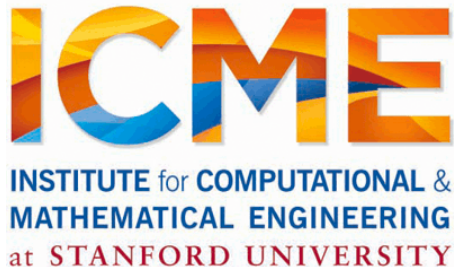


Advanced Data Science on Spark

Reza Zadeh

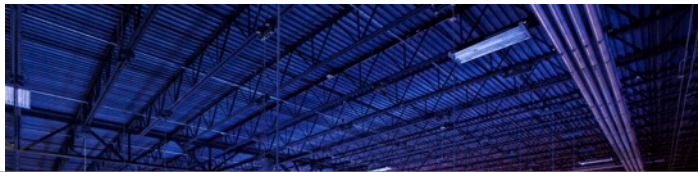


Data Science Problem

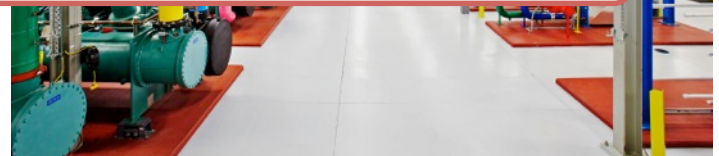
Data growing faster than processing speeds

Only solution is to parallelize on large clusters

» Wide use in both enterprises and web industry



How do we program these things?



Use a Cluster

Convex Optimization

Numerical Linear Algebra

Matrix Factorization

Large Graph analysis

Machine Learning

Streaming and online
algorithms

Following lectures on <http://stanford.edu/~rezab/dao>

Outline

Data Flow Engines and Spark

The Three Dimensions of Machine Learning

Communication Patterns

Advanced Optimization

State of Spark Ecosystem

Traditional Network Programming

Message-passing between nodes (e.g. MPI)

Very difficult to do at scale:

- » How to split problem across nodes?
 - Must consider network & data locality
- » How to deal with failures? (inevitable at scale)
- » Even worse: stragglers (node not failed, but slow)
- » Ethernet networking not fast
- » Have to write programs for each machine

Rarely used in commodity datacenters

Disk vs Memory

L1 cache reference:	0.5 ns
L2 cache reference:	7 ns
Mutex lock/unlock:	100 ns
Main memory reference:	100 ns
Disk seek:	10,000,000 ns

Network vs Local

Send 2K bytes over 1 Gbps network:	20,000 ns
Read 1 MB sequentially from memory:	250,000 ns
Round trip within same datacenter:	500,000 ns
Read 1 MB sequentially from network:	10,000,000 ns
Read 1 MB sequentially from disk:	30,000,000 ns
Send packet CA->Netherlands->CA:	150,000,000 ns

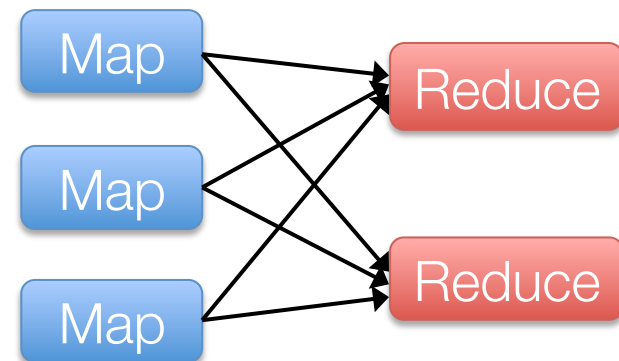
Data Flow Models

Restrict the programming interface so that the system can do more automatically

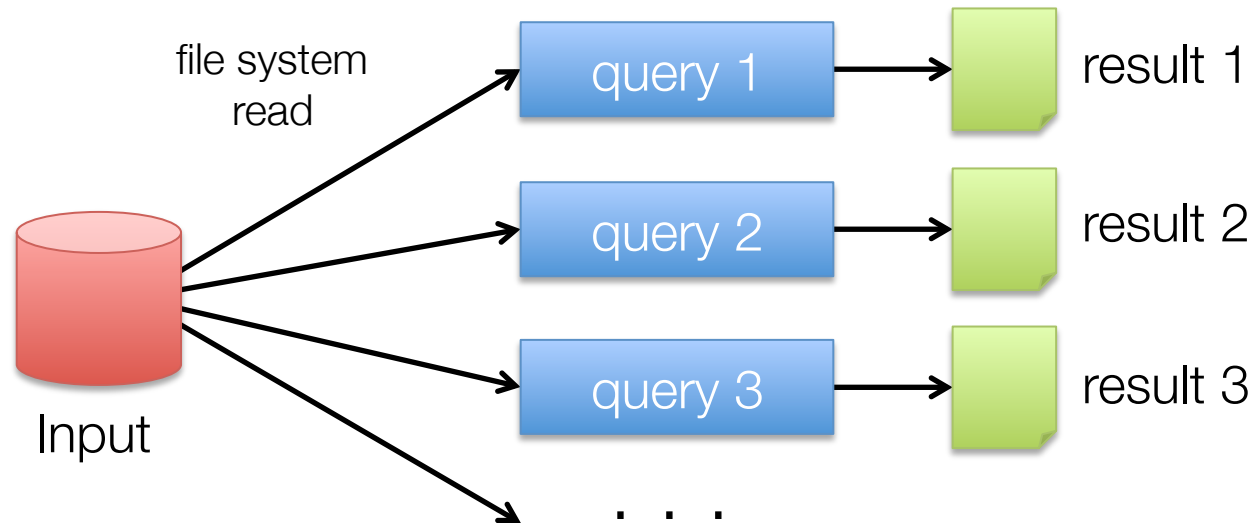
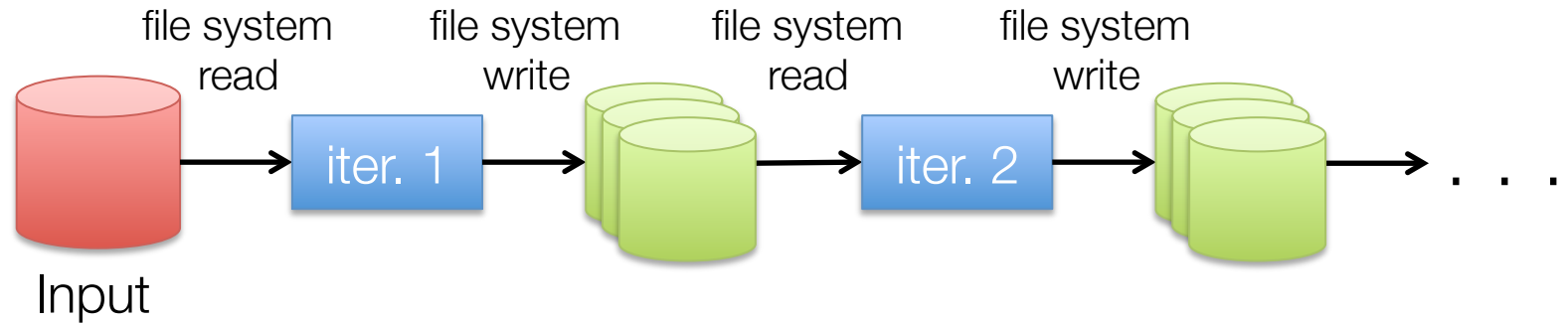
Express jobs as graphs of high-level operators

