

What is Spark?

Fast and Expressive Cluster Computing System
Compatible with Apache Hadoop

Efficient

- General execution graphs
- In-memory storage

Up to 10x faster on disk,
100x in memory

Usable

- Rich APIs in Java, Scala, Python, R
- Interactive shell

2-5x less code

Key Concepts

Write programs in terms of transformations on distributed datasets

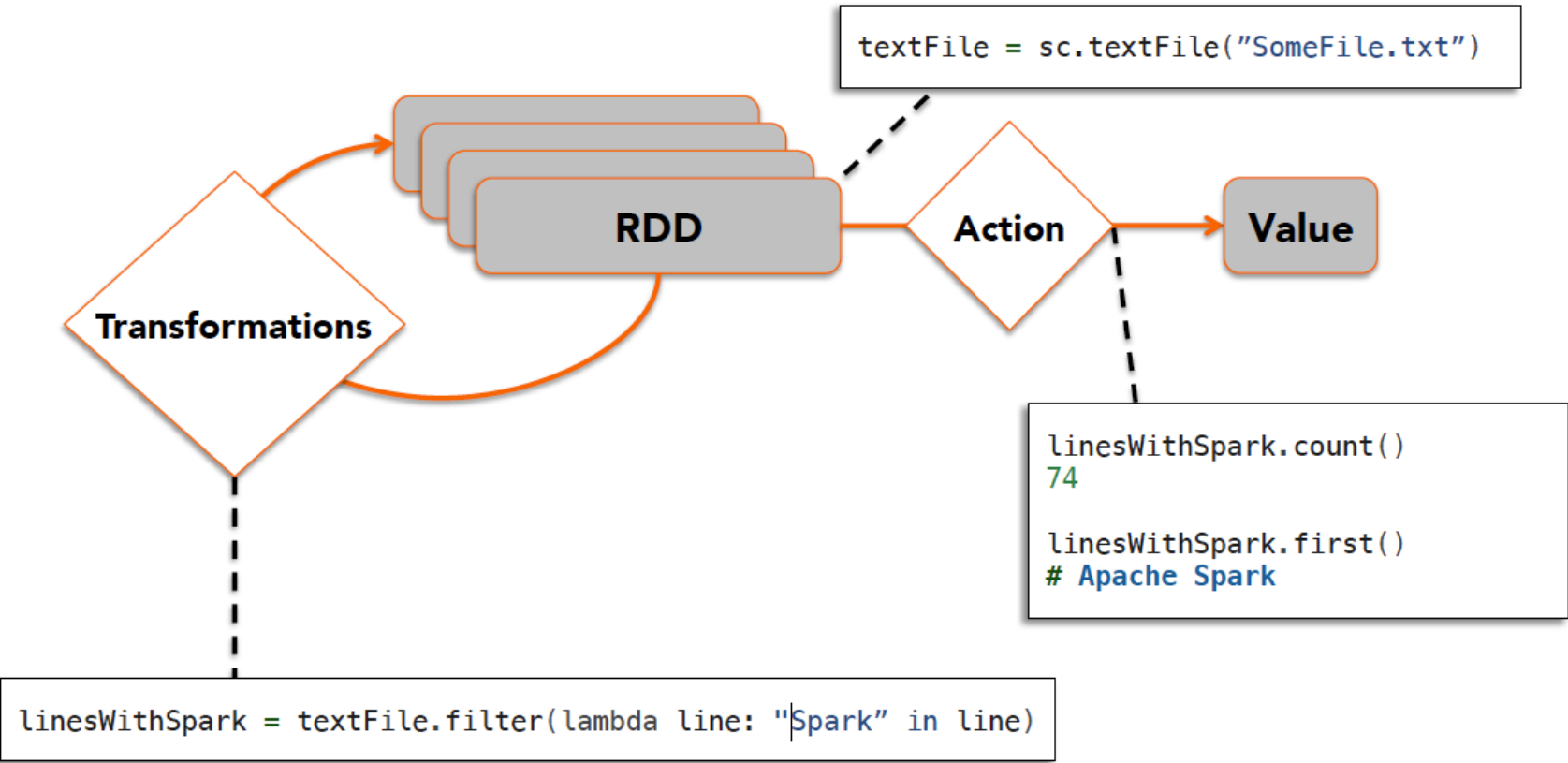
Resilient Distributed Datasets

- Collections of objects spread across a cluster, stored in RAM or on Disk
- Built through parallel transformations
- Automatically rebuilt on failure

