Advanced Spark Programming (2)

Shared variables:

- When we pass functions such map()
- Every node gets a copy of the variable
- The change to these variables is not communicated back
- After starting of the map(), changes to the variable on driver doesn't impact the worker.

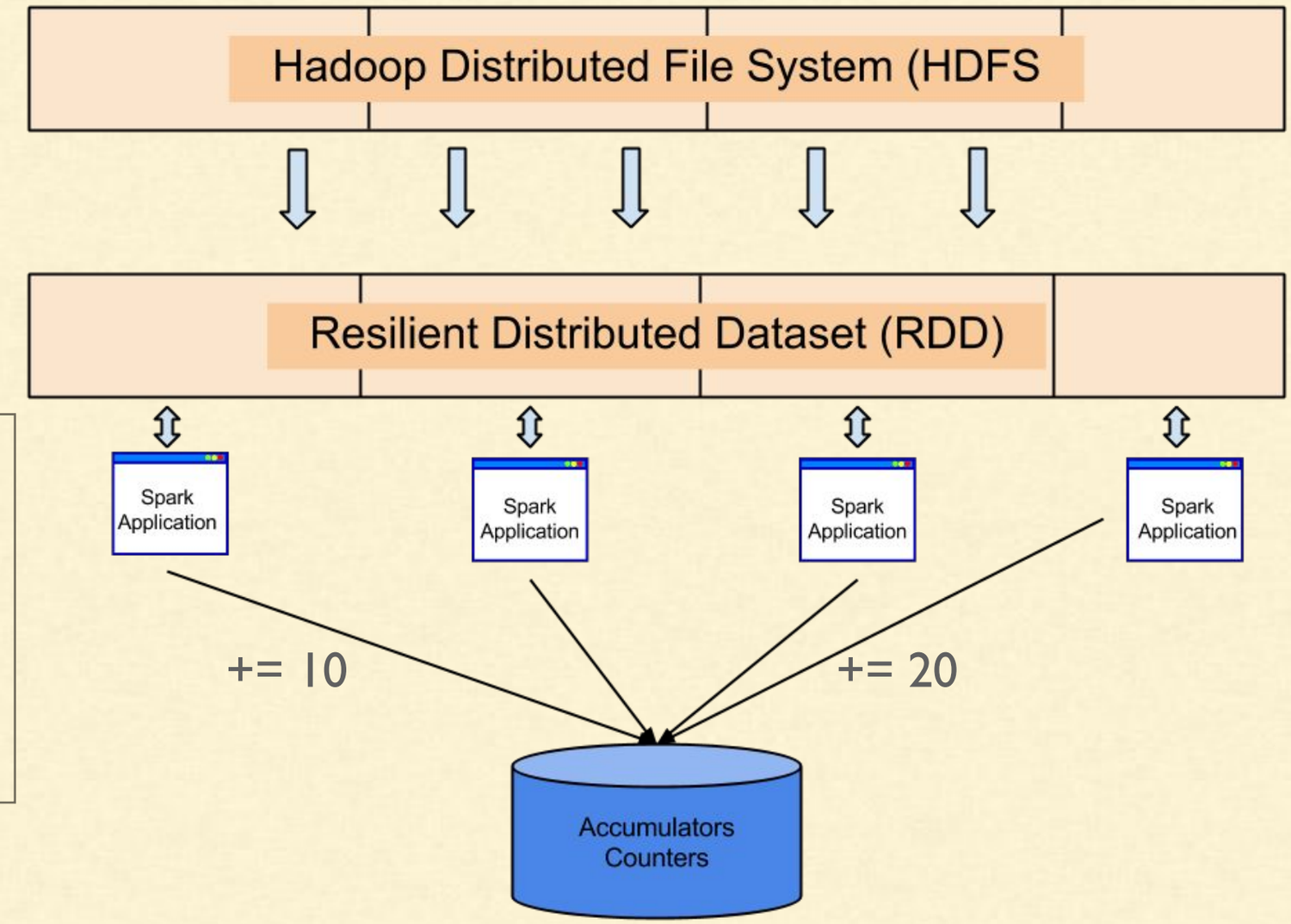
Two Kinds:

1. **Accumulators** to aggregate information
2. **Broadcast variables** to efficiently distribute large values

SHARED MEMORY - Accumulators

are only "added" to through associative operation

assoc.: 2+3+4=2+4+3=9



Accumulators

- Accumulators are variables that are only “added” to through an associative operation
- Can therefore be efficiently supported in parallel.
- They can be used to implement counters (as in MapReduce) or sums.

Accumulator : Empty line count

https://gist.github.com/girisandeep/161d1d5ea09517b1ab44df81b9b148c0

Accumulator : Empty line count

```
sc.setLogLevel("ERROR")
```

https://gist.github.com/girisandeep/161d1d5ea09517b1ab44df81b9b148c0

Accumulator : Empty line count

```
sc.setLogLevel("ERROR")
var file = sc.textFile("/data/mr/wordcount/input/")
```

https://gist.github.com/girisandeep/161d1d5ea09517b1ab44df81b9b148c0

Accumulator : Empty line count

```
sc.setLogLevel("ERROR")
var file = sc.textFile("/data/mr/wordcount/input/")
var numBlankLines = sc.accumulator(0)
```

https://gist.github.com/girisandeep/161d1d5ea09517b1ab44df81b9b148c0

Accumulator : Empty line count

```
sc.setLogLevel("ERROR")
var file = sc.textFile("/data/mr/wordcount/input/")
var numBlankLines = sc.accumulator(0)

def toWords(line:String): Array[String] = {
  if(line.length == 0) {numBlankLines += 1}
  return line.split(" ");
}
```