- » System picks how to split each operator into tasks and where to run each task
- » Run parts twice fault recovery

Biggest example: MapReduce



Example: Iterative Apps



Commonly spend 90% of time doing I/O

MapReduce evolved

MapReduce is great at one-pass computation, but inefficient for *multi-pass* algorithms

No efficient primitives for data sharing

- » State between steps goes to distributed file system
- » Slow due to replication & disk storage

Verdict

MapReduce algorithms research doesn't go to waste, it just gets sped up and easier to use

Still useful to study as an algorithmic framework, silly to use directly

Spark Computing Engine

Extends a programming language with a distributed collection data-structure

- » “Resilient distributed datasets” (RDD)

Open source at Apache

- » Most active community in big data, with 50+ companies contributing

Clean APIs in Java, Scala, Python

Community: SparkR, being released in 1.4!

Key Idea

Resilient Distributed Datasets (RDDs)

- » Collections of objects across a cluster with user controlled partitioning & storage (memory, disk, ...)
- » Built via parallel transformations (map, filter, ...)
- » The world only lets you make make RDDs such that they can be:

Automatically rebuilt on failure

Resilient Distributed Datasets (RDDs)

Main idea: Resilient Distributed Datasets

- » Immutable collections of objects, spread across cluster
- » Statically typed: `RDD[T]` has objects of type `T`

```
val sc = new SparkContext()  
val lines = sc.textFile("log.txt")    // RDD[String]
```

```
// Transform using standard collection operations
```

```
val errors = lines.filter(_.startsWith("ERROR"))
```

```
val messages = errors.map(_.split('\t')(2))
```

→ lazily evaluated

```
messages.saveAsTextFile("errors.txt")
```

→ kicks off a computation

MLlib: Available algorithms

classification: logistic regression, linear SVM, naïve Bayes, least squares, classification tree

regression: generalized linear models (GLMs), regression tree

collaborative filtering: alternating least squares (ALS), non-negative matrix factorization (NMF)

clustering: k-means||

decomposition: SVD, PCA

optimization: stochastic gradient descent, L-BFGS

The Three Dimensions

ML Objectives

Almost all machine learning objectives are optimized using this update

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^n g(w; x_i, y_i)$$

w is a vector of dimension d
we're trying to find the best w via optimization

Scaling

1) Data size

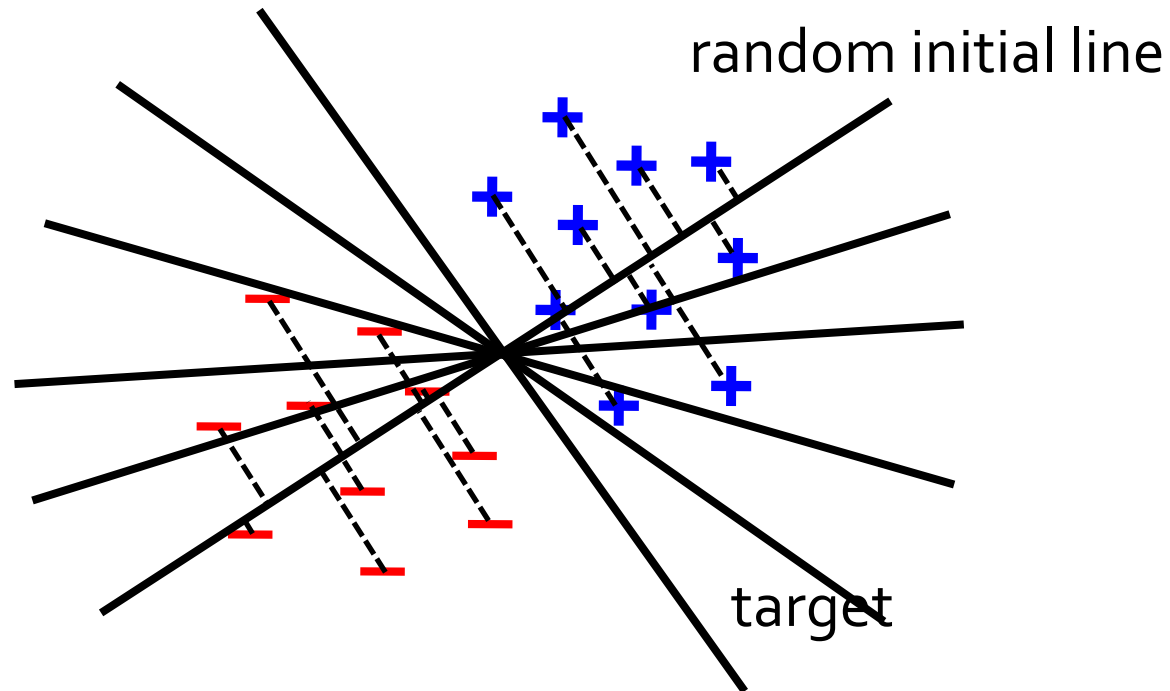
$$w \leftarrow w - \alpha \cdot \sum_{i=1}^n g(w; x_i, y_i)$$