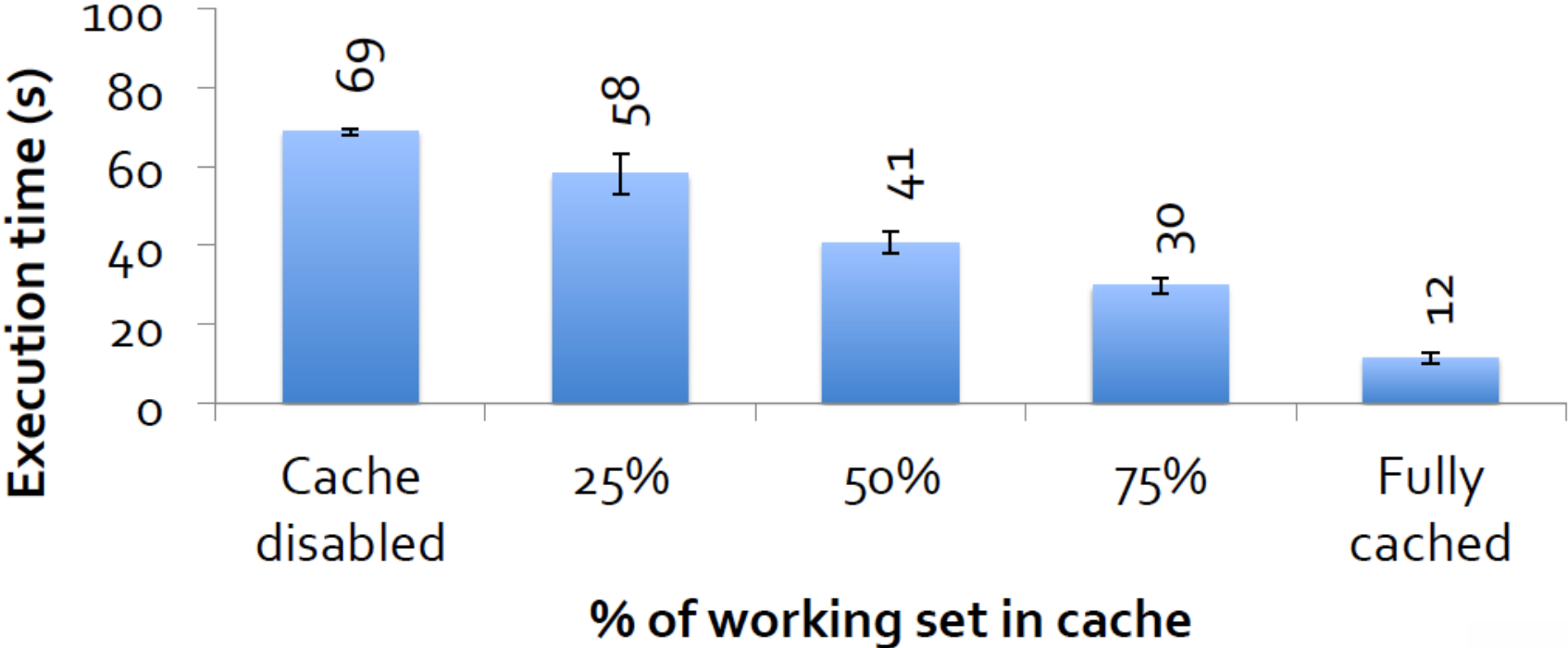
Operations

- Transformations (e.g. map, filter, groupBy)
- Actions (e.g. count, collect, save)

