# Spark Essentials

**Spark Essentials:**

Intro apps, showing examples in both Scala and Python…

Let's start with the basic concepts in:

**spark.apache.org/docs/latest/scala-programming-guide.html**

using, respectively:

```
./bin/spark-shell
```

```
./bin/pyspark
```

alternatively, with IPython Notebook:

```
IPYTHON_OPTS="notebook --pylab inline" ./bin/pyspark
```

**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>
```
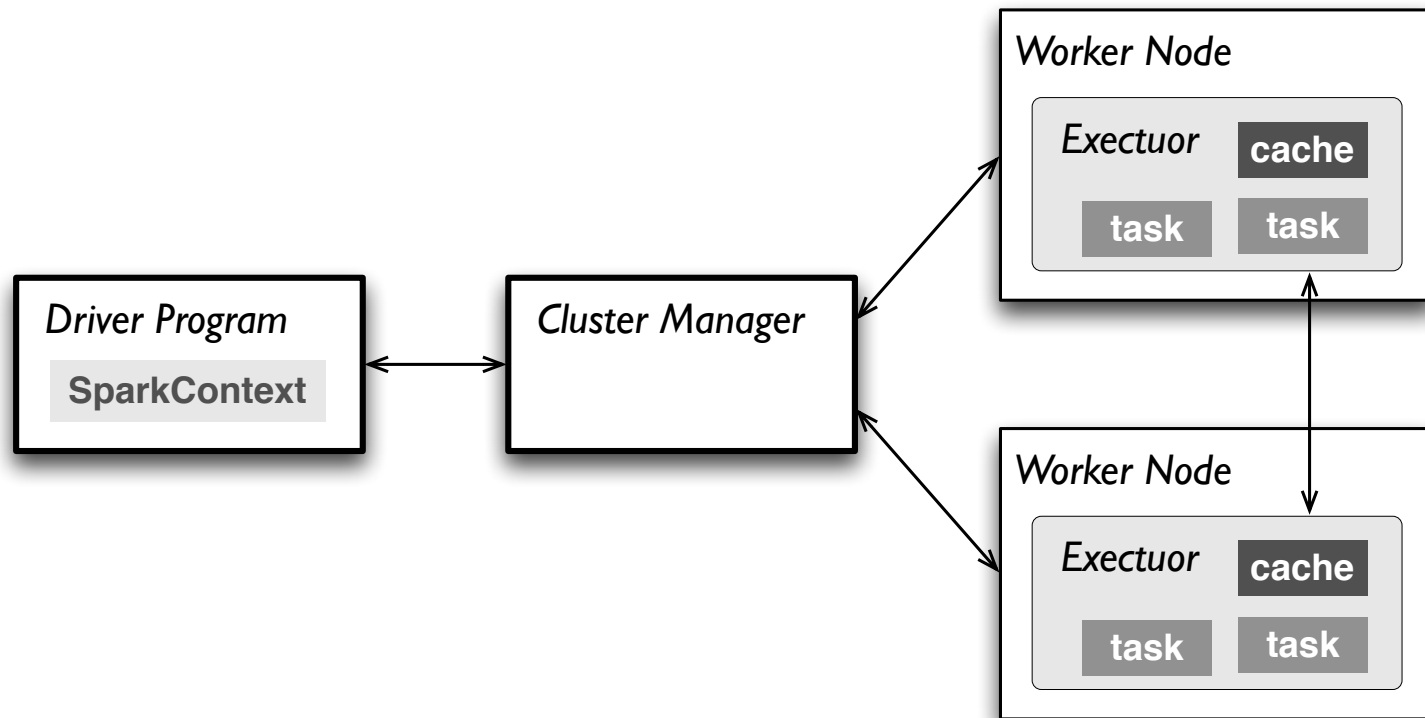
**Spark Essentials:** *Master*

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

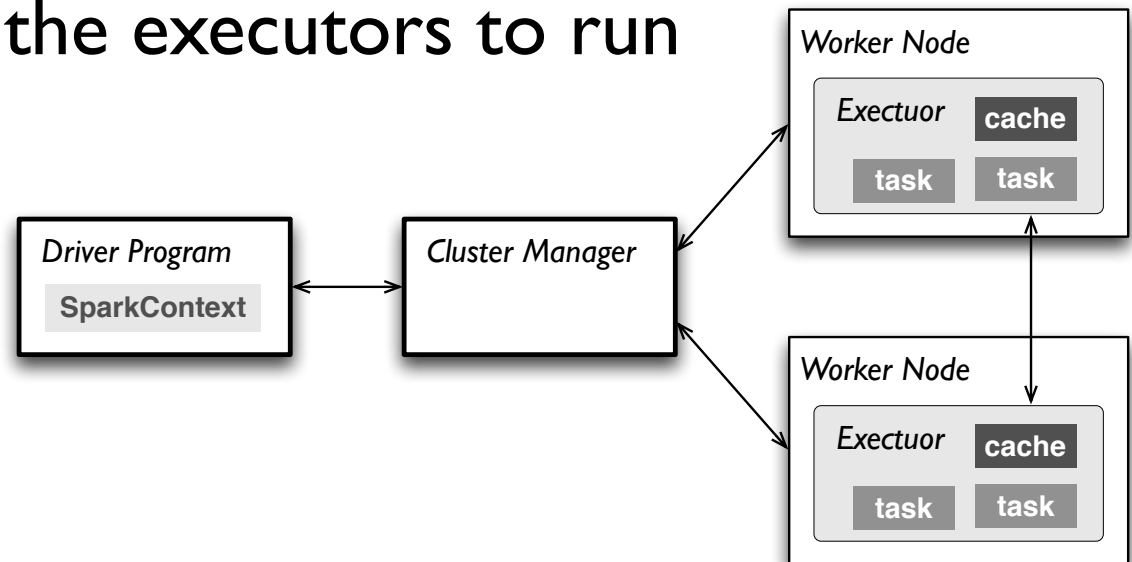| master | description |
| --- | --- |
| **local** | run Spark locally with one worker thread (no parallelism) |
| **local[K]** | run Spark locally with K worker threads (ideally set to # cores) |
| **spark://HOST:PORT** | connect to a Spark standalone cluster; PORT depends on config (7077 by default)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

# Spark Essentials: *Master*

**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

Worker Node

Exectuor    cache

task    task

Driver Program

SparkContext

Cluster Manager

Worker Node

