Advanced Spark Features

Matei Zaharia

UC Berkeley

www.spark-project.org

Motivation

You've now seen the core primitives of Spark: RDDs, transformations and actions

As we've built applications, we've added other primitives to improve speed & usability

» A key goal has been to keep Spark a small, extensible platform for *research*

These work seamlessly with the existing model

Spark Model

Process distributed collections with functional operators, the same way you can for local ones

```
val points: RDD[Point] = // ...
var clusterCenters = new Array[Point](k)

val closestCenter = points.map {
  p => findClosest(clusterCenters, p)
}
...
```

Spark Model

Process distributed collections with functional operators, the same way you can for local ones

```
val points: RDD[Point] = // ...
var clusterCenters = new Array[Point](k)

val closestCenter = points.map {
  p => findClosest(clusterCenters, p)
}
...
```

Two foci for extension: collection storage & layout, and interaction of functions with program

Spark Model

Process distributed collections with functional operators, the same way you can for local ones

```
val points: RDD[Point] = // ...
var clusterCenters = new Array[Point](k)

val closestCenter = points.map {
  p => findClosest(clusterCenters, p)
}
...
```

How should this be split across nodes?

Two foci for extension: collection storage & layout, and interaction of functions with program

Spark Model

Process distributed collections with functional operators, the same way you can for local ones

```
val points: RDD[Point] = // ...
var clusterCenters = new Array[Point](k)

val closestCenter = points.map {
  p => findClosest(clusterCenters, p)
}
...
```

How should this variable be shipped?

Two foci for extension: collection storage & layout, and interaction of functions with program

Outline

Broadcast variables

Accumulators

Controllable partitioning

Extending Spark

Richer shared variables

Data layout

Motivation

Normally, Spark closures, including variables they use, are sent separately with each task

In some cases, a large read-only variable needs to be shared *across* tasks, or across operations

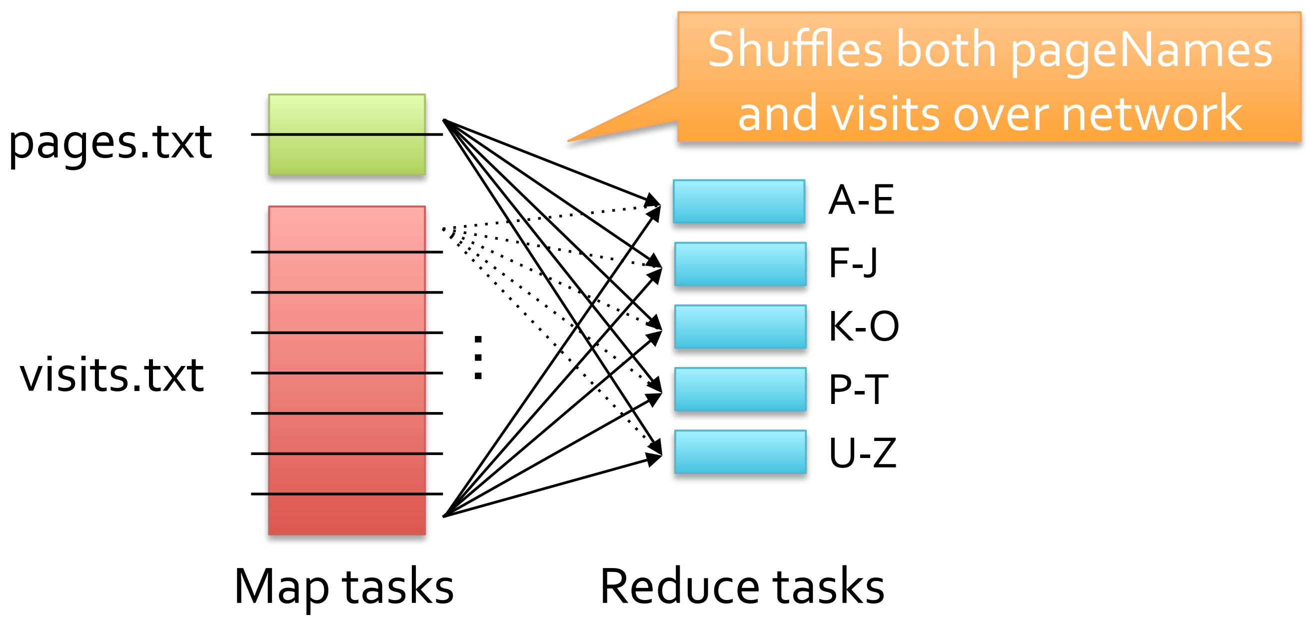
Examples: large lookup tables, “map-side join”

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)
```

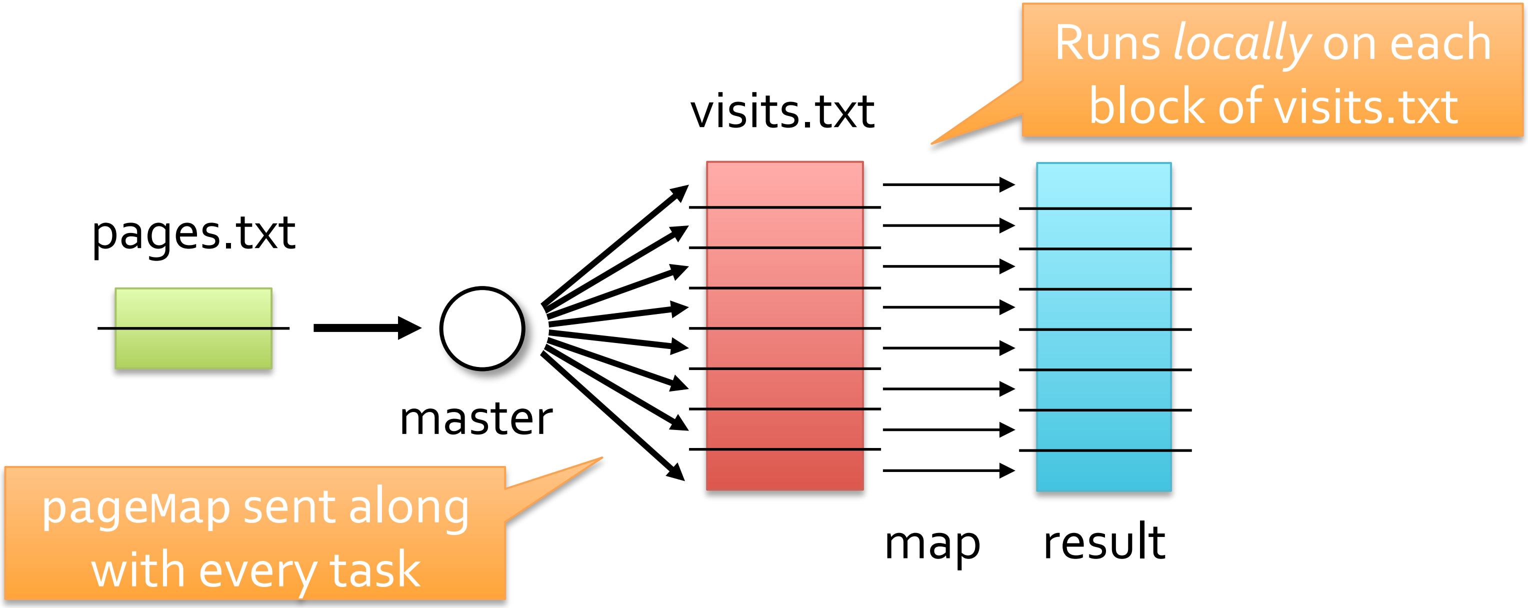


Alternative if One Table is Small