https://gist.github.com/girisandeep/161d1d5ea09517b1ab44df81b9b148c0

Accumulator : Empty line count

```
sc.setLogLevel("ERROR")
var file = sc.textFile("/data/mr/wordcount/input/")
var numBlankLines = sc.accumulator(0)

def toWords(line:String): Array[String] = {
  if(line.length == 0) {numBlankLines += 1}
  return line.split(" ");
}

var words = file.flatMap(toWords)
words.saveAsTextFile("words3")
```

https://gist.github.com/girisandeep/161d1d5ea09517b1ab44df81b9b148c0

Accumulator : Empty line count

```
sc.setLogLevel("ERROR")
var file = sc.textFile("/data/mr/wordcount/input/")
var numBlankLines = sc.accumulator(0)

def toWords(line:String): Array[String] = {
  if(line.length == 0) {numBlankLines += 1}
  return line.split(" ");
}

var words = file.flatMap(toWords)
words.saveAsTextFile("words3")
printf("Blank lines: %d", numBlankLines.value)
//Blank lines: 24857
```

https://gist.github.com/girisandeep/161d1d5ea09517b1ab44df81b9b148c0

Accumulators and Fault Tolerance

- Spark Re-executes failed or slow tasks.
- Preemptively launches “speculative” copy of slow worker task

The net result is ???

Accumulators and Fault Tolerance

- Spark Re-executes failed or slow tasks.
- Preemptively launches “speculative” copy of slow worker task

The net result is: The same function may run multiple times on the same data.

Accumulators and Fault Tolerance

- Spark Re-executes failed or slow tasks.
- Preemptively launches “speculative” copy of slow worker task

The net result is: The same function may run multiple times on the same data.

Does it mean accumulators will give wrong result?

Accumulators and Fault Tolerance

- Spark Re-executes failed or slow tasks.
- Preemptively launches “speculative” copy of slow worker task

The net result is: The same function may run multiple times on the same data.

Does it mean accumulators will give wrong result?
YES, for accumulators in Transformation.
No, for accumulators in Action

Accumulators and Fault Tolerance

- For accumulators in actions, Each task's accumulator update applied once.
- For reliable absolute value counter, put it inside an action
- In transformations, this guarantee doesn't exist.
- In transformations, use accumulators for debug only.

Custom Accumulators

- Out of the box, Spark supports accumulators of type Double, Long, and Float.
- Spark also includes an API to define custom accumulator types and custom aggregation operations
  - (e.g., finding the maximum of the accumulated values instead of adding them).
- Custom accumulators need to extend AccumulatorV2.

Custom Accumulators - version 1.x

Custom Accumulators - version 1.x

```
class MyComplex(var x: Int, var y: Int) extends Serializable{
  def reset(): Unit = {
    x = 0
    y = 0
  }
  def add(p:MyComplex): MyComplex = {
    x = x + p.x
    y = y + p.y
    return this
  }
}
```

https://gist.github.com/girisandeep/450ff3d29f20f2e31cdd09ad0f1c0df2

Custom Accumulators - version 1.x

```
import org.apache.spark.AccumulatorParam
class ComplexAccumulatorV1 extends AccumulatorParam[MyComplex] {

   def zero(initialVal: MyComplex): MyComplex = {
      return initialVal
   }

   def addInPlace(v1: MyComplex, v2: MyComplex): MyComplex = {
      v1.add(v2)
      return v1;
   }
}
```

https://gist.github.com/girisandeep/450ff3d29f20f2e31cdd09ad0f1c0df2

Custom Accumulators - version 1.x

```
val vecAccum = sc.accumulator(new MyComplex(0,0))(new ComplexAccumulatorV1)
```

https://gist.github.com/girisandeep/450ff3d29f20f2e31cdd09ad0f1c0df2

Custom Accumulators - version 1.x

```
val vecAccum = sc.accumulator(new MyComplex(0,0))(new ComplexAccumulatorV1)

var myrdd = sc.parallelize(Array(1,2,3))
def myfunc(x:Int):Int = {
   vecAccum += new MyComplex(x, x)
   return x * 3
}
var myrdd1 = myrdd.map(myfunc)
```

https://gist.github.com/girisandeep/450ff3d29f20f2e31cdd09ad0f1c0df2

Custom Accumulators - version 1.x

```
val vecAccum = sc.accumulator(new MyComplex(0,0))(new ComplexAccumulatorV1)

var myrdd = sc.parallelize(Array(1,2,3))
def myfunc(x:Int):Int = {
   vecAccum += new MyComplex(x, x)
   return x * 3
}
var myrdd1 = myrdd.map(myfunc)
myrdd1.collect()
vecAccum.value.x
vecAccum.value.y
```

https://gist.github.com/girisandeep/450ff3d29f20f2e31cdd09ad0f1c0df2

Custom Accumulators - version 2.x

```
import org.apache.spark.util.AccumulatorV2
object ComplexAccumulatorV2 extends AccumulatorV2[MyComplex, MyComplex] {
   private val myc:MyComplex = new MyComplex(0,0)
   def reset(): Unit = {
      myc.reset()
   }
   def add(v: MyComplex): Unit = {
      myc.add(v)
   }
   def value():MyComplex = {
      return myc
   }
   def isZero(): Boolean = {
      return (myc.x == 0 && myc.y == 0)
   }
   def copy():AccumulatorV2[MyComplex, MyComplex] = {
      return ComplexAccumulatorV2
   }
   def merge(other:AccumulatorV2[MyComplex, MyComplex]) = {
      myc.add(other.value)
   }
}
sc.register(ComplexAccumulatorV2, "mycomplexacc")
```

https://gist.github.com/girisandeep/35b21cca890157afe0084a9e400e2e70

Broadcast Variables : Introduction

```
commonWords = ["a", "an", "the", "of", "at", "is",
"am", "are", "this", "that", '', 'at']
```

If we need to remove the common words from our wordcount, what do we need to do?