2) Number of models

3) Model size

Logistic Regression

Goal: find best line separating two sets of points



Data Scaling

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^n g(w; x_i, y_i)$$

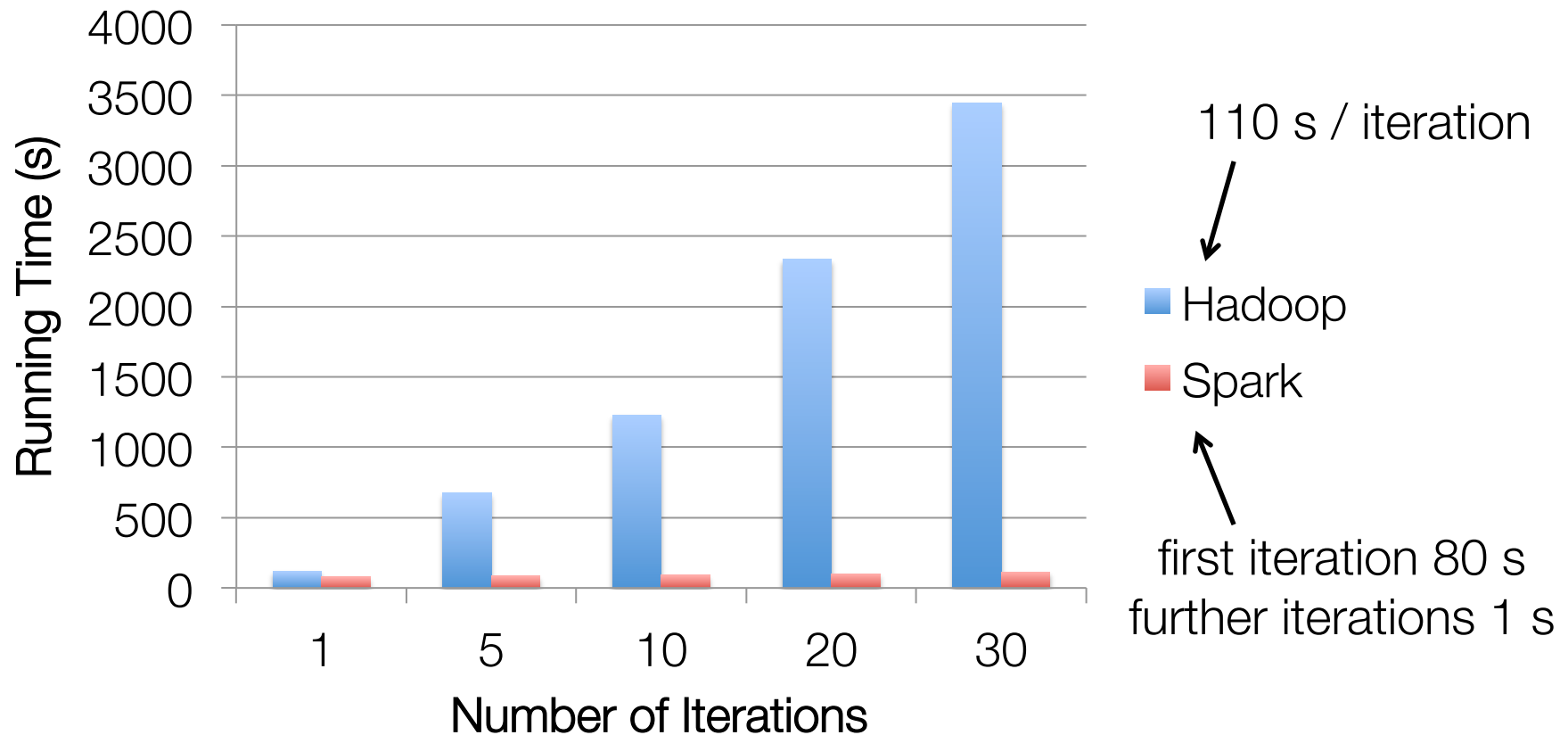
```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= alpha * gradient
}
```

Separable Updates

Can be generalized for

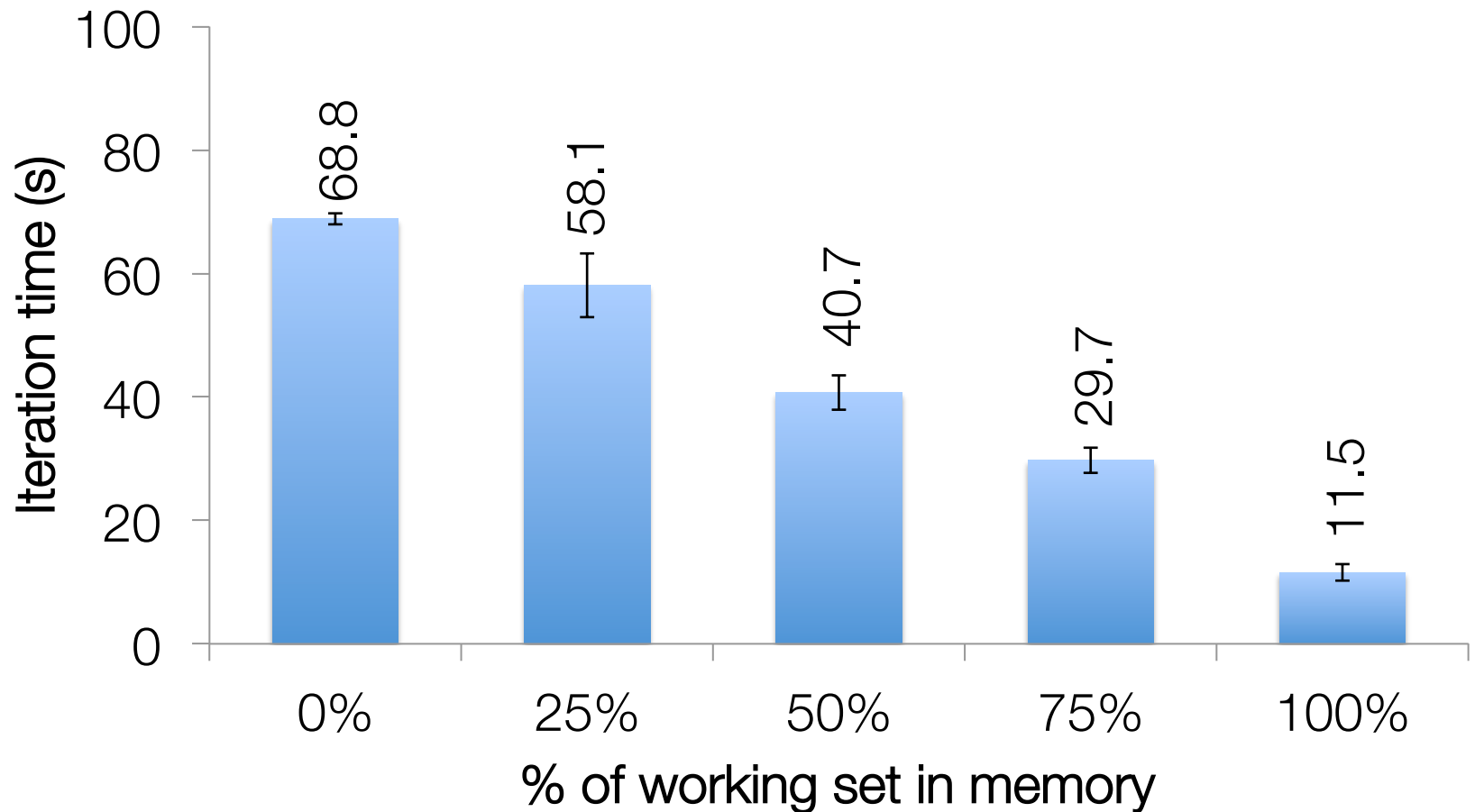
- » Unconstrained optimization
- » Smooth or non-smooth
- » LBFGS, Conjugate Gradient, Accelerated Gradient methods, ...

