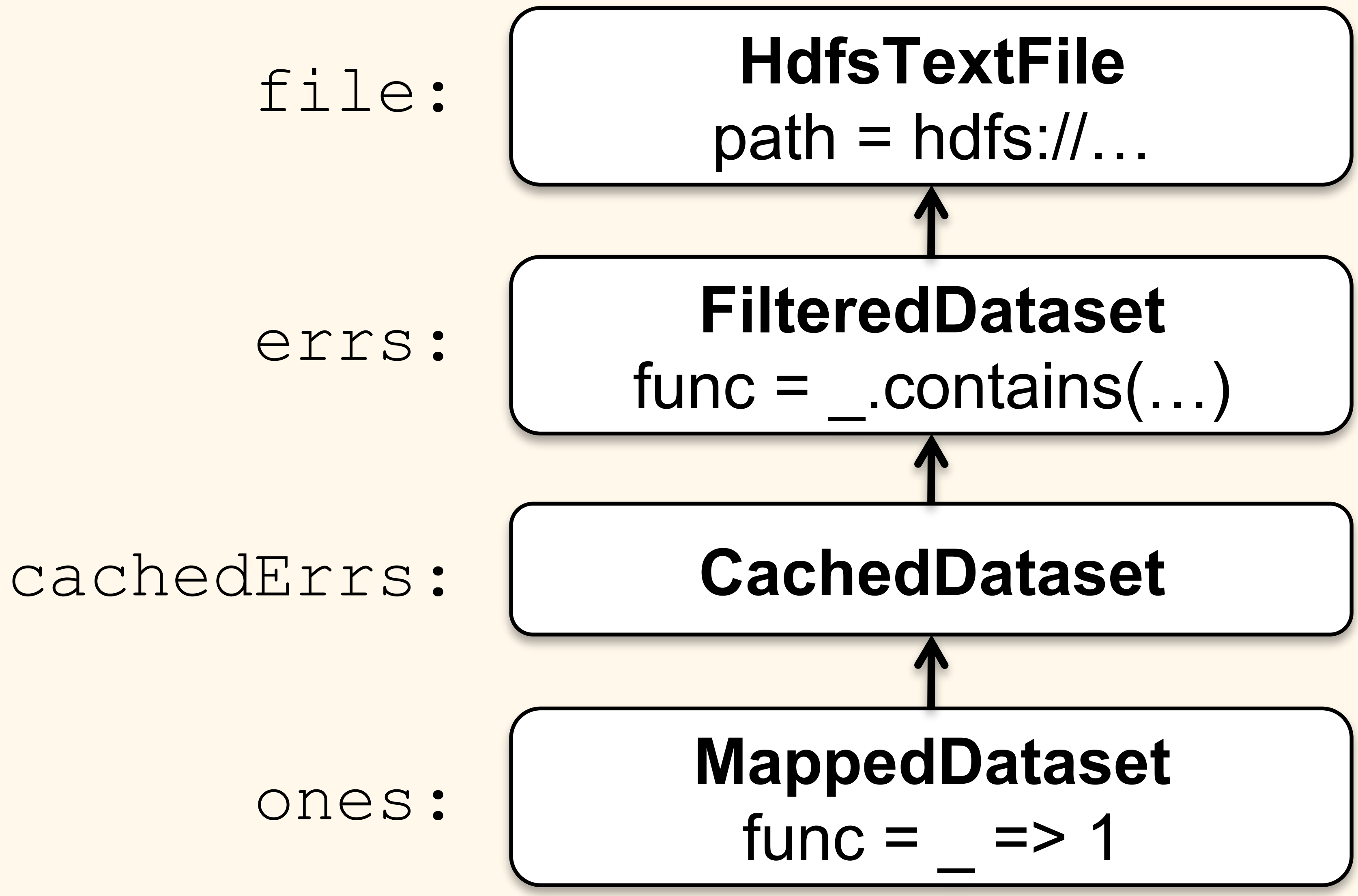
An array of partitions...