Exectuor    cache

task    task

**R**esilient **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
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
```
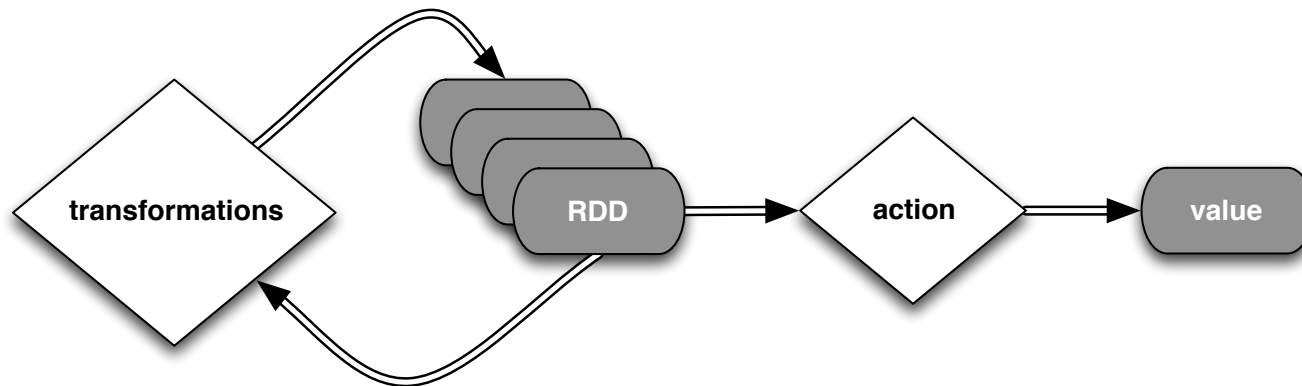
## Python:

```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*

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

# Spark Essentials: *Transformations*

| transformation | description |
|---|---|
| **groupByKey([***numTasks***])** | when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, Seq[V])` pairs |
| **reduceByKey(***func,*** [***numTasks***])** | 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([***ascending***], [***numTasks***])** | 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(***otherDataset,*** [***numTasks***])** | 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(***otherDataset,*** [***numTasks***])** | when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, Seq[V], Seq[W])` tuples – also called `groupWith` |
| **cartesian(***otherDataset***)** | when called on datasets of types `T` and `U`, returns a dataset of `(T, U)` pairs (all pairs of elements) |

## Spark Essentials: *Transformations*

### Scala:

```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:

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

**Spark Essentials:** *Transformations*

## Scala:

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

## Python:

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

*closures*

**Spark Essentials:** *Transformations*

## Scala:

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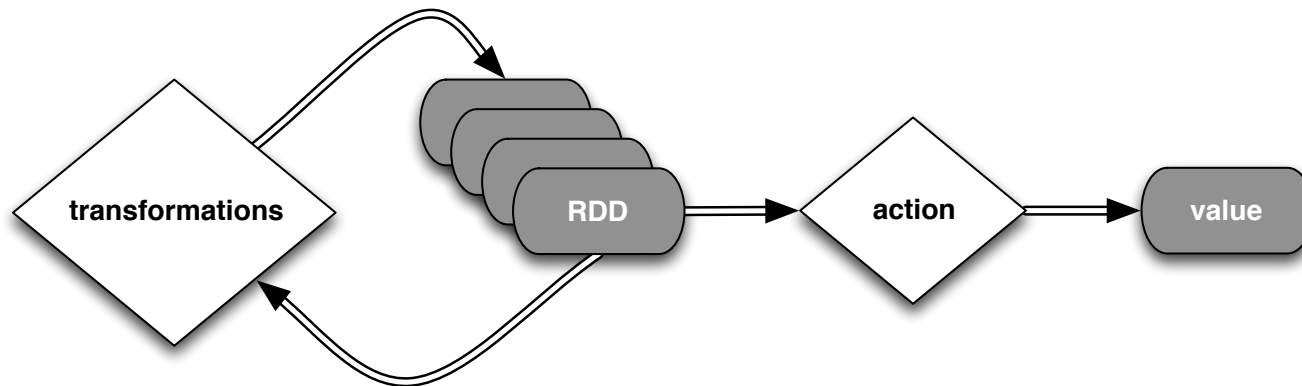
## Python:

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

*closures*

*looking at the output, how would you compare results for map() vs. flatMap() ?*

**Spark Essentials:** *Transformations*

Using closures is now possible in Java 8 with *lambda expressions* support, see the tutorial:

**databricks.com/blog/2014/04/14/Spark-with-Java-8.html**

# Spark Essentials: *Transformations*

## Java 7:

```java
JavaRDD<String> distFile = sc.textFile("README.md");

// Map each line to multiple words
JavaRDD<String> words = distFile.flatMap(
  new FlatMapFunction<String, String>() {
    public Iterable<String> call(String line) {
      return Arrays.asList(line.split(" "));
    }
});
```

## Java 8:

```java
JavaRDD<String> distFile = sc.textFile("README.md");
JavaRDD<String> words =
    distFile.flatMap(line -> Arrays.asList(line.split(" ")));
```

# Spark Essentials: *Actions*

| action | description |
|--------|-------------|
| **reduce(***func***)** | aggregate the elements of the dataset using a function *func* (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 *take(1)* |
| **take(***n***)** | return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements |
| **takeSample(***withReplacement, fraction, seed***)** | return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed |

# Spark Essentials: *Actions*

| action | description |
|---|---|
| **saveAsTextFile(***path***)** | 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 `toString` on each element to convert it to a line of text in the file |
| **saveAsSequenceFile(***path***)** | write the elements of the dataset as a Hadoop `SequenceFile` 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 `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc). |
| **countByKey()** | only available on RDDs of type `(K, V)`. Returns a `Map` of `(K, Int)` pairs with the count of each key |
| **foreach(***func***)** | run a function *func* 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:

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

## Python:

```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*

| transformation | description |
| --- | --- |
| **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:

```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:

```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:

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

## Python:

```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:

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

accum.value
```

Python:

```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: *Accumulators*

## Scala:

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

accum.value
```

driver-side

## Python:

```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*

## Scala:

```scala
val pair = (a, b)

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

## Python:

```python
pair = (a, b)

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

## Java:

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

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

**Spark Essentials:** *API Details*

For more details about the Scala/Java API:

**spark.apache.org/docs/latest/api/scala/
index.html#org.apache.spark.package**


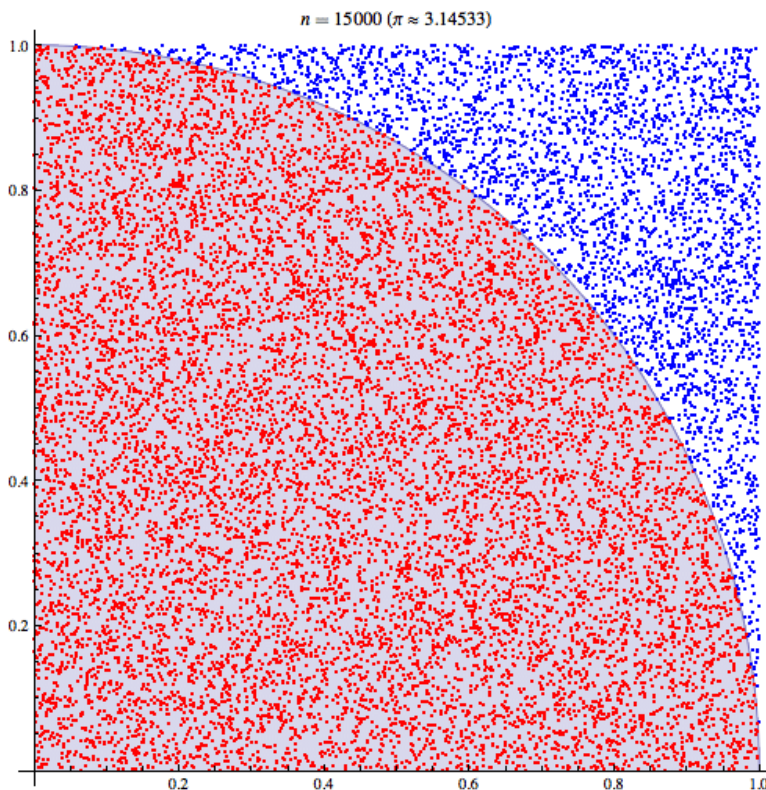For more details about the Python API:

**spark.apache.org/docs/latest/api/python/**

# Spark Examples

**Spark Examples:** *Estimate Pi*

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

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



$n = 15000 \ (\pi \approx 3.14533)$

**wikipedia.org/wiki/Monte_Carlo_method**

# Spark Examples: *Estimate Pi*

```scala
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()
  }
}
```

# Spark Examples: *Estimate Pi*

```scala
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(_ + _)
```
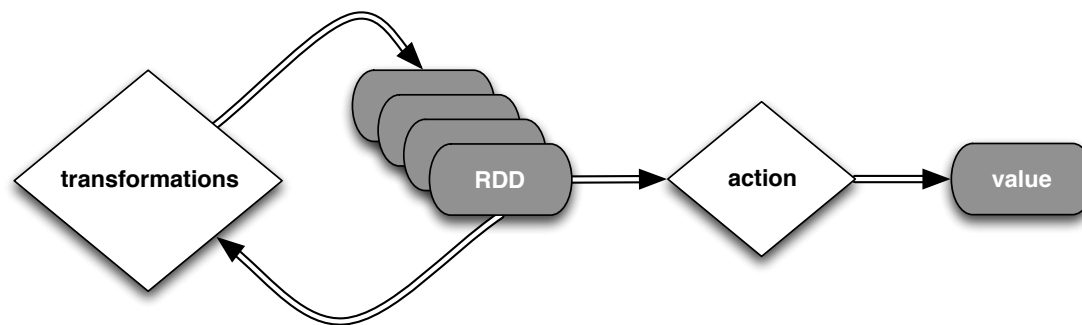
*base RDD*

*transformed RDD*

*action*

**Spark Examples:** *Estimate Pi*

```
val count

 .map
  val
  val
  if
 }

 .reduce
```

base RDD

transformed RDD

# Checkpoint:
# what estimate do you get for Pi?

transformations → RDD → action → value
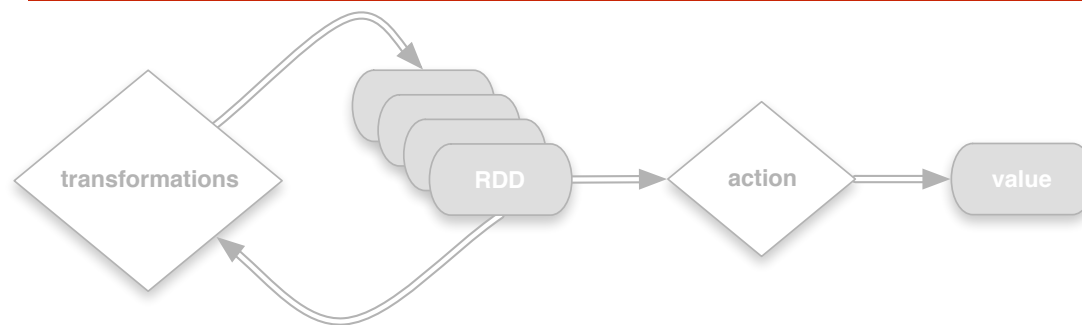
**Spark Examples:** *K-Means*

Next, try using **K-Means** to cluster a set of vector values:

```
cp ../data/examples-data/kmeans_data.txt .
./bin/run-example SparkKMeans kmeans_data.txt 3 0.01 local
```

Based on the data set:

```
0.0 0.0 0.0
0.1 0.1 0.1
0.2 0.2 0.2
9.0 9.0 9.0
9.1 9.1 9.1
9.2 9.2 9.2
```

Please refer to the source code in:

```
examples/src/main/scala/org/apache/spark/examples/SparkKMeans.scala
```

**Spark Examples:** *PageRank*

Next, try using **PageRank** to rank the relationships in a graph:

```
cp ../data/examples-data/pagerank_data.txt .
./bin/run-example SparkPageRank pagerank_data.txt 10 local
```

Based on the data set:

```
1 2
1 3
1 4
2 1
3 1
4 1
```

Please refer to the source code in:

```
examples/src/main/scala/org/apache/spark/examples/SparkPageRank.scala
```