Broadcast Variables : Introduction

```
commonWords = ["a", "an", "the", "of", "at", "is",
"am", "are", "this", "that", '', 'at']

If we need to remove the common words from our
wordcount, what do we need to do?

> We can create a local variable and use it
```

Broadcast Variables : Introduction

```
commonWords = ["a", "an", "the", "of", "at", "is",
"am", "are", "this", "that", '', 'at']

If we need to remove the common words from our
wordcount, what do we need to do?

> We can create a local variable and use it

> Is it inefficient?
```

Broadcast Variables : Introduction

Yes, because

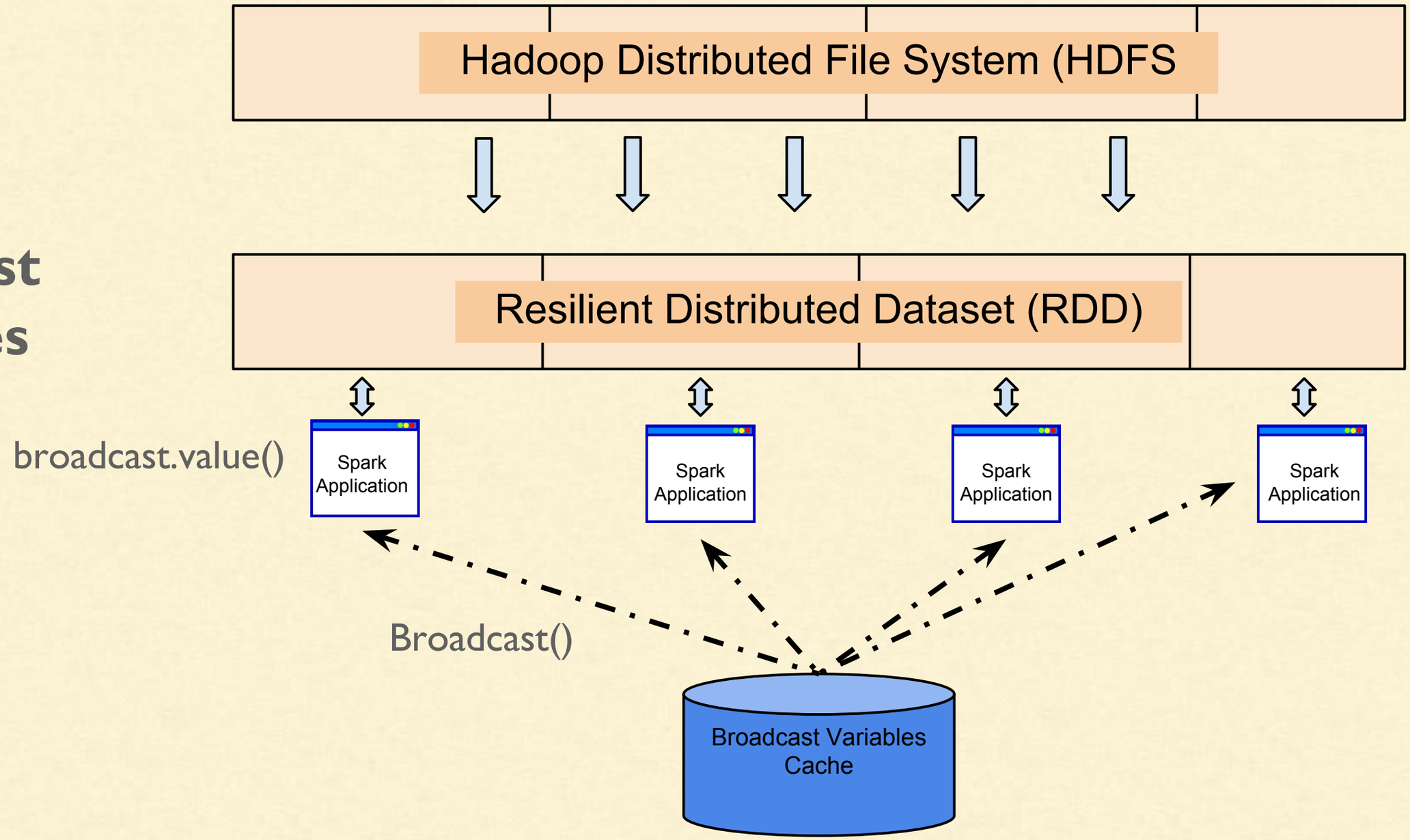
1. Spark sends referenced variables to all workers.
1. The default task launching mechanism is optimised for small task sizes.
2. If using multiple times, spark will be sending it again to all nodes

So, we use broadcast variable instead.

SHARED MEMORY

Broadcast Variables

broadcast.value()



Broadcast Variables

- Efficiently send a large, read-only value to workers

Broadcast Variables

- Efficiently send a large, read-only value to workers
- For example:
  - Send a large, read-only lookup table to all the nodes

Broadcast Variables

- Efficiently send a large, read-only value to workers
- For example:
  - Send a large, read-only lookup table to all the nodes
  - Large feature vector in a machine learning algorithm

Broadcast Variables

- Efficiently send a large, read-only value to workers
- For example:
  - Send a large, read-only lookup table to all the nodes
  - Large feature vector in a machine learning algorithm
- It is like a distributed cache of Hadoop
- Spark distributes broadcast variables efficiently to reduce communication cost.