Working With RDDs



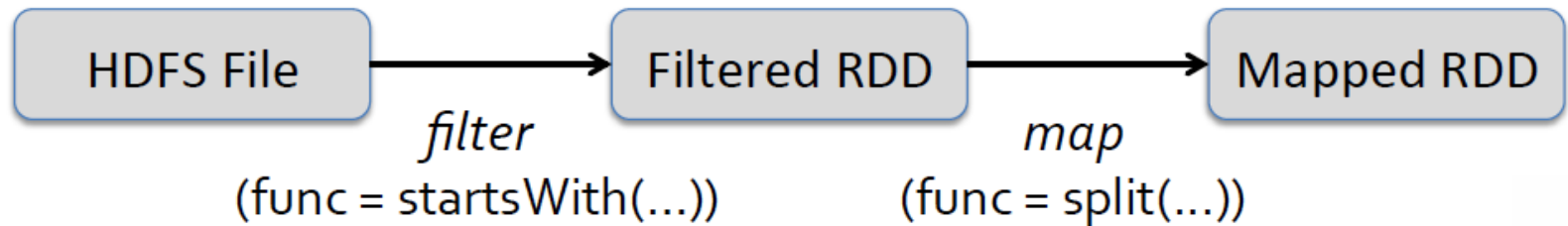
Scaling Down



Fault Recovery

RDDs track lineage information that can be used to efficiently recompute lost data

```
msgs = textFile.filter(lambda s: s.startsWith("ERROR"))  
                .map(lambda s: s.split("\t")[2])
```



Language Support

Standalone Programs

- Python, Scala, Java, R

Interactive Shells

- Python & Scala

Performance

- Java & Scala are faster due to static typing
- ...but Python is often fine

Administrative GUIs

http://<Standalone Master>:8080 (by default)

The image shows two browser windows. The left window is the Spark Master GUI at localhost:8080. The right window is the Spark Stages GUI at localhost:4040/stages/. An orange arrow points from the Spark logo in the right window to the Spark logo in the left window. Another orange arrow points from the application ID 'app-20131202231712-0000' in the left window to the Spark logo in the right window.

Spark Master at spark://mbp-2.local:7077

URL: spark://mbp-2.local:7077
Workers: 3
Cores: 24 Total, 24 Used
Memory: 45.0 GB Total, 1536.0 MB Used
Applications: Running, 0 Completed

Workers

Id
worker-20131202231645-192.168.1.106-56789
worker-20131202231657-192.168.1.106-56801
worker-20131202231705-192.168.1.106-56806

Running Applications

ID	Name
app-20131202231712-0000	Spark shell

Spark shell - Spark Stages

localhost:4040/stages/

Stages Storage Environment Executors

Spark Stages

Total Duration: 3.8 m
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 2
Failed Stages: 0

Active Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read
----------	-------------	-----------	----------	------------------------	--------------

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle
0	count at <console>:13	2013/12/02 21:07:55	83 ms	2/2	754.0 B
1	reduceByKey at <console>:13	2013/12/02 21:07:55	345 ms	2/2	

Failed Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read
----------	-------------	-----------	----------	------------------------	--------------

Spark Essentials: *SparkContext*

First thing that a Spark program does is create a `SparkContext` object, which tells Spark how to access a cluster

In the shell for either Scala or Python, this is the `sc` variable, which is created automatically

Other programs must use a constructor to instantiate a new `SparkContext`

Then in turn `SparkContext` gets used to create other variables

Spark Essentials: *SparkContext*

Scala:

```
scala> sc  
res: spark.SparkContext = spark.SparkContext@470d1f30
```

Python:

```
>>> sc  
<pyspark.context.SparkContext object at 0x7f7570783350>
```

