Spark

Zaharia et al., "Resilient Distributed Datasets:
A Fault-Tolerant Abstraction for In-Memory Cluster Computing", Proc. NSDI 2012

Another Abstraction: Spark

- Let’s think of just having a big block of RAM, partitioned across machines…
  - And a series of operators that can be executed in parallel across the different partitions
- That’s basically Spark's resilient distributed datasets (RDDs)
  - Spark programs are written by defining functions to be called over items within collections
  - (similar model to LINQ, FlumeJava, Apache Crunch, and several other environments)

Spark: Transformations and actions

- RDDs are read-only, partitioned collections
- Programmer starts by defining a new RDD based on data in stable storage
  - Example: `lines = spark.textFile("hdfs://foo/bar");`
- Programmer can create more RDDs by applying transformations to existing ones
  - Example: `errors = lines.filter(_.startsWith("ERROR"));`
- Only when an action is performed does Spark do actual work:
  - Example: `errors.count()`
  - Example: `errors.filter(_contains("HDFS")).map(_split("\t")(3)).collect()`

Programming Model

Resilient distributed datasets (RDDs)

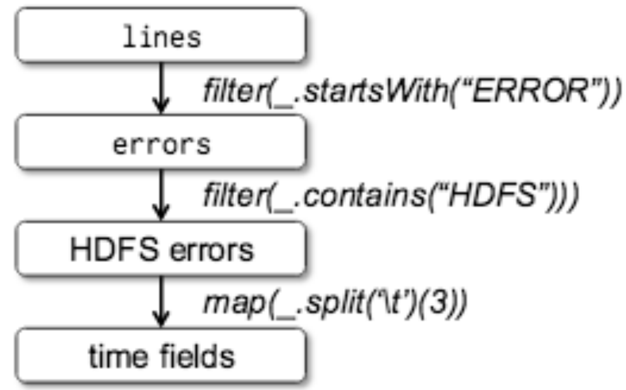
- Immutable, partitioned collections of objects
- Created through parallel *transformations* (map, filter, groupBy, join, …) on data in stable storage
- Can be *cached* for efficient reuse

Actions on RDDs

- Count, reduce, collect, save, …

Spark: Lineage

- Spark keeps track of how RDDs have been constructed
  - Result is a lineage graph
  - Vertexes represent RDDSs, edges represent transformations



- What could this be useful for?
  - Fault tolerance: When a machine fails, the corresponding piece of the RDD can be recomputed efficiently
    - How would a multi-stage MapReduce program achieve this?
  - Efficiency: Not all RDDs have to be 'materialized' (i.e., kept in RAM as a full copy)

Spark Examples

```
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)

counts.saveAsTextFile("hdfs://...")
```

```
def sample(p):
    x, y = random(), random()
        return 1 if x*x + y*y < 1 else 0

count = sc.parallelize(xrange(0,  NUM_SAMPLES)).map(sample) \
           .reduce(lambda a, b: a + b)

print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
```

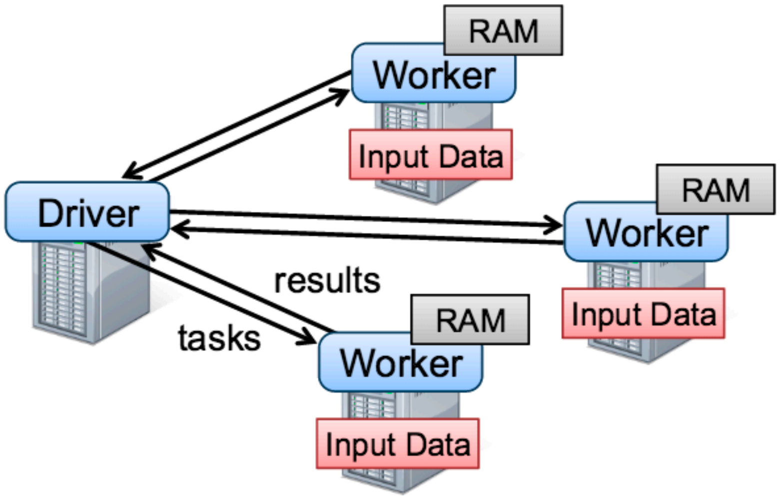
```
textFile = sc.textFile("hdfs://...")

# Creates a DataFrame having a single column named "line"
df = textFile.map(lambda r: Row(r)).toDF(["line"])
errors = df.filter(col("line").like("%ERROR%"))
# Counts all the errors
errors.count()
# Counts errors mentioning MySQL
errors.filter(col("line").like("%MySQL%")).count()
# Fetches the MySQL errors as an array of strings
errors.filter(col("line").like("%MySQL%")).collect()
```