Broadcast Variables

- Efficiently send a large, read-only value to workers
- For example:
  - Send a large, read-only lookup table to all the nodes
  - Large feature vector in a machine learning algorithm
- It is like a distributed cache of Hadoop
- Spark distributes broadcast variables efficiently to reduce communication cost.
- Useful when
  - Tasks across multiple stages need the same data
  - Caching the data in deserialized form is important.

Broadcast Variables : Example

Removing Common Words using Broadcast.

https://gist.github.com/girisandeep/f12ab4bf2536dc5f0a8ca673efbac1db

Broadcast Variables : Example

Removing Common Words using Broadcast.

```
var commonWords = Array("a", "an", "the", "of", "at", "is", "am","are","this","that","at",
"in", "or", "and", "or", "not", "be", "for", "to", "it")
```

https://gist.github.com/girisandeep/f12ab4bf2536dc5f0a8ca673efbac1db

Broadcast Variables : Example

Removing Common Words using Broadcast.

```
var commonWords = Array("a", "an", "the", "of", "at", "is", "am","are","this","that","at",
"in", "or", "and", "or", "not", "be", "for", "to", "it")
val commonWordsMap = collection.mutable.Map[String, Int]()
for(word <- commonWords){
    commonWordsMap(word) = 1
}
var commonWordsBC = sc.broadcast(commonWordsMap)
```

https://gist.github.com/girisandeep/f12ab4bf2536dc5f0a8ca673efbac1db

Broadcast Variables : Example

Removing Common Words using Broadcast.

```
var commonWords = Array("a", "an", "the", "of", "at", "is", "am","are","this","that","at",
"in", "or", "and", "or", "not", "be", "for", "to", "it")
val commonWordsMap = collection.mutable.Map[String, Int]()
for(word <- commonWords){
    commonWordsMap(word) = 1
}
var commonWordsBC = sc.broadcast(commonWordsMap)

var file = sc.textFile("/data/mr/wordcount/input/big.txt")
```

https://gist.github.com/girisandeep/f12ab4bf2536dc5f0a8ca673efbac1db

Broadcast Variables : Example

Removing Common Words using Broadcast.

```
var commonWords = Array("a", "an", "the", "of", "at", "is", "am","are","this","that","at",
"in", "or", "and", "or", "not", "be", "for", "to", "it")
val commonWordsMap = collection.mutable.Map[String, Int]()
for(word <- commonWords){
    commonWordsMap(word) = 1
}
var commonWordsBC = sc.broadcast(commonWordsMap)

var file = sc.textFile("/data/mr/wordcount/input/big.txt")
def toWords(line:String):Array[String] = {
    var words = line.split(" ")
    var output = Array[String]();
    for(word <- words){
        if(! (commonWordsBC.value contains word.toLowerCase.trim.replaceAll("[^a-z]","")))
output = output :+ word;
    }
    return output;
}
var uncommonWords = file.flatMap(toWords)
```

https://gist.github.com/girisandeep/f12ab4bf2536dc5f0a8ca673efbac1db

Broadcast Variables : Example

Removing Common Words using Broadcast.

```
var commonWords = Array("a", "an", "the", "of", "at", "is", "am","are","this","that","at",
"in", "or", "and", "or", "not", "be", "for", "to", "it")
val commonWordsMap = collection.mutable.Map[String, Int]()
for(word <- commonWords){
    commonWordsMap(word) = 1
}
var commonWordsBC = sc.broadcast(commonWordsMap)

var file = sc.textFile("/data/mr/wordcount/input/big.txt")
def toWords(line:String):Array[String] = {
    var words = line.split(" ")
    var output = Array[String]();
    for(word <- words){
        if(! (commonWordsBC.value contains word.toLowerCase.trim.replaceAll("[^a-z]","")))
output = output :+ word;
    }
    return output;
}
var uncommonWords = file.flatMap(toWords)
uncommonWords.take(100)
```

https://gist.github.com/girisandeep/f12ab4bf2536dc5f0a8ca673efbac1db

Key Performance Considerations

1. Level of Parallelism
2. Serialization Format
3. Memory Management
4. Hardware Provisioning

Level of Parallelism

By Default

- A single task per one partition,
- A single core in the cluster to execute.
- Default partitions are based on underlying storage or CPU
- HDFS RDDs - One partition per block

Level of Parallelism

By Default

- A single task per one partition,
- A single core in the cluster to execute.
- Default partitions are based on underlying storage or CPU
- HDFS RDDs - One partition per block

Too Less ⇒ Might leave resources idle

Too Much ⇒ Small overheads due to each partition adds up

Key Performance Considerations

1. Level of Parallelism - How many default partitions?

Key Performance Considerations - Partitions

$ ***hadoop fs -ls /data/msprojects/in_table.csv***
-rw-r--r-- 3 sandeep sandeep 8303338297 2017-04-18 02:26 /data/msprojects/in_table.csv

$ ***python***
>>> 8303338297.0/128.0/1024.0/1024.0
61.86469120532274
>>>

/data/msprojects/in_table.csv has 62 blocks theoratically. Lets check.

$ ***hdfs fsck /data/msprojects/in_table.csv***
.....
Total blocks (validated): **62** (avg. block size 133924811 B)

Yes, it has 62 blocks actually.

Key Performance Considerations - Partitions

$ ***spark-shell --packages net.sf.opencsv:opencsv:2.3 --master yarn***

scala> var myrdd = sc.textFile("/data/msprojects/in_table.csv")
scala> myrdd.partitions.length
res1: Int = **62**

So, number of partitions is a function of number of data blocks in case of sc.textFile.