```
val pageNames = sc.textFile("pages.txt").map(...)
val pageMap = pageNames.collect().toMap()

val visits = sc.textFile("visits.txt").map(...)

val joined = visits.map(v => (v._1, (pageMap(v._1), v._2)))
```



Better Version with Broadcast

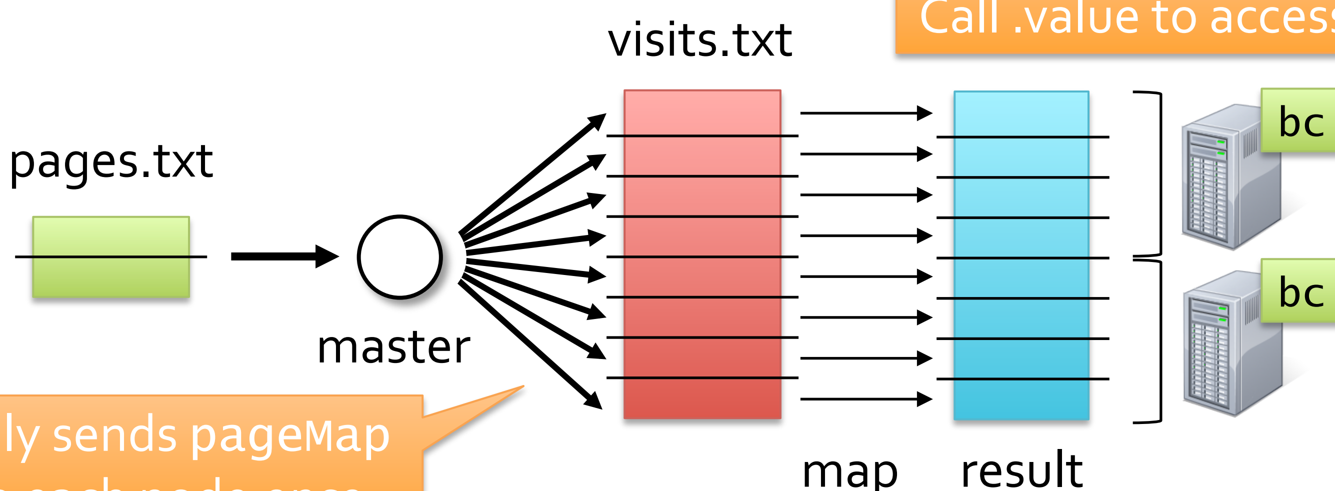
```
val pageNames = sc.textFile(“pages.txt”).map(...)
val pageMap = pageNames.collect().toMap()
val bc = sc.broadcast(pageMap)

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



Only sends pageMap to each node *once*

Broadcast Variable Rules

Create with `SparkContext.broadcast(initialVal)`

Access with `.value` inside tasks

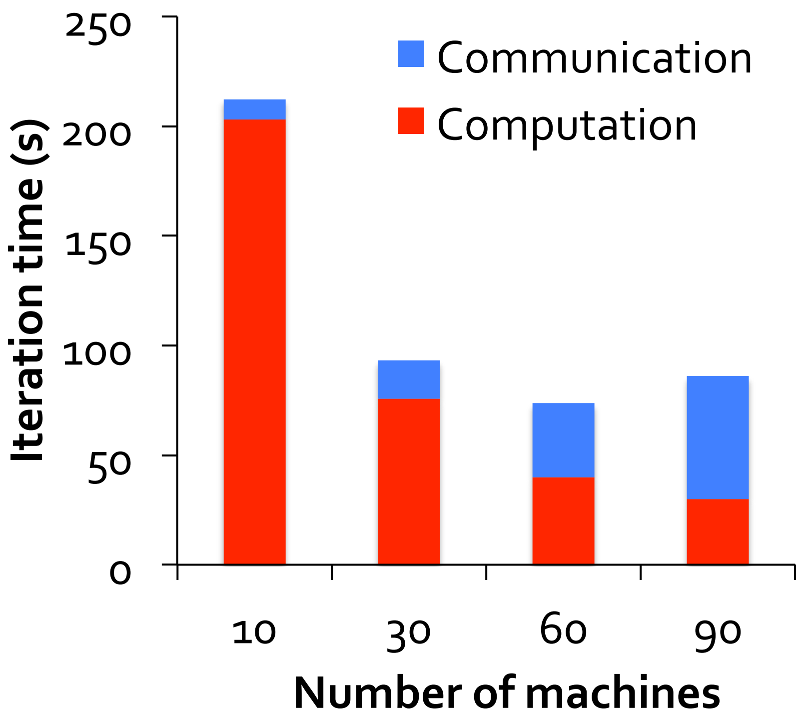
» First task to do so on each node fetches the value