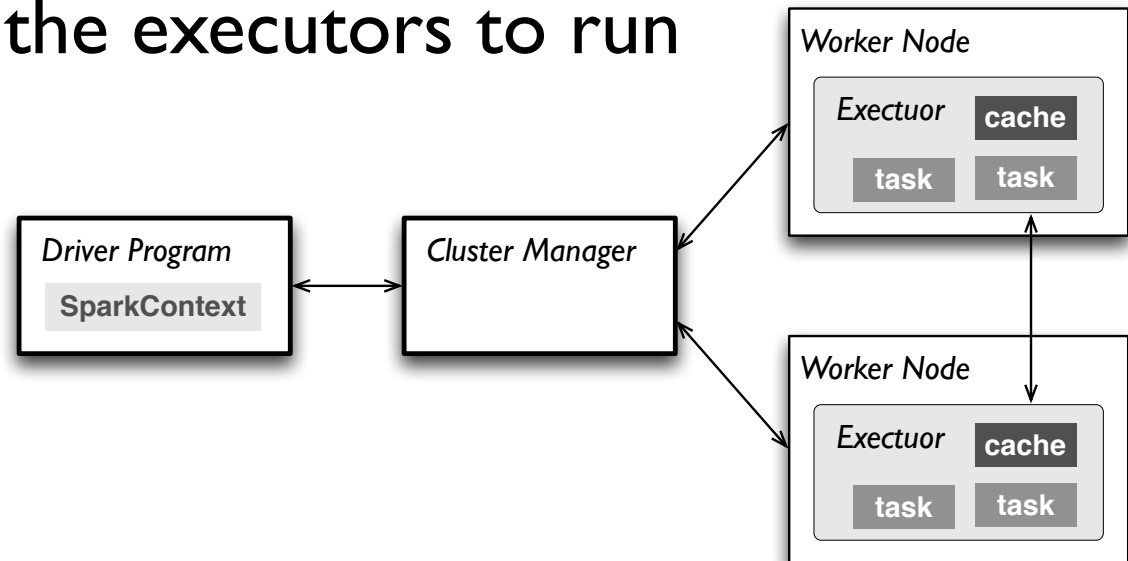
Spark Essentials: *Master*

The `master` parameter for a `SparkContext` determines which cluster to use

<i>master</i>	<i>description</i>
local	run Spark locally with one worker thread (no parallelism)
local[K]	run Spark locally with K worker threads (ideally set to # cores)
spark://HOST:PORT	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
mesos://HOST:PORT	connect to a Mesos cluster; PORT depends on config (5050 by default)

Spark Essentials: Master

1. connects to a *cluster manager* which allocate resources across applications
2. acquires *executors* on cluster nodes – worker processes to run computations and store data
3. sends *app code* to the executors
4. sends *tasks* for the executors to run



Spark Essentials: *RDD*

Resilient **D**istributed **D**atasets (RDD) are the primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel

There are currently two types:

- *parallelized collections* – take an existing Scala collection and run functions on it in parallel
- *Hadoop datasets* – run functions on each record of a file in Hadoop distributed file system or any other storage system supported by Hadoop

Spark Essentials: *RDD*

- two types of operations on RDDs:
transformations and *actions*
- transformations are lazy
(not computed immediately)
- the transformed RDD gets recomputed
when an action is run on it (default)
- however, an RDD can be *persisted* into
storage in memory or disk

Spark Essentials: *RDD*

Scala:

```
scala> val data = Array(1, 2, 3, 4, 5)
data: Array[Int] = Array(1, 2, 3, 4, 5)
```

```
scala> val distData = sc.parallelize(data)
distData: spark.RDD[Int] = spark.ParallelCollection@10d13e3e
```

Python:

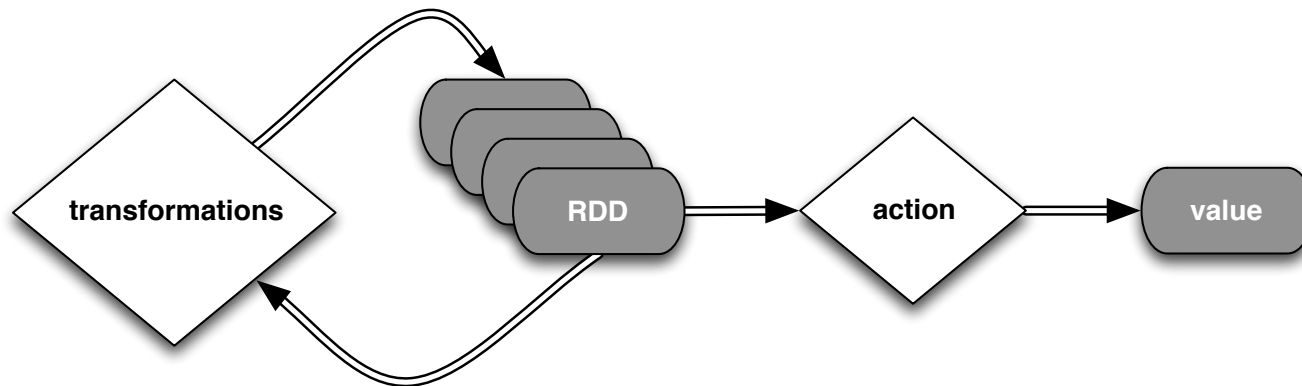
```
>>> data = [1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]
```

```
>>> distData = sc.parallelize(data)
>>> distData
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

Spark Essentials: *RDD*

Spark can create RDDs from any file stored in HDFS or other storage systems supported by Hadoop, e.g., local file system, Amazon S3, Hypertable, HBase, etc.

Spark supports text files, SequenceFiles, and any other Hadoop `InputFormat`, and can also take a directory or a glob (e.g. `/data/201404*`)



Spark Essentials: *RDD*

Scala:

```
scala> val distFile = sc.textFile("README.md")
distFile: spark.RDD[String] = spark.HadoopRDD@1d4cee08
```

Python:

```
>>> distFile = sc.textFile("README.md")
14/04/19 23:42:40 INFO storage.MemoryStore: ensureFreeSpace(36827) called
with curMem=0, maxMem=318111744
14/04/19 23:42:40 INFO storage.MemoryStore: Block broadcast_0 stored as
values to memory (estimated size 36.0 KB, free 303.3 MB)
>>> distFile
MappedRDD[2] at textFile at NativeMethodAccessorImpl.java:-2
```

Spark Essentials: *Transformations*

Transformations create a new dataset from an existing one

All transformations in Spark are *lazy*: they do not compute their results right away – instead they remember the transformations applied to some base dataset

- optimize the required calculations
- recover from lost data partitions

Spark Essentials: *Transformations*

<i>transformation</i>	<i>description</i>
map (<i>func</i>)	return a new distributed dataset formed by passing each element of the source through a function <i>func</i>
filter (<i>func</i>)	return a new dataset formed by selecting those elements of the source on which <i>func</i> returns true
flatMap (<i>func</i>)	similar to map, but each input item can be mapped to 0 or more output items (so <i>func</i> should return a Seq rather than a single item)
sample (<i>withReplacement</i> , <i>fraction</i> , <i>seed</i>)	sample a fraction <i>fraction</i> of the data, with or without replacement, using a given random number generator <i>seed</i>
union (<i>otherDataset</i>)	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct ([<i>numTasks</i>])	return a new dataset that contains the distinct elements of the source dataset

Spark Essentials: *Transformations*

<i>transformation</i>	<i>description</i>
groupByKey ([<i>numTasks</i>])	when called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs
reduceByKey (<i>func</i> , [<i>numTasks</i>])	when called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function
sortByKey ([<i>ascending</i>], [<i>numTasks</i>])	when called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
join (<i>otherDataset</i> , [<i>numTasks</i>])	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key
cogroup (<i>otherDataset</i> , [<i>numTasks</i>])	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples – also called <code>groupWith</code>
cartesian (<i>otherDataset</i>)	when called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)

Spark Essentials: *Transformations*

Scala:

```
val distFile = sc.textFile("README.md")  
distFile.map(l => l.split(" ")).collect()  
distFile.flatMap(l => l.split(" ")).collect()
```

distFile is a collection of lines

Python:

```
distFile = sc.textFile("README.md")  
distFile.map(lambda x: x.split(' ')).collect()  
distFile.flatMap(lambda x: x.split(' ')).collect()
```

Spark Essentials: *Transformations*

Scala:

```
val distFile = sc.textFile("README.md")  
distFile.map(l => l.split(" ")).collect()  
distFile.flatMap(l => l.split(" ")).collect()
```

closures



Python:

```
distFile = sc.textFile("README.md")  
distFile.map(lambda x: x.split(' ')).collect()  
distFile.flatMap(lambda x: x.split(' ')).collect()
```

Spark Essentials: Actions

<i>action</i>	<i>description</i>
reduce (<i>func</i>)	aggregate the elements of the dataset using a function <i>func</i> (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect ()	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
count ()	return the number of elements in the dataset
first ()	return the first element of the dataset – similar to <i>take(1)</i>
take (<i>n</i>)	return an array with the first <i>n</i> elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample (<i>withReplacement</i> , <i>fraction</i> , <i>seed</i>)	return an array with a random sample of <i>num</i> elements of the dataset, with or without replacement, using the given random number generator <i>seed</i>

Spark Essentials: Actions

<i>action</i>	<i>description</i>
saveAsTextFile (<i>path</i>)	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call <code>toString</code> on each element to convert it to a line of text in the file
saveAsSequenceFile (<i>path</i>)	write the elements of the dataset as a Hadoop <code>SequenceFile</code> in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's <code>Writable</code> interface or are implicitly convertible to <code>Writable</code> (Spark includes conversions for basic types like <code>Int</code> , <code>Double</code> , <code>String</code> , etc).
countByKey ()	only available on RDDs of type (K, V) . Returns a <code>Map</code> of (K, Int) pairs with the count of each key
foreach (<i>func</i>)	run a function <i>func</i> on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

Spark Essentials: *Actions*

Scala:

```
val f = sc.textFile("README.md")
val words = f.flatMap(l => l.split(" ")).map(word => (word, 1))
words.reduceByKey(_ + _).collect.foreach(println)
```

Python:

```
from operator import add
f = sc.textFile("README.md")
words = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1))
words.reduceByKey(add).collect()
```

Spark Essentials: *Persistence*

Spark can *persist* (or cache) a dataset in memory across operations

Each node stores in memory any slices of it that it computes and reuses them in other actions on that dataset – often making future actions more than 10x faster

The cache is *fault-tolerant*: if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it

Spark Essentials: Persistence

<i>transformation</i>	<i>description</i>
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc	Same as the levels above, but replicate each partition on two cluster nodes.

Spark Essentials: *Persistence*

Scala:

```
val f = sc.textFile("README.md")
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).collect.foreach(println)
```

Python:

```
from operator import add
f = sc.textFile("README.md")
w = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).cache()
w.reduceByKey(add).collect()
```

Spark Essentials: *Broadcast Variables*

Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks

For example, to give every node a copy of a large input dataset efficiently

Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost

Spark Essentials: *Broadcast Variables*

Scala:

```
val broadcastVar = sc.broadcast(Array(1, 2, 3))  
broadcastVar.value
```

Python:

```
broadcastVar = sc.broadcast(list(range(1, 4)))  
broadcastVar.value
```

Spark Essentials: *Accumulators*

Accumulators are variables that can only be “added” to through an *associative* operation

Used to implement counters and sums, efficiently in parallel

Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types

Only the driver program can read an accumulator’s value, not the tasks

Spark Essentials: Accumulators

Scala:

```
val accum = sc.accumulator(0)  
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
```

accum.value

driver-side

A red rectangular box containing the text 'driver-side' in white, italicized font. Two red arrows originate from the left side of this box. One arrow points diagonally up and to the left towards the Scala code block. The other arrow points diagonally down and to the left towards the Python code block.

Python:

```
accum = sc.accumulator(0)  
rdd = sc.parallelize([1, 2, 3, 4])  
def f(x):  
    global accum  
    accum += x
```

```
rdd.foreach(f)
```

accum.value

Spark Essentials: (K, V) pairs

Spark's “distributed reduce” transformations operate on RDDs of key-value pairs

Scala:

```
val pair = (a, b)

    pair._1 // => a
    pair._2 // => b
```

Python:

```
pair = (a, b)

    pair[0] # => a
    pair[1] # => b
```

Java:

```
Tuple2 pair = new Tuple2(a, b);

    pair._1 // => a
    pair._2 // => b
```

Spark Essentials: (K,V) pairs

`reduceByKey` also automatically implements
combiners on the map side