Key Performance Considerations - Partitions

```
scala> var myrdd = sc.parallelize(1 to 100000)
scala> myrdd.partitions.length
res1: Int = 4

[sandeep@ip-172-31-60-179 ~]$ cat /proc/cpuinfo|grep processor
processor       : 0
processor       : 1
processor       : 2
processor       : 3
```

Since my machine has 4 cores, it has created 4 partitions.

Key Performance Considerations - Partitions

$ ***spark-shell --master yarn***

scala> ***var myrdd = sc.parallelize(1 to 100000)***
scala> **myrdd.partitions.length**
res6: Int = **2**

When we are running in yarn mode, the number of partitions is function of tasks that can be executed on a node, Here it is 2.

Level of Parallelism

How to control parallelism?

1. Specify number of partitions in sc.parallelize and sc.textFile
2. Shuffling operations accept degree of parallelism in parameter
3. repartition()
4. To efficiently shrink, prefer coalesce() over repartition()

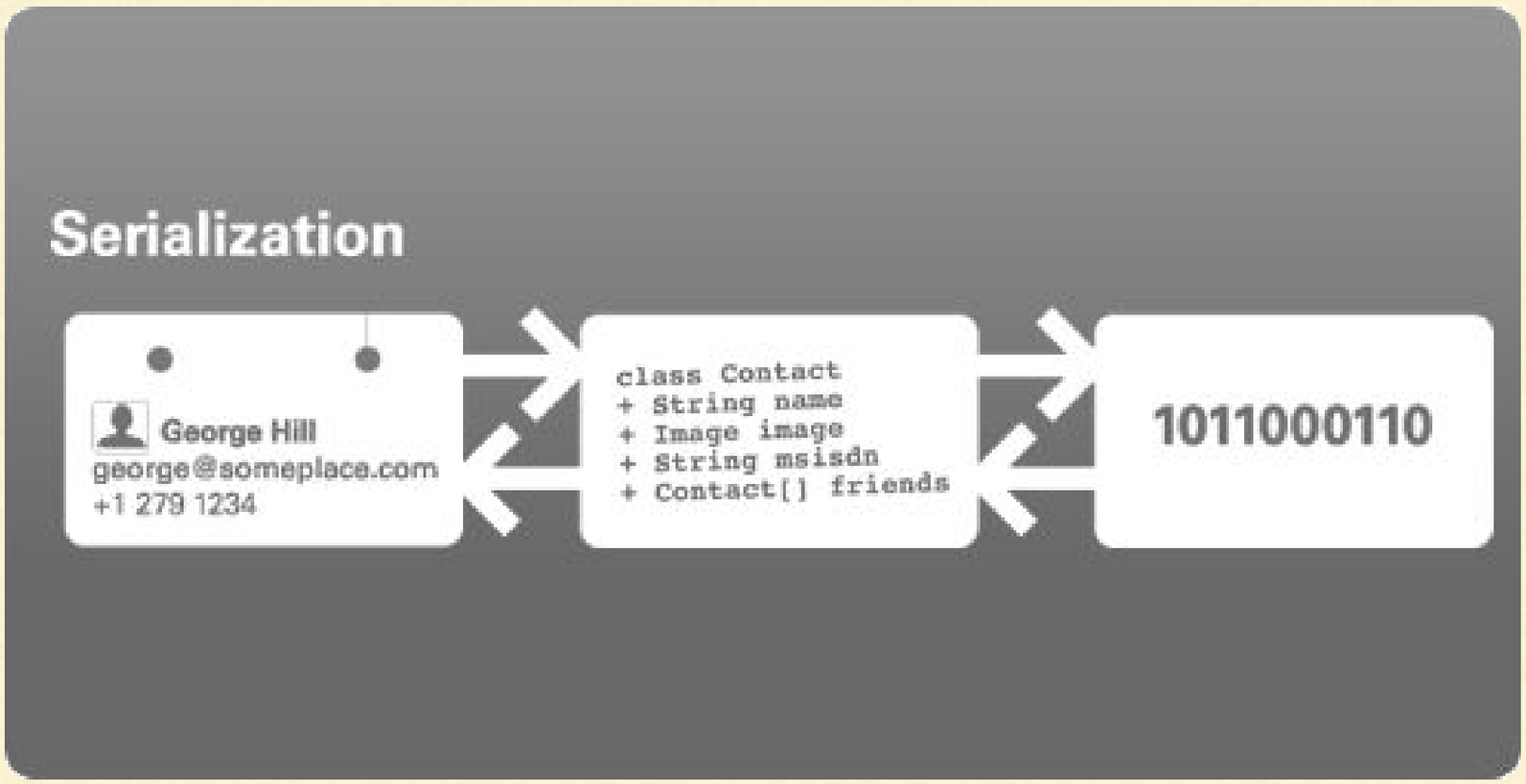
Level of Parallelism

Example

1. We are reading a large amount of data from S3.
2. filter() operation is likely to leave a tiny fraction
3. Result of filter() will have same size RDD as parent but with many empty or small partitions.
4. Improve the application's performance by coalescing