Cannot modify value after creation

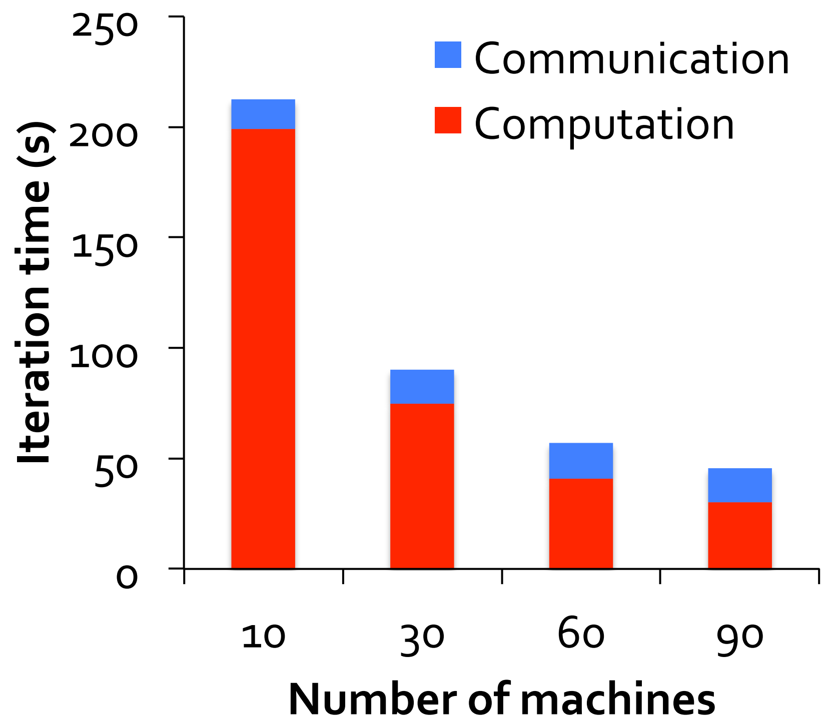
» If you try, change will only be on one node

Scaling Up Broadcast

Initial version (HDFS)

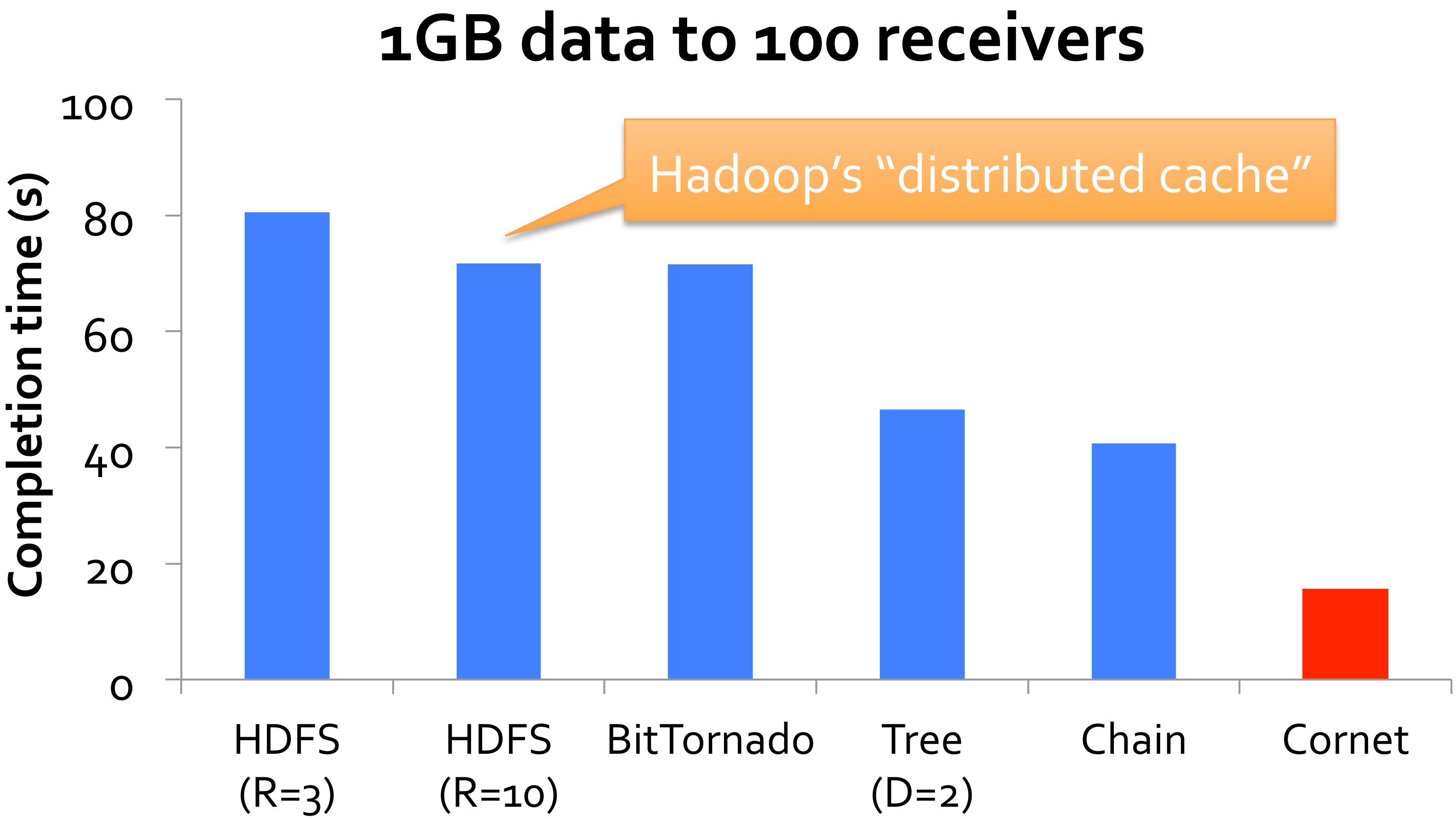


Cornet P2P broadcast



[Chowdhury et al, SIGCOMM 2011]

Cornet Performance



[Chowdhury et al, SIGCOMM 2011]