Logistic Regression Results



100 GB of data on 50 m1.xlarge EC2 machines

Behavior with Less RAM



Lots of little models

Is embarrassingly parallel

Most of the work should be handled by data flow paradigm

ML pipelines does this

Hyper-parameter Tuning

```
// Build a parameter grid.
val paramGrid = new ParamGridBuilder()
  .addGrid(hashingTF.numFeatures, Array(10, 20, 40))
  .addGrid(lr.regParam, Array(0.01, 0.1, 1.0))
  .build()

// Set up cross-validation.
val cv = new CrossValidator()
  .setNumFolds(3)
  .setEstimator(pipeline)
  .setEstimatorParamMaps(paramGrid)
  .setEvaluator(new BinaryClassificationEvaluator)

// Fit a model with cross-validation.
val cvModel = cv.fit(trainingDataset)
```

Model Scaling

Linear models only need to compute the dot product of each example with model

Use a BlockMatrix to store data, use joins to compute dot products

Coming in 1.5

Model Scaling

Data joined with model (weight):



Life of a Spark Program

Life of a Spark Program

- 1) Create some input RDDs from external data or parallelize a collection in your driver program.
- 2) Lazily **transform** them to define new RDDs using transformations like `filter()` or `map()`
- 3) Ask Spark to `cache()` any intermediate RDDs that will need to be reused.
- 4) Launch **actions** such as `count()` and `collect()` to kick off a parallel computation, which is then optimized and executed by Spark.

Example Transformations

<code>map()</code>	<code>intersection()</code>	<code>cartesion()</code>
<code>flatMap()</code>	<code>distinct()</code>	<code>pipe()</code>
<code>filter()</code>	<code>groupByKey()</code>	<code>coalesce()</code>
<code>mapPartitions()</code>	<code>reduceByKey()</code>	<code>repartition()</code>
<code>mapPartitionsWithIndex()</code>	<code>sortByKey()</code>	<code>partitionBy()</code>
<code>sample()</code>	<code>join()</code>	<code>...</code>
<code>union()</code>	<code>cogroup()</code>	<code>...</code>

Example Actions

`reduce()`

`collect()`

`count()`

`first()`

`take()`

`takeSample()`

`saveToCassandra()`

`takeOrdered()`

`saveAsTextFile()`

`saveAsSequenceFile()`

`saveAsObjectFile()`

`countByKey()`

`foreach()`

`...`

Communication Patterns

None:

Map, Filter (embarrassingly parallel)

All-to-one:

reduce

One-to-all:

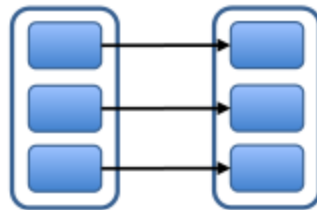
broadcast

All-to-all:

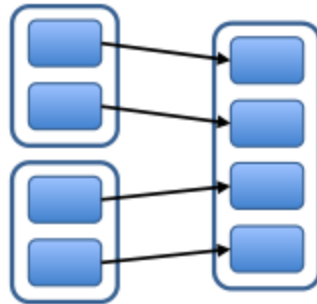
reduceByKey, groupByKey, Join

Communication Patterns

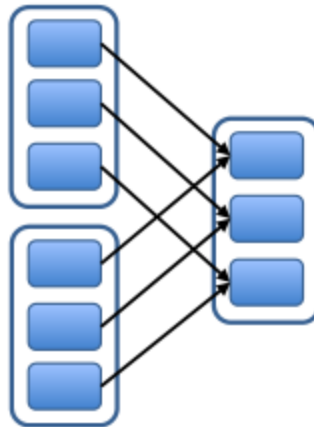
Narrow Dependencies:



map, filter



union

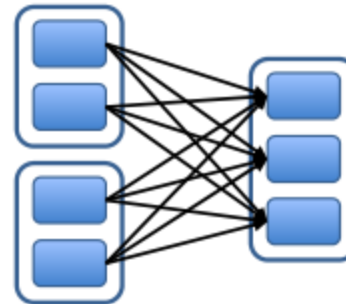


join with inputs
co-partitioned

Wide Dependencies:



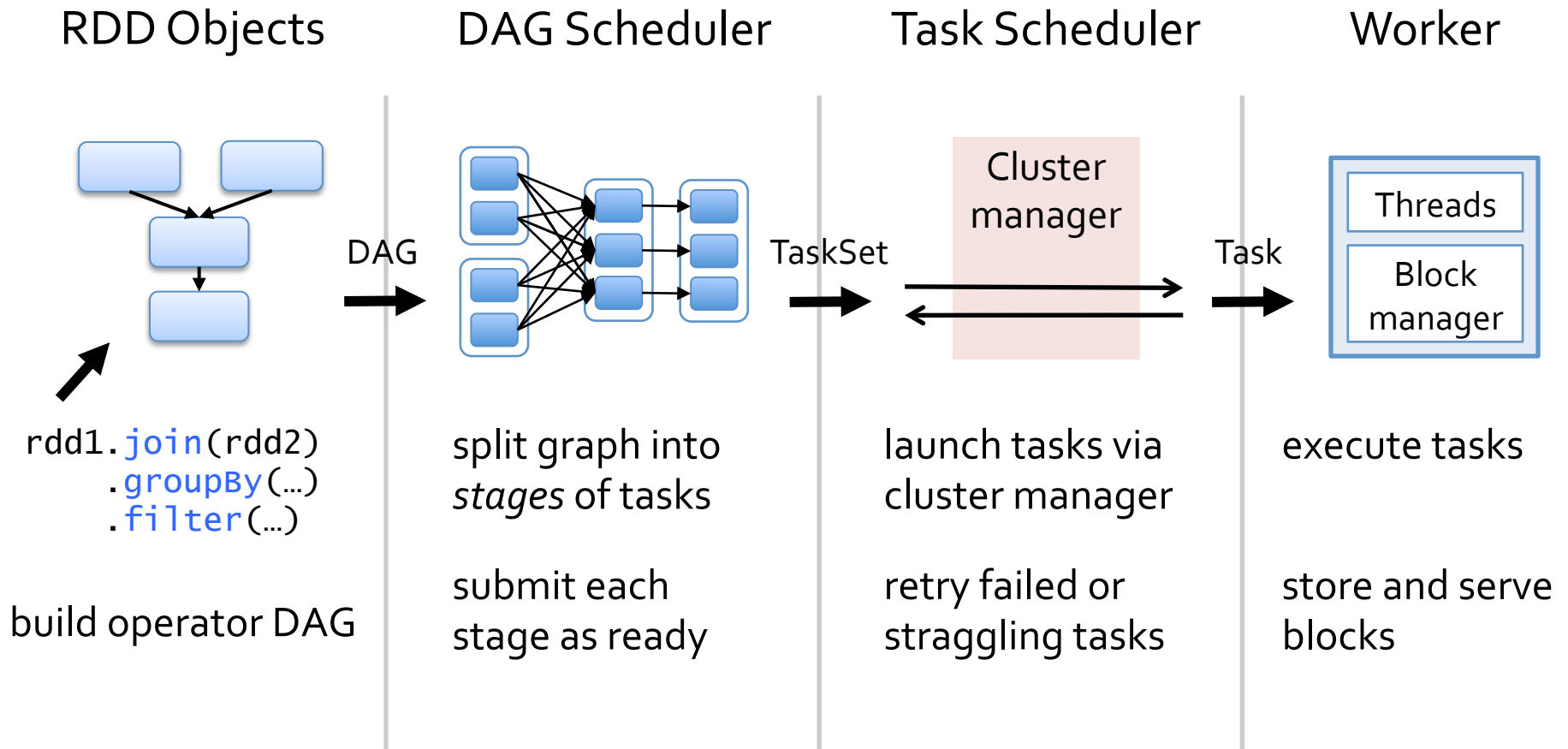
groupByKey




join with inputs not
co-partitioned

Shipping code to the cluster

RDD → Stages → Tasks

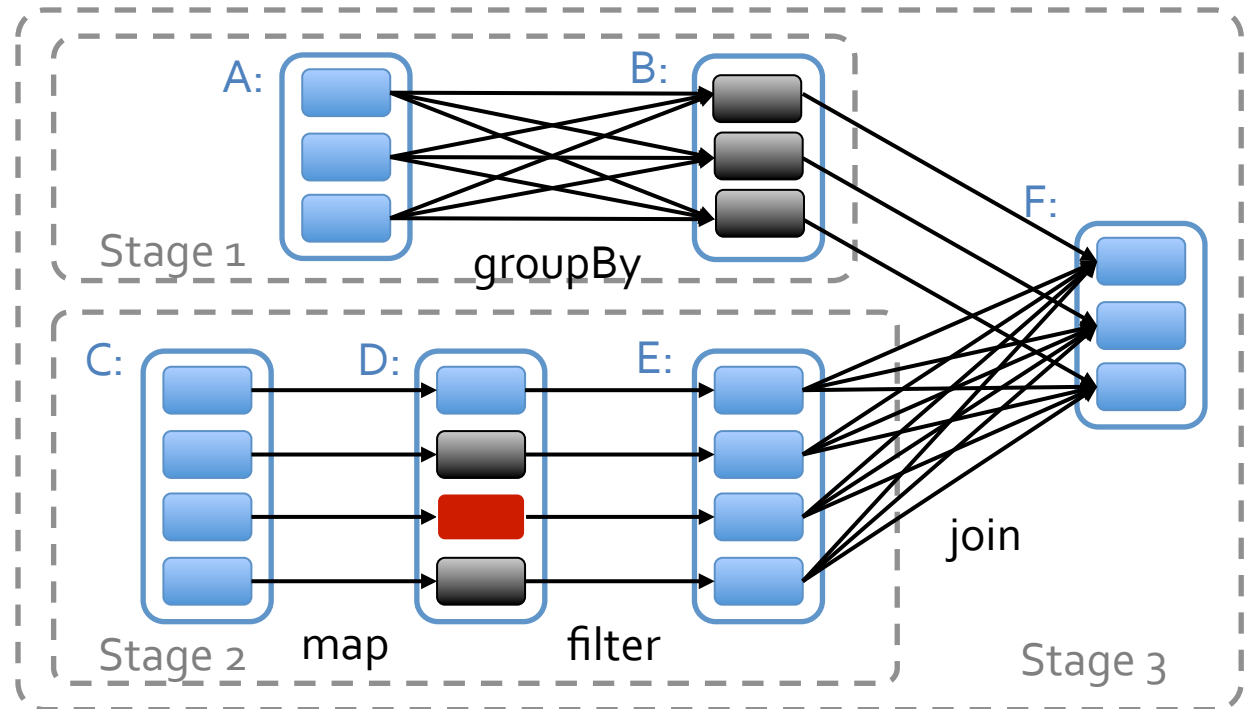


Example Stages

 = RDD

 = cached partition

 = lost partition



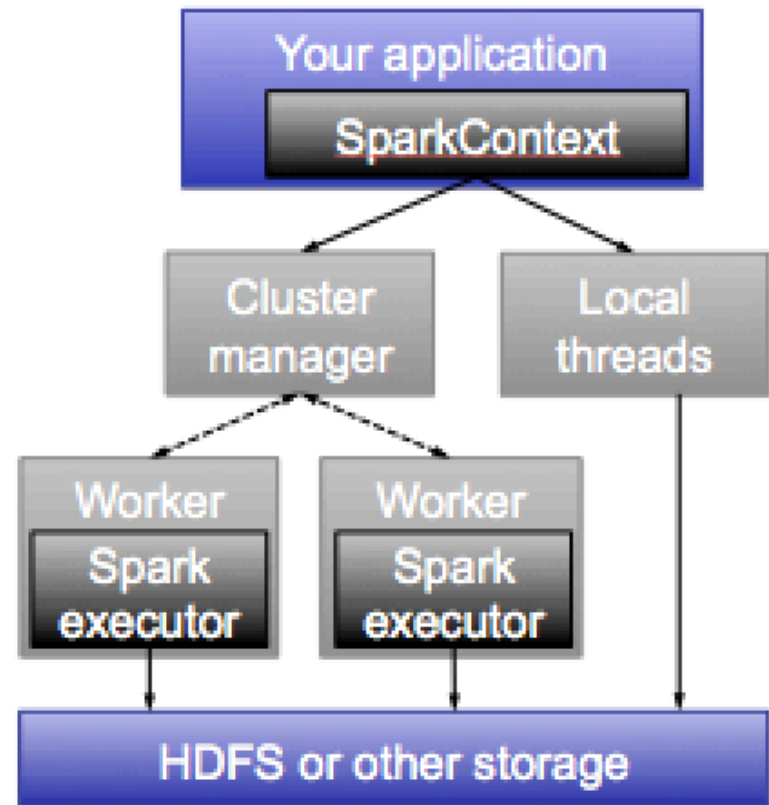
Talking to Cluster Manager

Manager can be:

YARN

Mesos

Spark Standalone

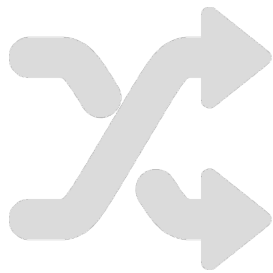


Shuffling (everyday)

How would you do a reduceByKey on a cluster?

Sort! Decades of research has given us algorithms such as TimSort

Shuffle



=

groupByKey

sortByKey

reduceByKey

Sort: use advances in sorting single-machine
memory-disk operations for all-to-all communication

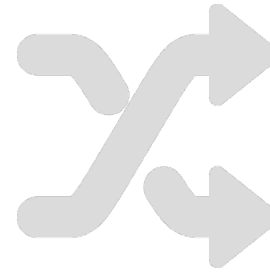