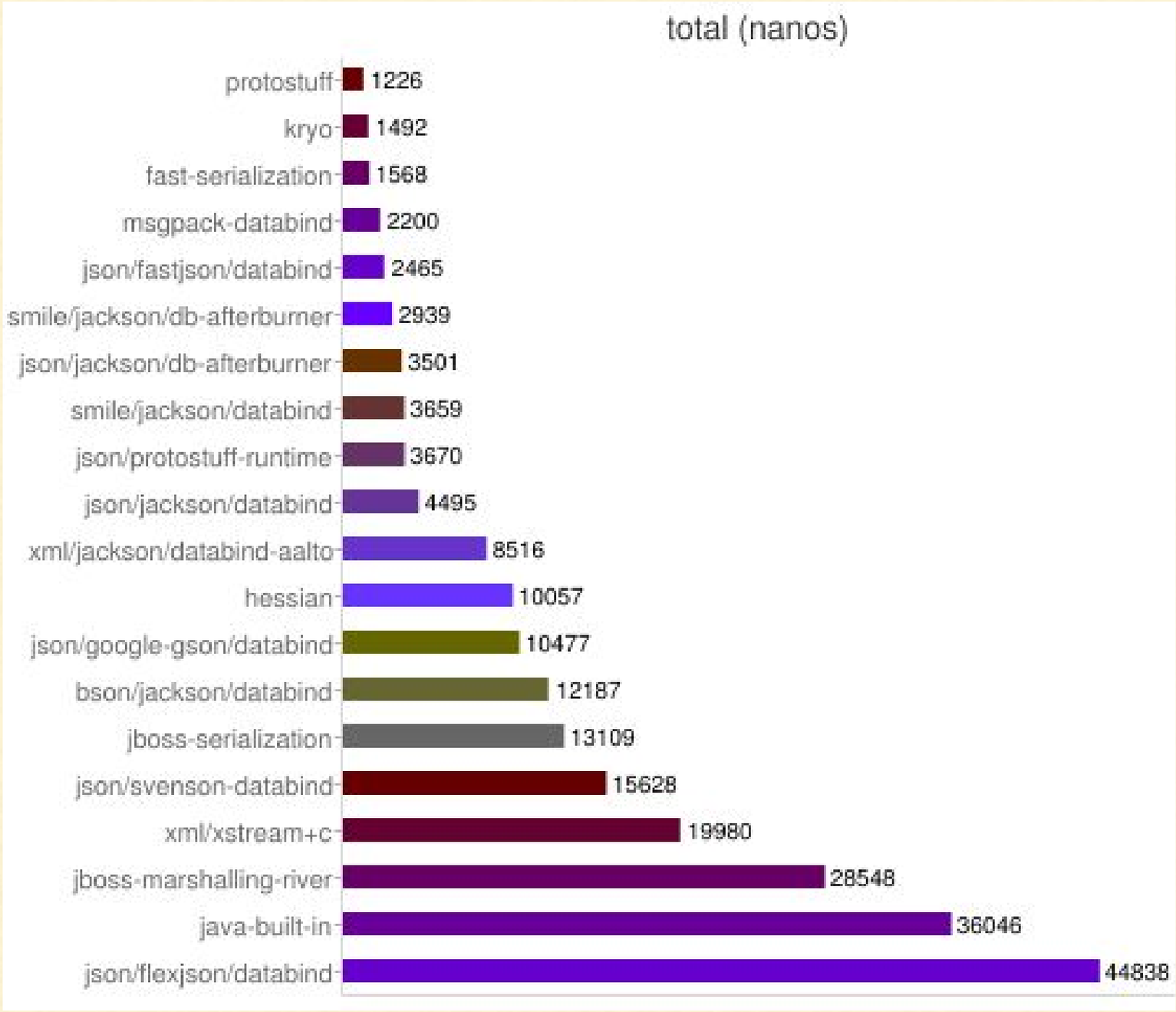
Serialization Format

- While transfering or saving objects need serialization
- Comes into play during large transfers
- By default Spark will use Java's built-in serializer.



Serialization Format

Benchmarks



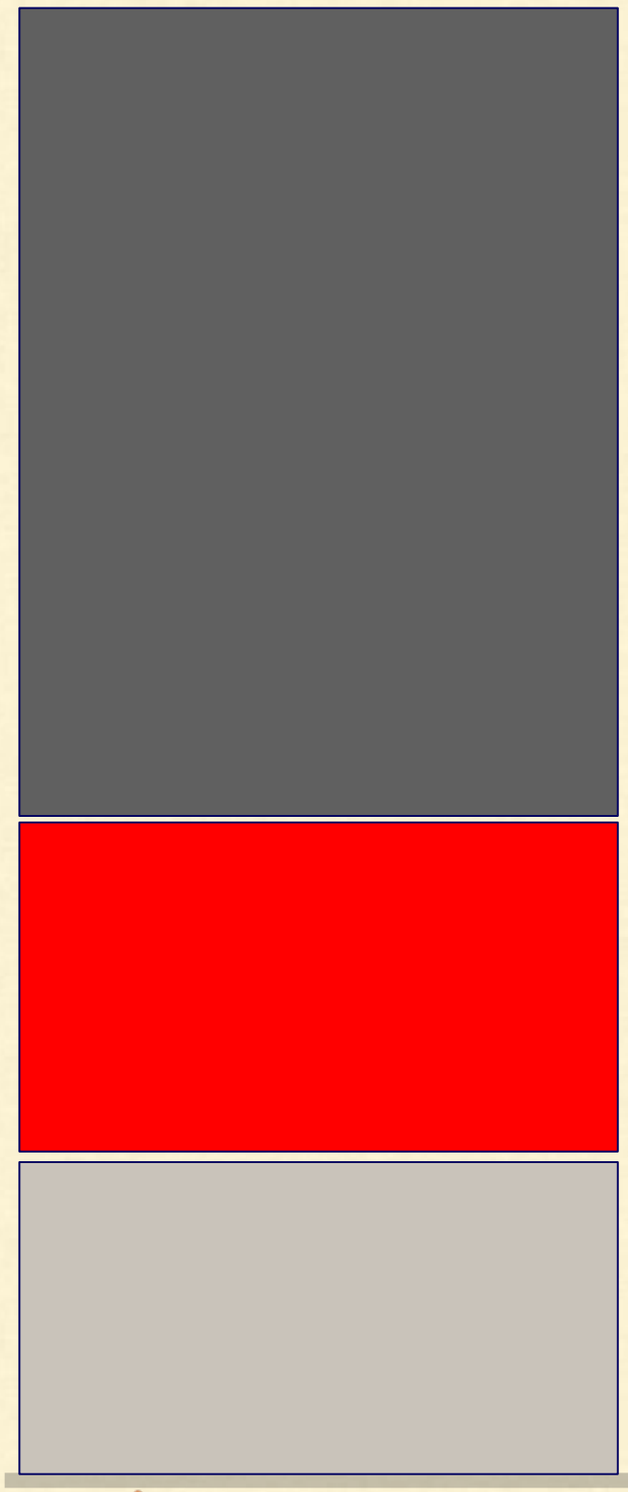
https://code.google.com/p/thrift-protobuf-compare/wiki/Benchmarking