Spark Operations

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

Spark: Implementation

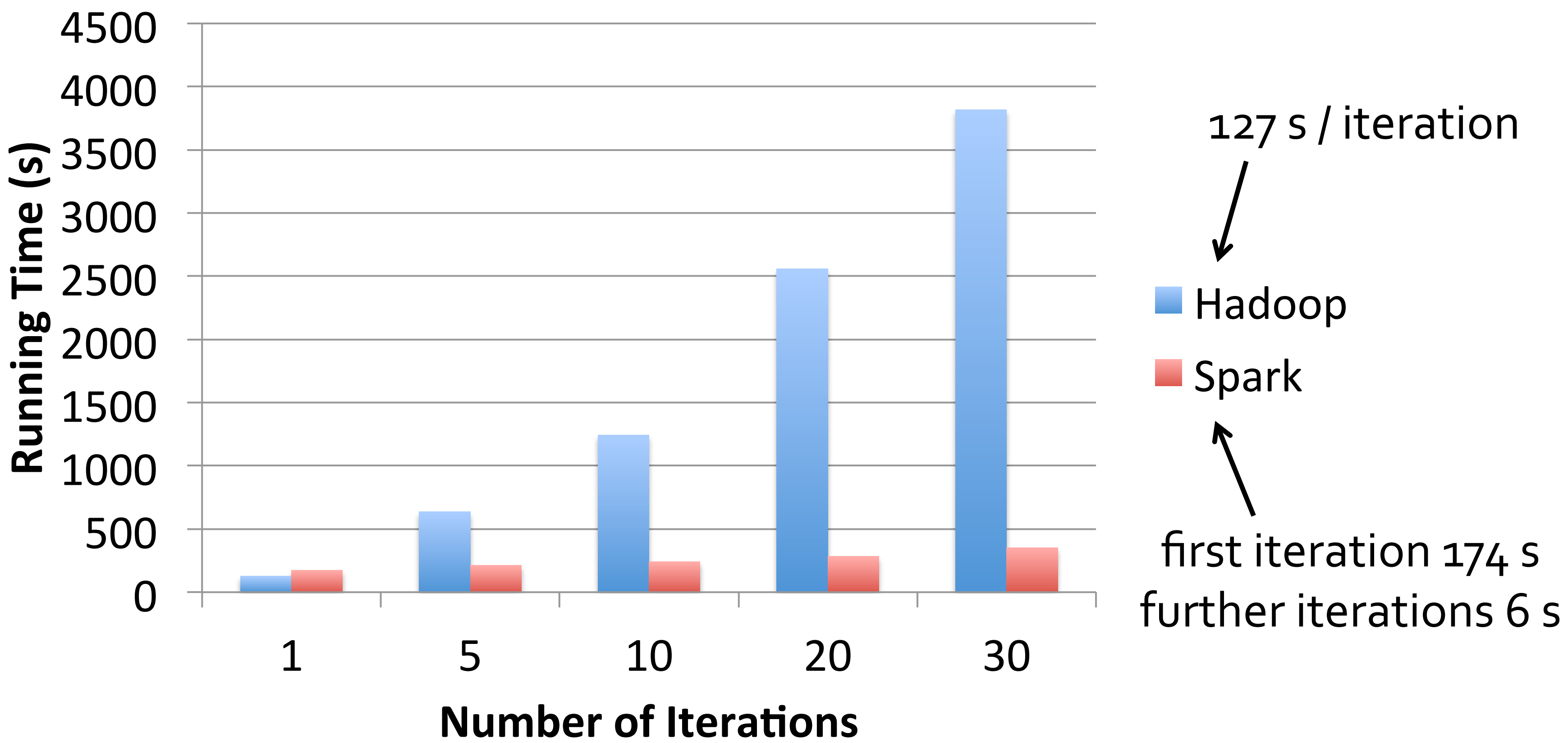


- Developer writes a driver program that connects to a cluster of workers
  - Driver defines RDDs, invokes actions, tracks lineage
  - Workers are long-lived processes that store pieces of RDDs in memory and perform computations on them
  - Many of the details will sound familiar: Scheduling, fault detection and recovery, handling stragglers, etc.

What can you do easily in Spark?