Sorting

Distribute Timsort, which is already well-adapted to respecting disk vs memory

Sample points to find good boundaries

Each machines sorts locally and builds an index

Sorting (shuffle)



	Hadoop World Record	Spark 100 TB *	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min
Sort Benchmark Daytona Rules	Yes	Yes	No
Environment	dedicated data center	EC2 (i2.8xlarge)	EC2 (i2.8xlarge)

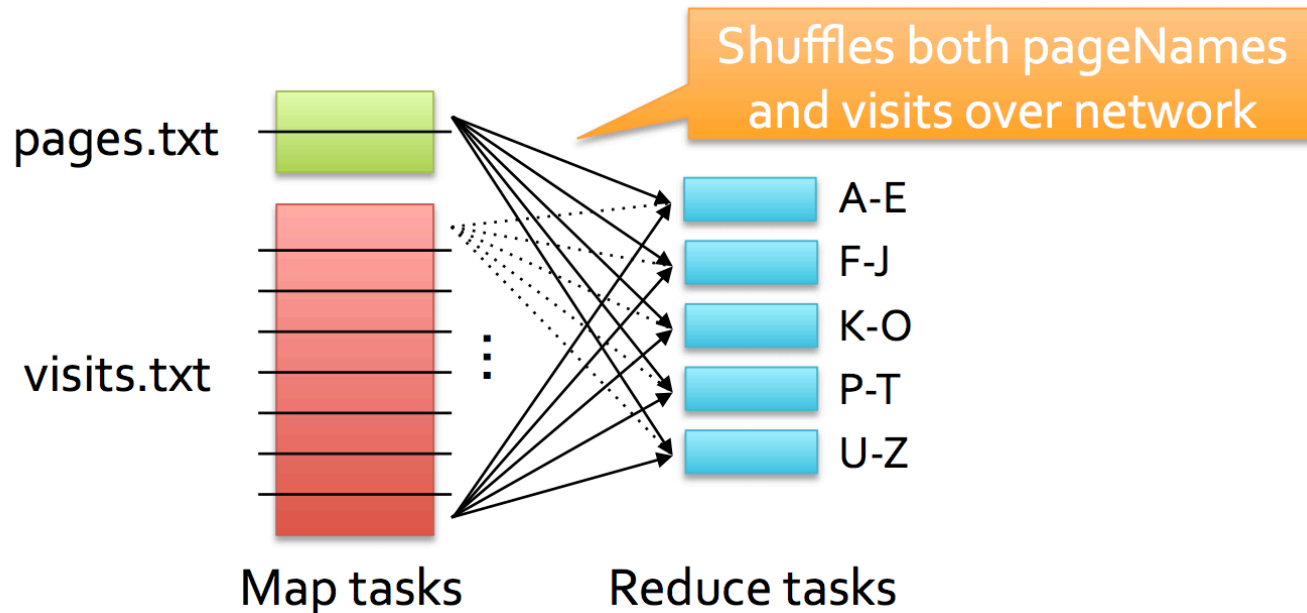
Distributed TimSort

Example Join

```
// Load RDD of (URL, name) pairs  
val pageNames = sc.textFile("pages.txt").map(...)
```

```
// Load RDD of (URL, visit) pairs  
val visits = sc.textFile("visits.txt").map(...)
```

```
val joined = visits.join(pageNames)
```