Outline

Broadcast variables

Accumulators

Controllable partitioning

Extending Spark

Motivation

Often, an application needs to aggregate multiple values as it progresses

Accumulators generalize MapReduce’s counters to enable this

Usage

```
val badRecords = sc.accumulator(0)
val badBytes = sc.accumulator(0.0)

records.filter(r => {
  if (isBad(r)) {
    badRecords += 1
    badBytes += r.size
    false
  } else {
    true
  }
}).save(...)

printf("Total bad records: %d, avg size: %f\n",
  badRecords.value, badBytes.value / badRecords.value)
```

Accumulator[Int]

Accumulator[Double]

Accumulator Rules

Create with `SparkContext.accumulator(initialVal)`

"Add" to the value with += inside tasks

» Each task's effect only counted once

Access with `.value`, but only on master

» Exception if you try it on workers

Custom Accumulators

Define an object extending `AccumulatorParam[T]`, where `T` is your data type, and providing:

» A `zero` element for a given T

» An `addInPlace` method to merge in values

```
class Vector(val data: Array[Double]) {...}

implicit object VectorAP extends AccumulatorParam[Vector] {
  def zero(v: Vector) = new Vector(new Array(v.data.size))

  def addInPlace(v1: Vector, v2: Vector) = {
    for (i <- 0 to v1.data.size-1) v1.data(i) += v2.data(i)
    return v1
  }
}
```

Now you can use `sc.accumulator(new Vector(…))`

Another Common Use

```
val sum = sc.accumulator(0.0)
val count = sc.accumulator(0.0)

records.foreach(r => {
  sum += r.size
  count += 1.0
})

val average = sum.value / count.value
```

Action that only runs for its side-effects

Outline

Broadcast variables

Accumulators

Controllable partitioning

Extending Spark

Motivation

Recall from yesterday that network bandwidth is ~100× as expensive as memory bandwidth

One way Spark avoids using it is through locality-aware scheduling for RAM and disk

Another important tool is controlling the *partitioning* of RDD contents across nodes

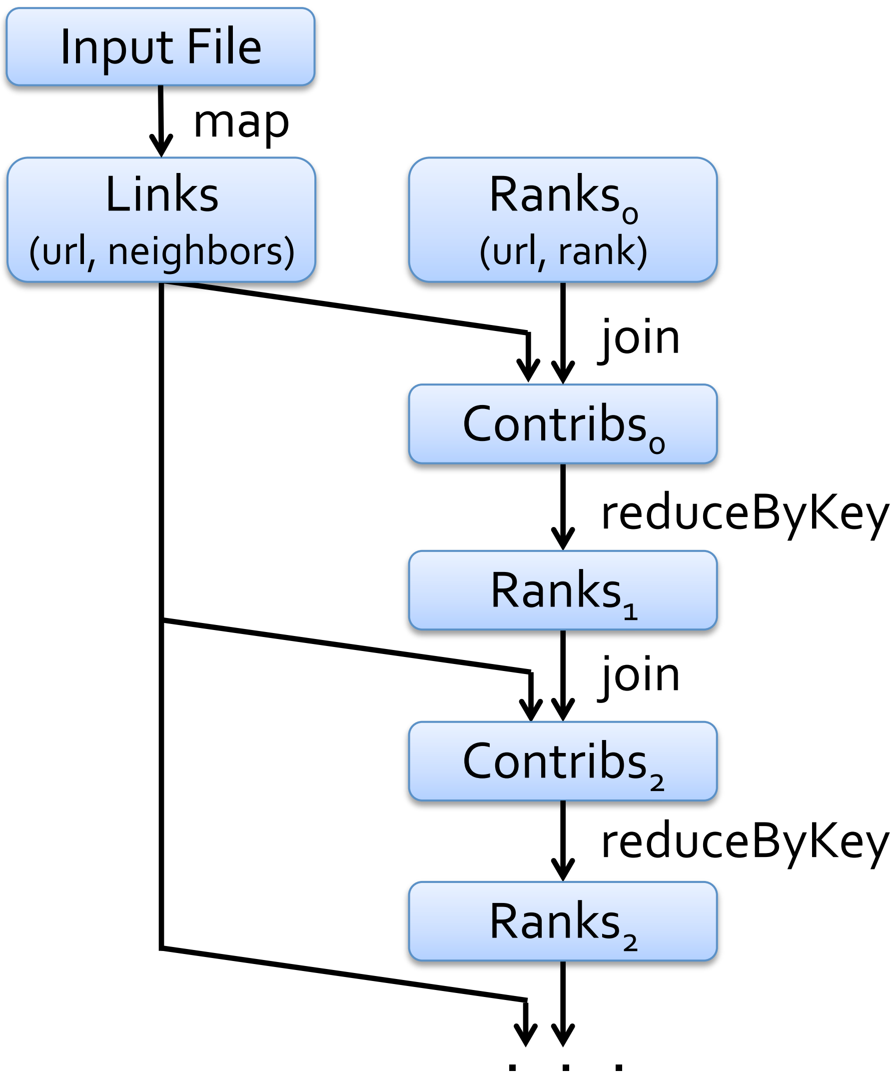
Example: PageRank

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$

```
val links = // RDD of (url, neighbors) pairs
var ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  val contribs = links.join(ranks).flatMap {
    case (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  ranks = contribs.reduceByKey(_ + _).mapValues(.15 + .85*_)
}
```