Serialization Format

Kryo

- Spark also supports the use of Kryo
- Faster and more compact
- But cannot serialize all types of objects “out of the box.”
- Almost all applications will benefit from shifting to Kryo
- To use,
  - sc.getConf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
- For best performance, register classes with Kryo
  - sc.getConf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
  - Class needs to implement Java’s Serializable interface

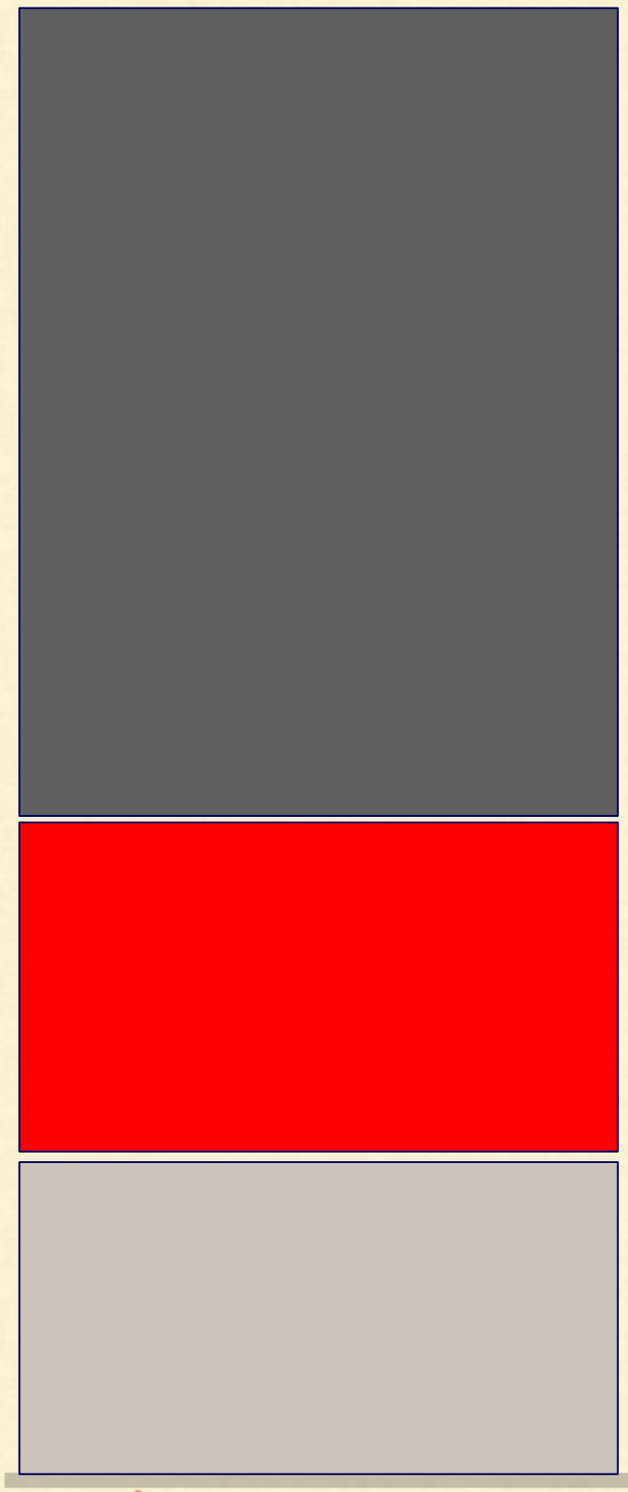
Memory Management



RDD storage

- persist()'ed memory
- spark.storage.memoryFraction - Default: 60%
- If exceeded, older will be dropped
  - will be computed on demand
- For huge data, use persist() with MEMORY_AND_DISK