Broadcasting

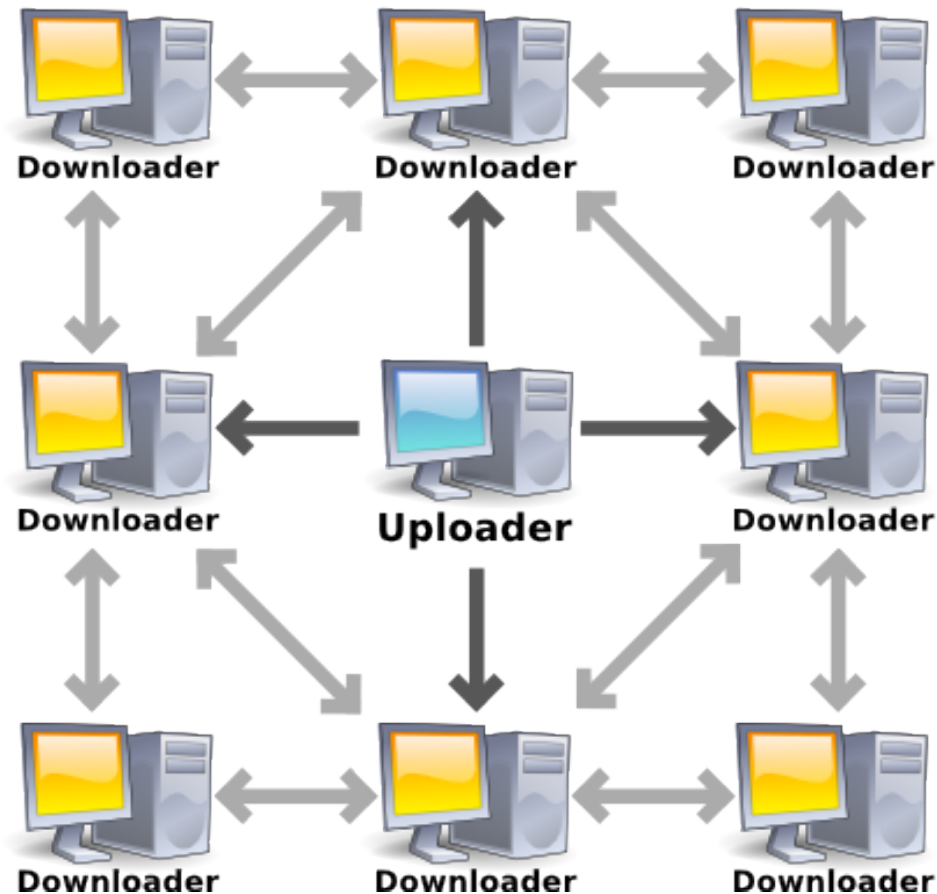
Broadcasting

Often needed to propagate current guess for optimization variables to all machines

The exact wrong way to do it is with “one machines feeds all” – use bit-torrent instead

Needs $\log(p)$ rounds of communication

Bit-torrent Broadcast



Broadcast Rules

Create with `SparkContext.broadcast(initialVal)`

Access with `.value` inside tasks (first task on each node to use it fetches the value)

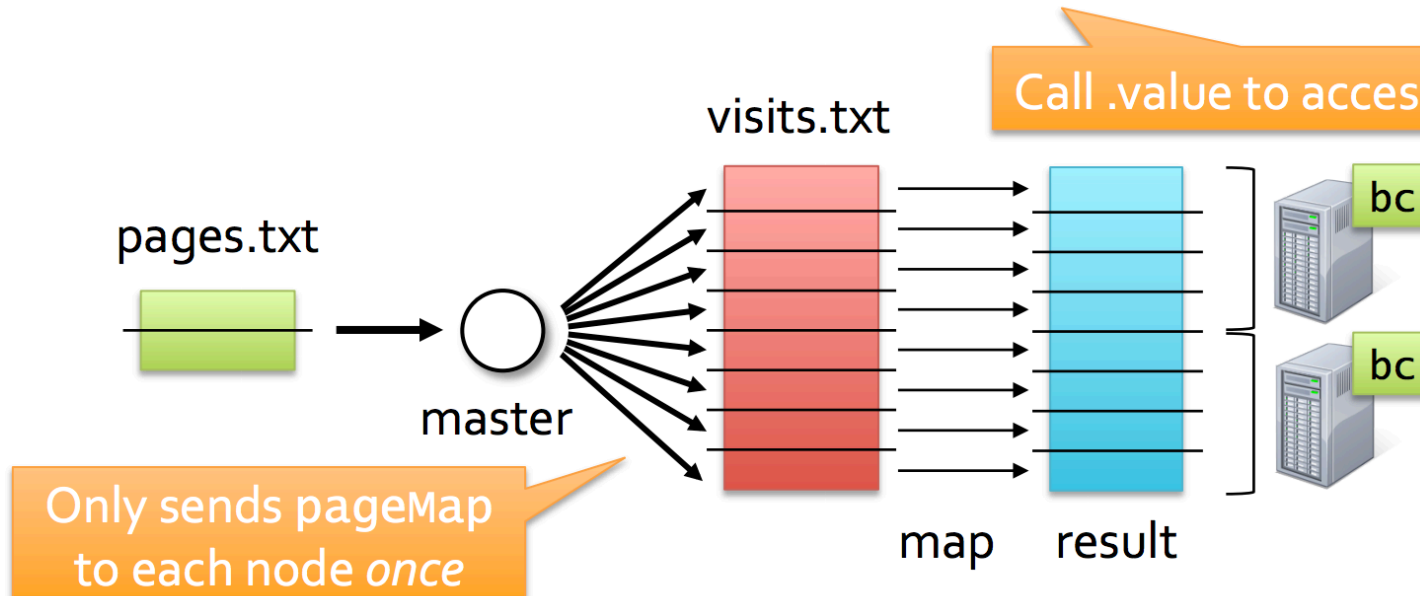
Cannot be modified after creation

Replicated Join

```
val pageNames = sc.textFile("pages.txt").map(...)
val pageMap = pageNames.collect().toMap()
val bc = sc.broadcast(pageMap)
val visits = sc.textFile("visits.txt").map(...)
val joined = visits.map(v => (v._1, (bc.value(v._1), v._2)))
```

Type is Broadcast[Map[...]]

Call .value to access value



Optimization Example: Gradient Descent

Logistic Regression

Already saw this with data scaling

Need to optimize with broadcast

Model Broadcast: LR

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^n g(w; x_i, y_i)$$

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= alpha * gradient
}
```

Model Broadcast: LR

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^n g(w; x_i, y_i)$$

Call sc.broadcast

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= alpha * gradient
}
```

Use via .value

Rebroadcast with sc.broadcast

Separable Updates

Can be generalized for

- » Unconstrained optimization
- » Smooth or non-smooth
- » LBFGS, Conjugate Gradient, Accelerated Gradient methods, ...