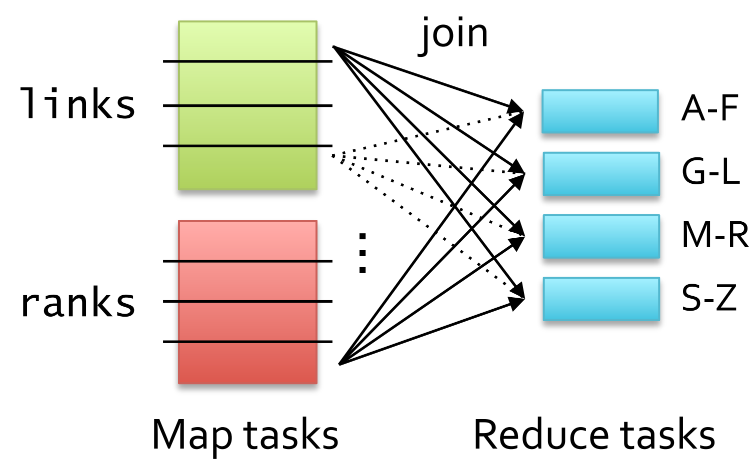
PageRank Execution



`links` and `ranks` are repeatedly joined

Each join requires a full shuffle over the network

» Hash both onto same nodes



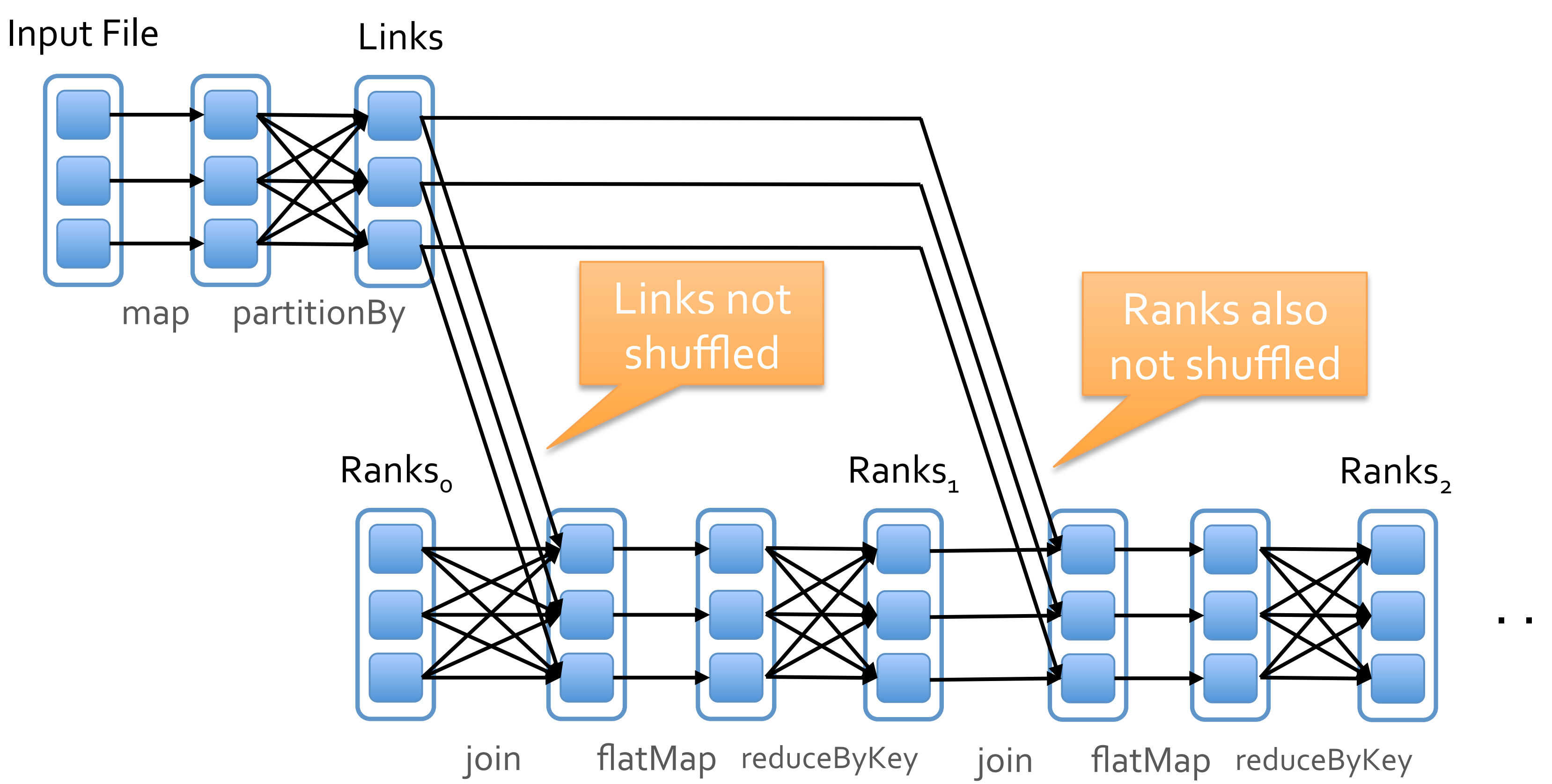
Solution

Pre-partition the `links` RDD so that links for URLs with the same hash code are on the same node

```
val ranks = // RDD of (url, rank) pairs
val links = sc.textFile(...).map(...)
              .partitionBy(new HashPartitioner(8))

for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
   .mapValues(0.15 + 0.85 * _)
}
```

New Execution



How it Works

Each RDD has an optional `Partitioner` object

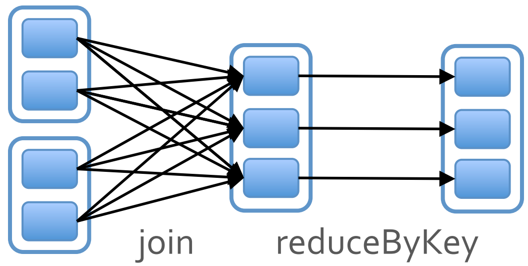
Any shuffle operation on an RDD with a `Partitioner` will respect that `Partitioner`

Any shuffle operation on two RDDs will take on the `Partitioner` of one of them, if one is set