```
pets = sc.parallelize([("cat", 1), ("dog", 1),  
("cat", 2)])  
  
pets.reduceByKey(lambda x, y: x + y) # => {(cat, 3),  
(dog, 1)}  
  
pets.groupByKey() # => {(cat, [1, 2]), (dog, [1])}  
  
pets.sortByKey() # => {(cat, 1), (cat, 2), (dog, 1)}
```

Spark Essentials: (K,V) pairs

```
visits = sc.parallelize([ ("index.html", "1.2.3.4"),  
                          ("about.html", "3.4.5.6"),  
                          ("index.html", "1.3.3.1") ])
```

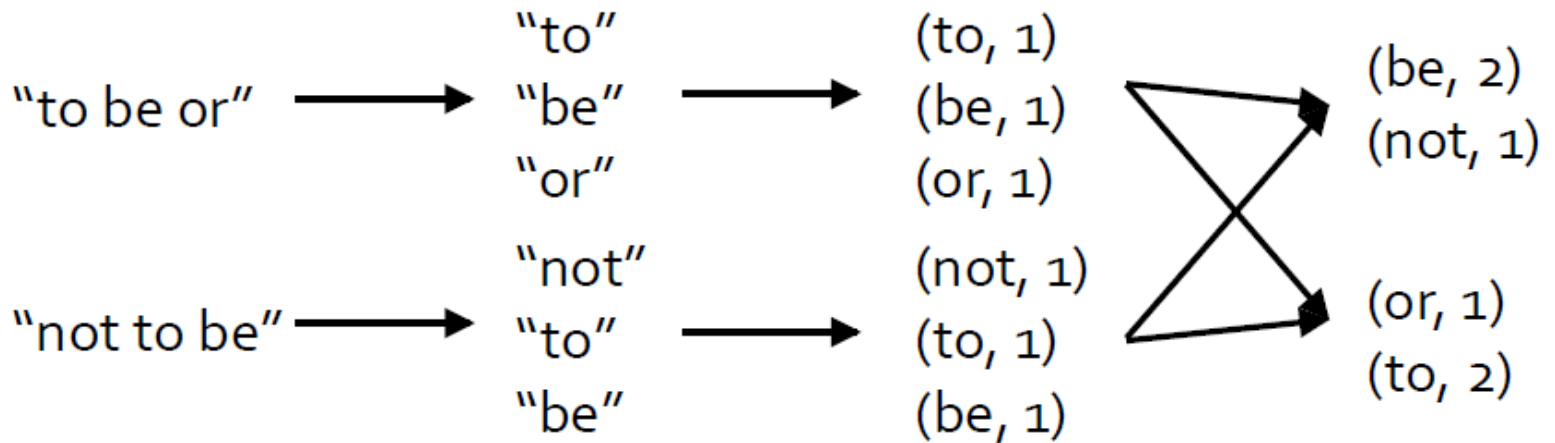
```
pageNames = sc.parallelize([ ("index.html", "Home"),  
                              ("about.html", "About") ])
```

```
visits.join(pageNames)  
# ("index.html", ("1.2.3.4", "Home"))  
# ("index.html", ("1.3.3.1", "Home"))  
# ("about.html", ("3.4.5.6", "About"))
```