Memory Management



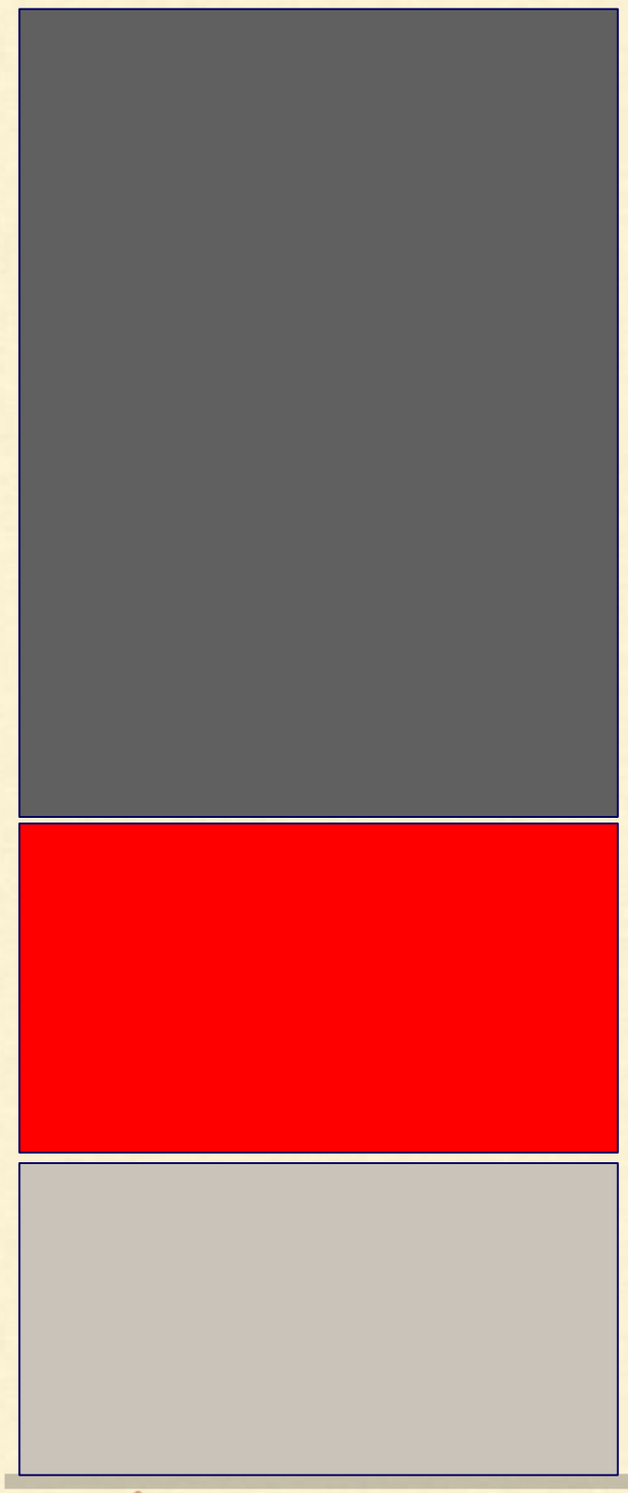
RDD storage

- persist()'ed memory
- spark.storage.memoryFraction - Default: 60%
- If exceeded, older will be dropped
  - will be computed on demand
- For huge data, use persist() with MEMORY_AND_DISK

Shuffle and aggregation buffers

- For storing shuffle output data
- spark.shuffle.memoryFraction - Default: 20%

Memory Management



RDD storage

- persist()'ed memory
- spark.storage.memoryFraction - Default: 60%
- If exceeded, older will be dropped
  - will be computed on demand
- For huge data, use persist() with MEMORY_AND_DISK

Shuffle and aggregation buffers

- For storing shuffle output data
- spark.shuffle.memoryFraction - Default: 20%

User Code

Remaining

Default: 20% of memory

Hardware Provisioning

- Main Parameters
  - Executor's Memory (spark.executor.memory)
  - Number of cores per Executor,
  - Total number of executors
  - No. of disks

Hardware Provisioning

- Main Parameters
  - Executor's Memory (spark.executor.memory)
  - Number of cores per Executor,
  - Total number of executors
  - No. of disks
- App Speed = (Impact of Memory + Cores)
  - Huge memory -> GC pauses
  - 64GB or less

Hardware Provisioning

- Main Parameters
  - Executor's Memory (spark.executor.memory)
  - Number of cores per Executor,
  - Total number of executors
  - No. of disks
- App Speed = (Impact of Memory + Cores)
  - Huge memory -> GC pauses
  - 64GB or less
- Linear scaling
  - 2 x Hardware == 2 x speed

Advanced Programming

Thank you!

Level of Parallelism

Example: Coalescing a large RDD

```
# Wildcard input that may match thousands of files
>>> input = sc.textFile("s3n://log-files/2014/*.log")
>>> input.getNumPartitions()
35154
# A filter that excludes almost all data
>>> lines = input.filter(lambda line: line.startswith("2014-10-17"))
>>> lines.getNumPartitions()
35154
# We coalesce the lines RDD before caching
>>> lines = lines.coalesce(5).cache()
>>> lines.getNumPartitions()
4
# Subsequent analysis can operate on the coalesced RDD...
>>> lines.count()
```

Level of Parallelism

Example: Coalescing a large RDD

```
» var myrdd = sc.textFile("/data/msprojects/in_table.csv")
» myrdd.partitions.size
2
» rdd1 = rdd.filter(lambda line: line.lower().startswith('this'));
» rdd1.getNumPartitions()
2
» rdd1.count()
```

Took 1.473270 s

```
» rdd = sc.textFile("hdfs://a.cloudxlab.com/data/mr/wordcount/input");
» rdd1 = rdd.filter(lambda line: line.lower().startswith('this'));
» rdd1 = rdd1.coalesce(1)
» rdd1.count()
»
```

Took 1.081873 s

Level of Parallelism

Example: Coalescing a large RDD

```
» rdd = sc.textFile("hdfs://a.cloudxlab.com/data/mr/wordcount/input");
» rdd.getNumPartitions()
2
» rdd1 = rdd.filter(lambda line: line.lower().startswith('this'));
» rdd1.getNumPartitions()
2
» rdd1.count()
```

Took 1.473270 s

```
» rdd = sc.textFile("hdfs://a.cloudxlab.com/data/mr/wordcount/input");
» rdd1 = rdd.filter(lambda line: line.lower().startswith('this'));
» rdd1 = rdd1.coalesce(1)
» rdd1.count()
»
```

Took 1.081873 s