Otherwise, by default use `HashPartitioner`

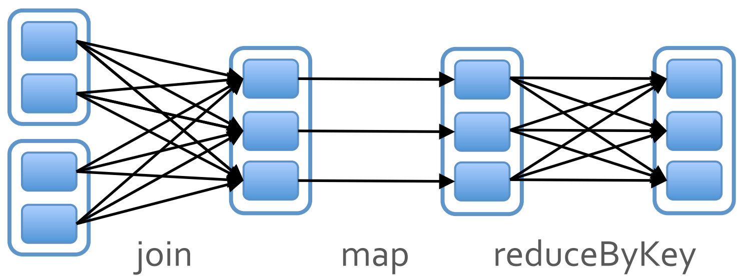
Examples

`pages.join(visits).reduceByKey(...)`



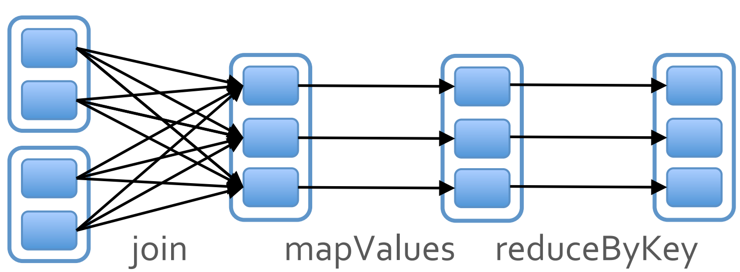
Output of join is already partitioned

`pages.join(visits).map(...).reduceByKey(...)`



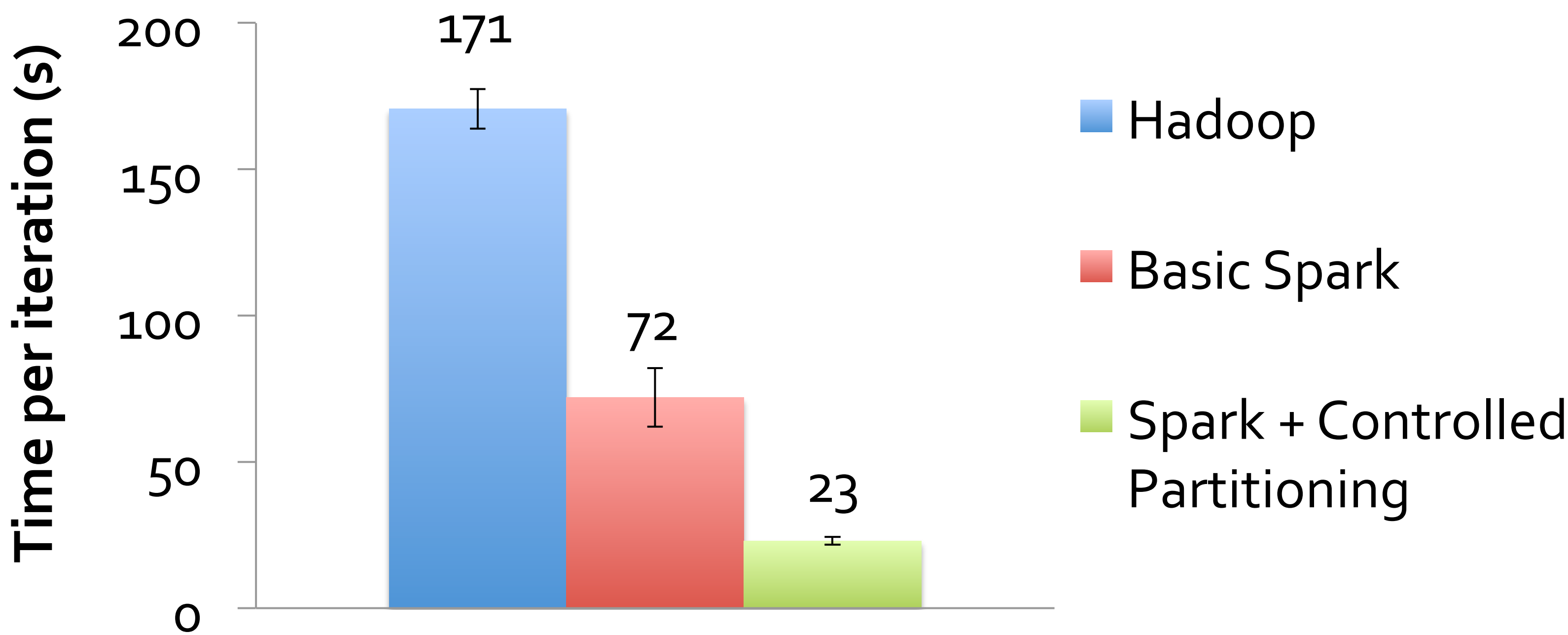
map loses knowledge about partitioning

`pages.join(visits).mapValues(...).reduceByKey(...)`



mapValues retains keys unchanged

PageRank Performance



Why it helps so much: `links` RDD is much bigger in bytes than `ranks`!

Telling How an RDD is Partitioned

Use the `.partitioner` method on RDD

```
scala> val a = sc.parallelize(List((1, 1), (2, 2)))
scala> val b = sc.parallelize(List((1, 1), (2, 2)))
scala> val joined = a.join(b)

scala> a.partitioner
res0: Option[Partitioner] = None

scala> joined.partitioner
res1: Option[Partitioner] = Some(HashPartitioner@286d41c0)
```

Custom Partitioning

Can define your own subclass of `Partitioner` to leverage domain-specific knowledge

Example: in PageRank, hash URLs by domain name, because may links are internal

```
class DomainPartitioner extends Partitioner {
  def numPartitions = 20

  def getPartition(key: Any): Int =
    parseDomain(key.toString).hashCode % numPartitions

  def equals(other: Any): Boolean =
    other.isInstanceOf[DomainPartitioner]
}
```

Needed for Spark to tell when two partitioners are equivalent

Outline

Broadcast variables

Accumulators

Controllable partitioning

Extending Spark

Extension Points

Spark provides several places to customize functionality:

Extending RDD: add new input sources or transformations

spark.cache.class: customize caching

spark.serializer: customize object storage

What People Have Done

New RDD transformations (`sample`, `glom`, `mapPartitions`, `leftOuterJoin`, `rightOuterJoin`)

New input sources (DynamoDB)

Custom serialization for memory and bandwidth efficiency

Why Change Serialization?