```
visits.cogroup(pageNames)  
# ("index.html", ([ "1.2.3.4", "1.3.3.1" ], [ "Home" ]))  
# ("about.html", ([ "3.4.5.6" ], [ "About" ]))
```

Spark Examples: *Word Count*

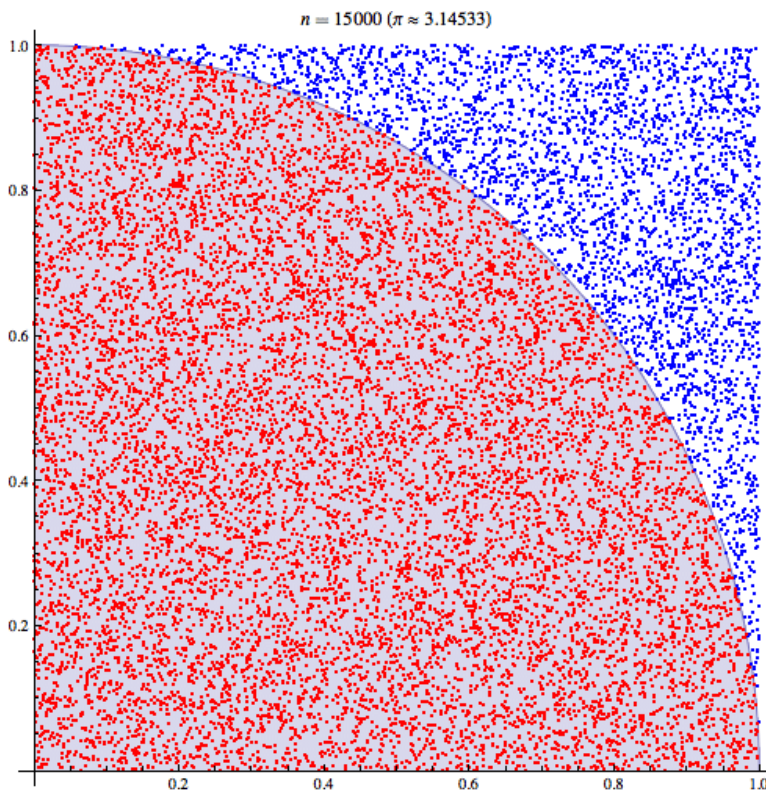
```
lines = sc.textFile("hamlet.txt")
counts = lines.flatMap(lambda line: line.split(" "))
               .map(lambda word => (word, 1))
               .reduceByKey(lambda x, y: x + y)
```



Spark Examples: *Estimate Pi*

Next, try using a **Monte Carlo method** to estimate the value of Pi

```
./bin/run-example SparkPi 2 local
```



wikipedia.org/wiki/Monte_Carlo_method

Spark Examples: Estimate Pi

```
val count = sc.parallelize(1 to n, slices)
```

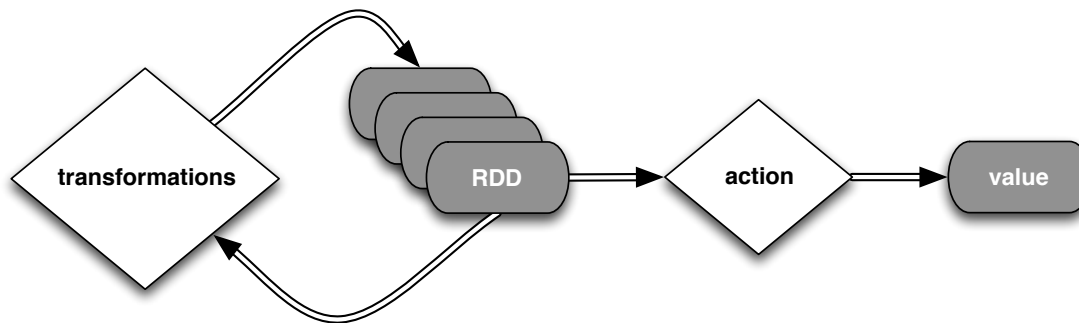
```
.map { i =>  
  val x = random * 2 - 1  
  val y = random * 2 - 1  
  if (x*x + y*y < 1) 1 else 0  
}
```

```
.reduce(_ + _)
```

base RDD

transformed RDD

action



Spark Examples: Estimate Pi

```
import scala.math.random
import org.apache.spark._

/** Computes an approximation to pi */
object SparkPi {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("Spark Pi")
    val spark = new SparkContext(conf)

    val slices = if (args.length > 0) args(0).toInt else 2
    val n = 100000 * slices

    val count = spark.parallelize(1 to n, slices).map { i =>
      val x = random * 2 - 1
      val y = random * 2 - 1
      if (x*x + y*y < 1) 1 else 0
    }.reduce(_ + _)

    println("Pi is roughly " + 4.0 * count / n)
    spark.stop()
  }
}
```