State of the Spark ecosystem

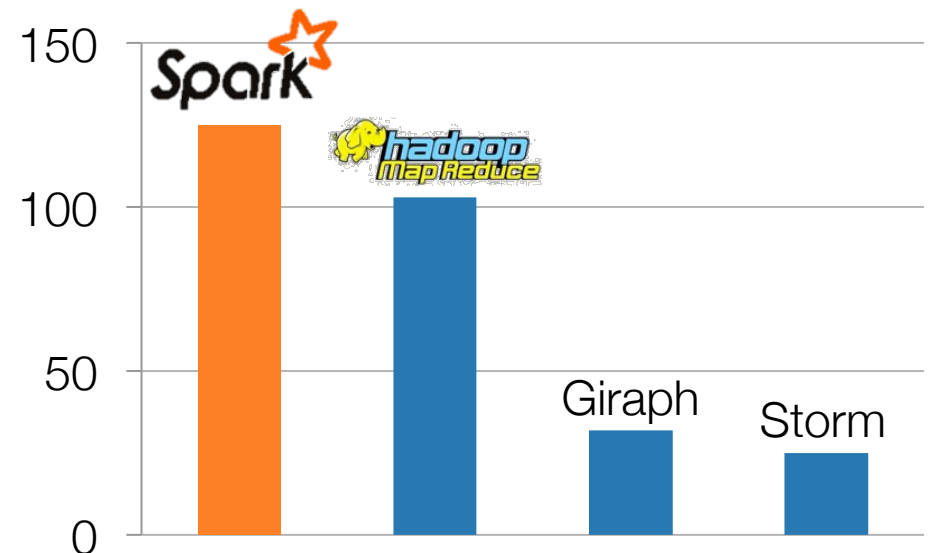
Spark Community

Most active open source community in big data

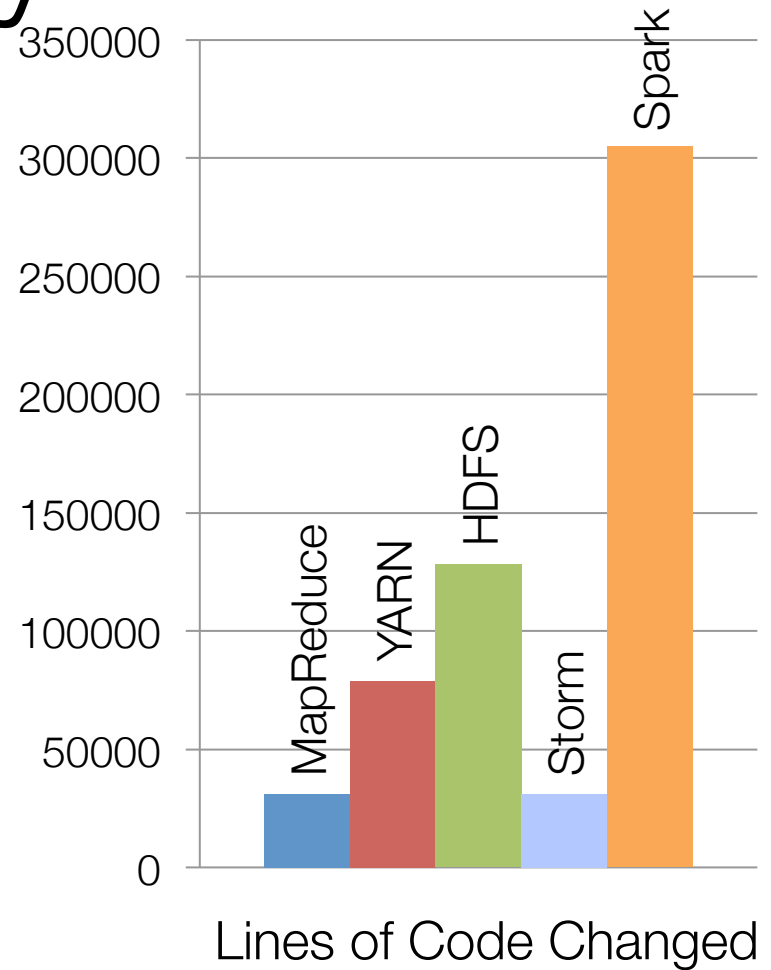
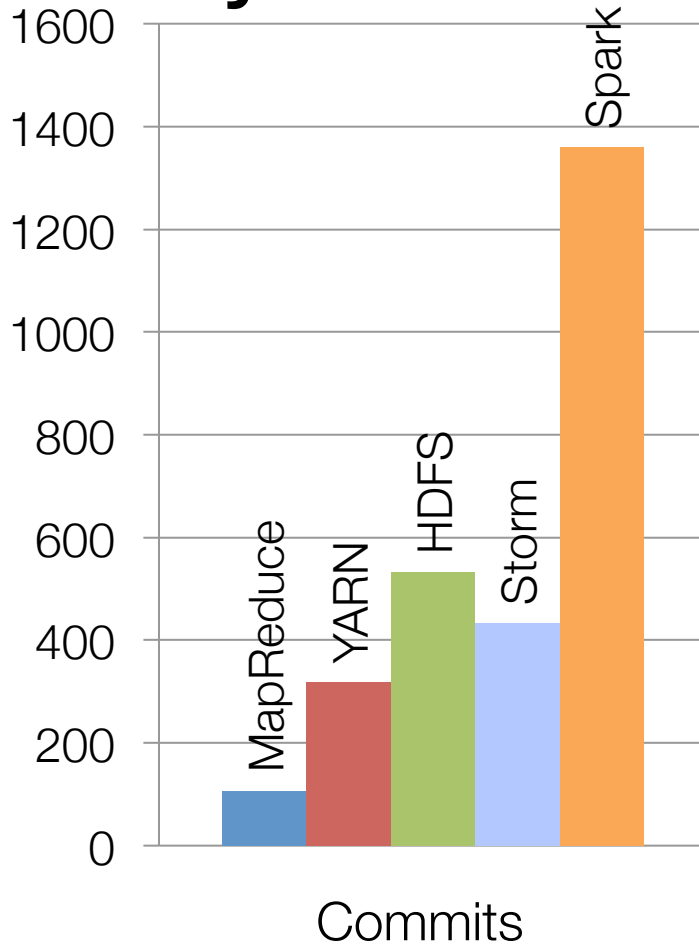
200+ developers, 50+ companies contributing



Contributors in past year



Project Activity



Activity in past 6 months

Continuing Growth



Contributors per month to Spark

Conclusions

Spark and Research

Spark has all its roots in research, so we hope to keep incorporating new ideas!

Conclusion

Data flow engines are becoming an important platform for numerical algorithms

While early models like MapReduce were inefficient, new ones like Spark close this gap

More info: spark.apache.org