Greatly impacts network usage

Can also be used to improve memory efficiency

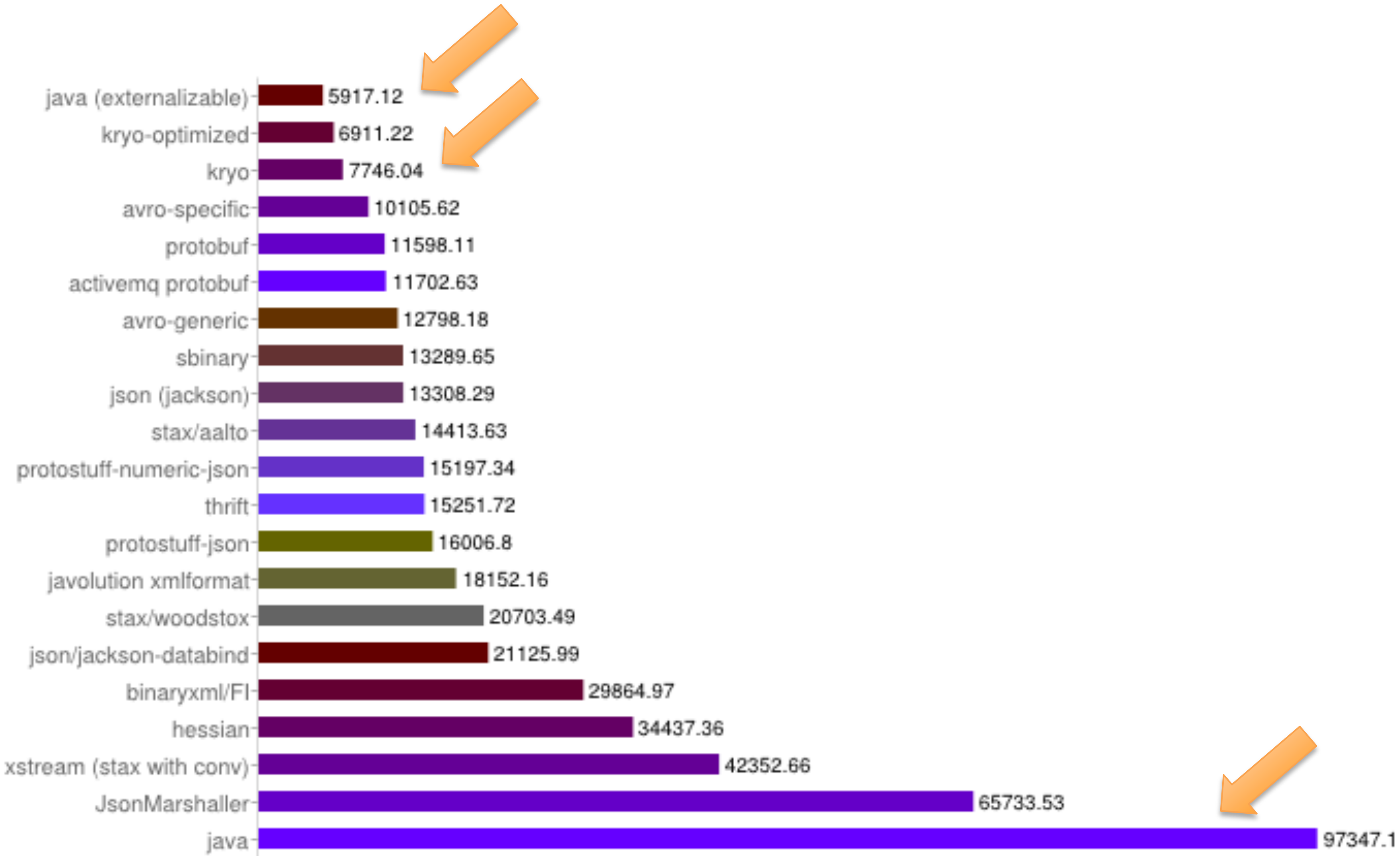
» Java objects are often larger than raw data

» Most compact way to keep large amounts of data in memory is SerializingCache

Spark's default choice of Java serialization is very simple to use, but very slow

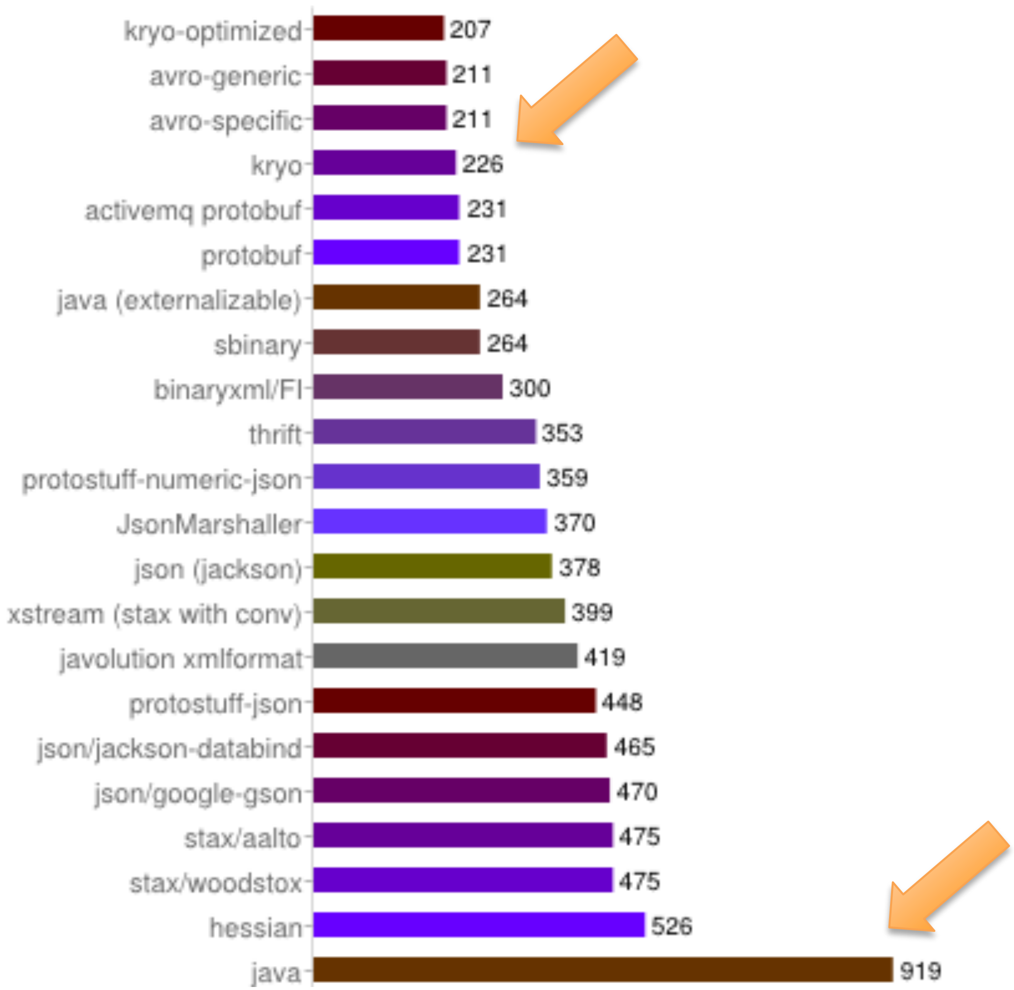
» High space & time cost due to forward compatibility