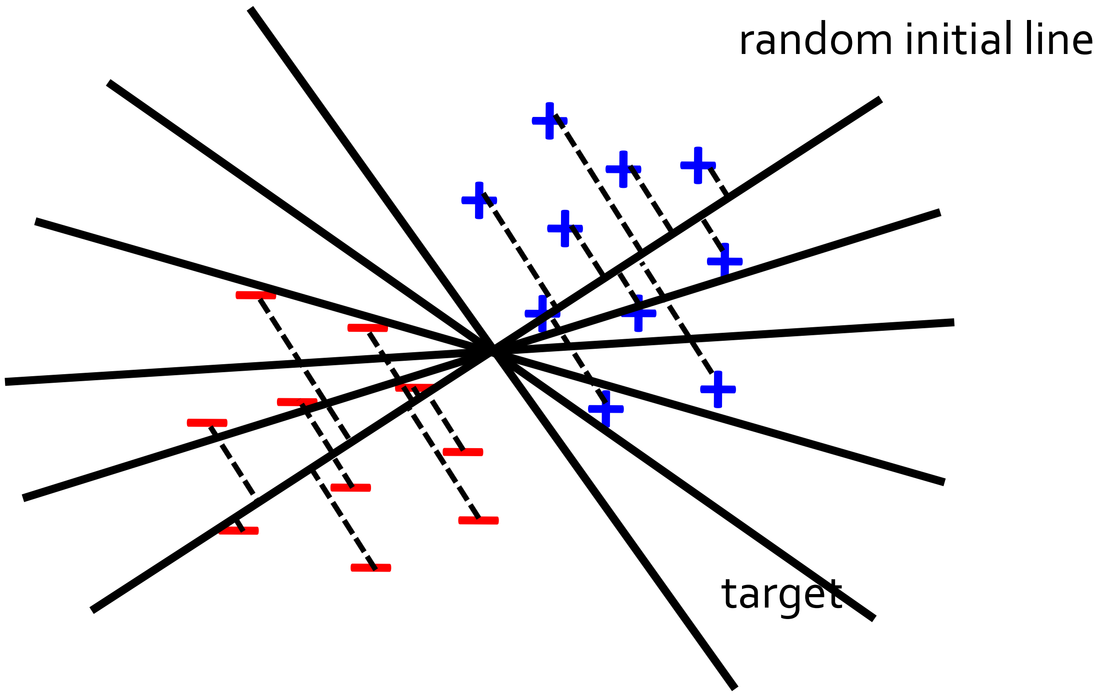
- Global aggregate computations that produce program state – compute the count() of an RDD, compute the max diff, etc.
- Loops!
- Built-in abstractions for some other common operations like joins
- See also Apache Crunch / Google FlumeJava for a very similar approach

Logistic Regression Performance Example



Example: Logistic Regression

Goal: find best line separating two sets of points



Example: Logistic Regression

```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```