A partition-level function...

A list of parent RDDs...

A partitioner function (optional)...

A list of partition-level "preferred locations" (optional).


file:
HdfsTextFile
path = hdfs://…
errs:
FilteredDataset
func = _.contains(…)
cachedErrs:
CachedDataset
ones:
MappedDataset
func = _ => 1

Spark is built around "RDD"s:

RDDs are *immutable*, and support two kinds of operation:

transformations and *actions*

RDD *transformations*:

1. Occur lazily;

2. Produce another RDD.

Spark is built around “RDD”s:

RDDs are *immutable*, and support two kinds of operation:

transformations and *actions*

RDD *actions*:

1. Trigger computation;
2. Produce *values*.

Types of transformations:

map, flatMap, filter, join, split, sort, reduce, etc.

There are two main families of transformation:

“Narrow” transformations live within a single partition (*map, filter*, etc.)…

“Wide” transformations require data from multiple partitions, and so involve a shuffle operation (*reduceByKey, groupByKey,* etc.)

Types of actions:

collect, count, first, min, etc.

Actions result in actual computation, and are synchronous.

```
val nytData = sc.textFile(nyt_path).cache()
val nytWords = nytData.flatMap(_.split("\\s+"))
val nytLongWords = nytWords.filter(_.length > 10)
val nytWordPairs = nytLongWords.map((_,1))
val nytWordCounts = nytWordPairs.reduceByKey(_ + _)

val top10 = nytWc.takeOrdered(10)
(Ordering[Int].reverse.on(_._2))
```

```
top10: Array[(String, Int)] = Array((information,373606), (administration,315473),
(Republicans,247374), (international,213471), (International,206628), (Association,
179004), (performance,176204), (presidential,173842), (particularly,166205),
(development,155663))
```

Spark lets you program in parallel very naturally:

```
// Read 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) { // Runs in parallel
    val s = (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
    grad += s * p.x
  }
  w -= grad.value
}
```

"Spark: Cluster Computing with Working Sets", Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica. HotCloud 2010. June 2010.

Spark lets you program in parallel very naturally:

```
// Read 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) { // Runs in parallel
    val s = (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
    grad += s * p.x
  }
  w -= grad.value
}
```

"Spark: Cluster Computing with Working Sets", Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica. HotCloud 2010. June 2010.

“Local” vs. “Broadcast” variables:

```
val Rb = spark.broadcast(R)
for (i <- 1 to ITERATIONS) {
  U = spark.parallelize(0 until u)
            .map(j => updateUser(j, Rb, M))
            .collect()
  M = spark.parallelize(0 until m)
            .map(j => updateUser(j, Rb, U))
            .collect()
}
```

If we hadn't broadcast Rb, the map() calls would have shipped a “fresh” copy of it each time.

"Spark: Cluster Computing with Working Sets", Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica. HotCloud 2010. June 2010.

This, in combination with RDDs durability, makes Spark well-suited to iterative algorithms.

It's also a lot nicer to program in than vanilla Map-Reduce!

Useful Spark APIs:

MLlib:

Clustering, classification, regression, linear algebra, feature extraction, etc.

GraphX:

Graph processing (stay tuned!)

SparkSQL:

Pandas-style data frames!

Running Spark programs:

1. Compile Scala program, run with `spark-submit`
2. Write Python script, run with `spark-submit`
3. Use `spark-shell` Scala REPL
4. Use `pyspark` Python REPL

Notes on our cluster:

1. Remember to set "`--master yarn`"

   (default is local standalone mode!)

2. Remember to set "`--num-executors`"

   (default is 2!)

Demo time!

Project proposals:

Due April 18

Should include:

1. Research question
2. What data set you'll be working with
3. What tools you'll be using
4. Your evaluation plan
5. What you'll do for a pilot study