Serializer Benchmark: Time



Source: http://code.google.com/p/thrift-protobuf-compare/wiki/Benchmarking

Serializer Benchmark: Space



Source: http://code.google.com/p/thrift-protobuf-compare/wiki/Benchmarking

Better Serialization

You can implement your own serializer by extending `spark.Serializer`

But as a good option that saves a lot of time, we recommend Kryo (code.google.com/p/kryo)

- One of the fastest, but minimal boilerplate
- *Note:* Spark currently uses Kryo 1.x, not 2.x

Using Kryo

```
class MyRegistrator extends spark.KryoRegistrator {
  def registerClasses(kryo: Kryo) {
    kryo.register(classOf[Class1])
    kryo.register(classOf[Class2])
  }
}

System.setProperty(
  "spark.serializer", "spark.KryoSerializer")
System.setProperty(
  "spark.kryo.registrator", "mypkg.MyRegistrator")
System.setProperty(    // Optional, for memory usage
  "spark.cache.class", "mypkg.SerializingCache")

val sc = new SparkContext(...)
```

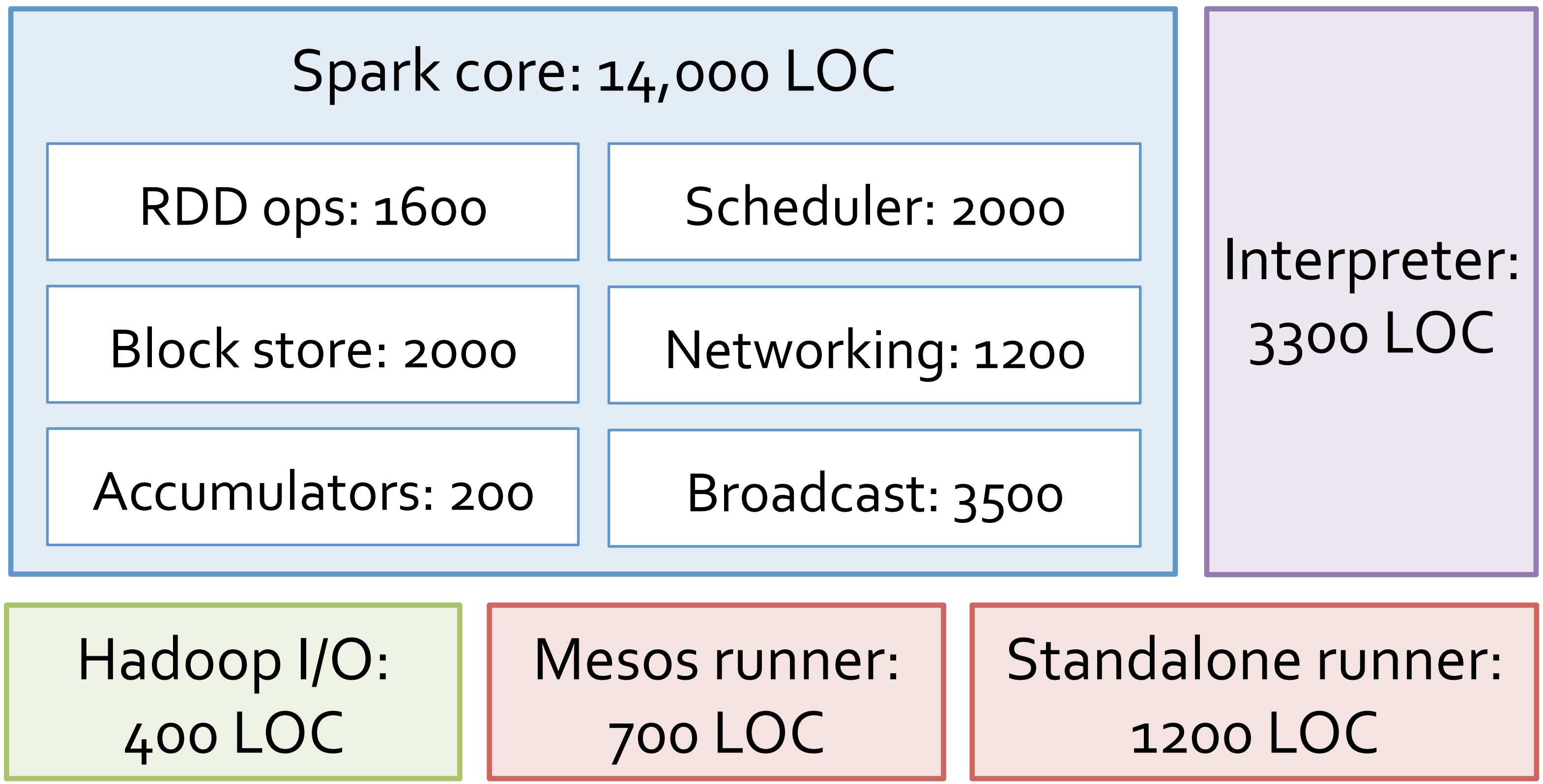
Impact of Serialization

Saw as much as 4× space reduction and 10× time reduction with Kryo

Simple way to test serialization cost in your program: profile it with `jstack` or `hprof`

We plan to work on this further in the future!

Codebase Size



Conclusion

Spark provides a variety of features to improve application performance

» Broadcast + accumulators for common sharing patterns

» Data layout through controlled partitioning

With in-memory data, the bottleneck often shifts to network or CPU

You can do more by hacking Spark itself – ask us if interested!