COMP 322: Fundamentals of Parallel Programming

Lecture 30: Distributed Map-Reduce using Hadoop and Spark Frameworks
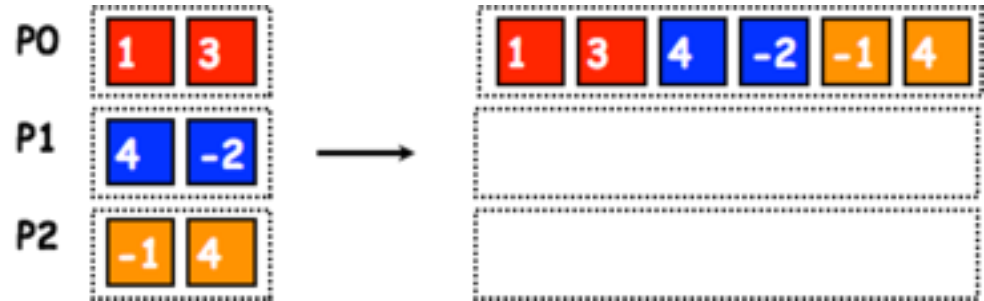
Mack Joyner and Zoran Budimlić
{mjoyner, zoran}@rice.edu

http://comp322.rice.edu

Worksheet #29 solution: MPI Gather

Indicate what value should be provided instead of ??? in line 6 to minimize space, and how it should depend on myrank.


```
1.     MPI.Init(args) ;
2.     int myrank = MPI.COMM_WORLD.Rank() ;
3.     int numProcs = MPI.COMM_WORLD.Size() ;
4.     int size = ...;
5.     int[] sendbuf = new int[size];
6.     int[] recvbuf = new int[???];
7.     . . . // Each process initializes sendbuf
8.     MPI.COMM_WORLD.Gather(sendbuf, 0, size, MPI.INT,
9.                           recvbuf, 0, size, MPI.INT,
10.                          0 /*root*/);
11.    . . .
12.    MPI.Finalize();
```

Solution: myrank == 0 ? (size * numProcs) : 0

# Organization of a Distributed-Memory Multiprocessor (Recap)

Figure (a)
- Host node (P_c) connected to a cluster of processor nodes ($P_0 \ldots P_m$)
- Processors $P_0 \ldots P_m$ communicate via an interconnection network which could be standard TCP/IP (e.g., for Map-Reduce) or specialized for high performance communication (e.g., for scientific computing)

Figure (b)
- Each processor node consists of a processor, memory, and a Network Interface Card (NIC) connected to a router node (R) in the interconnect

In MPI, processes communicate by sending messages to each other. Distributed Map-Reduce offers an alternative approach for programming distributed-memory multiprocessors.
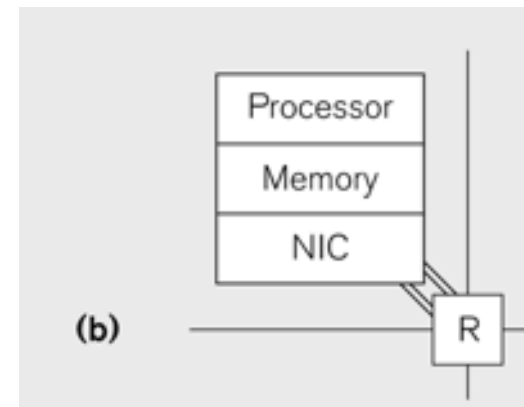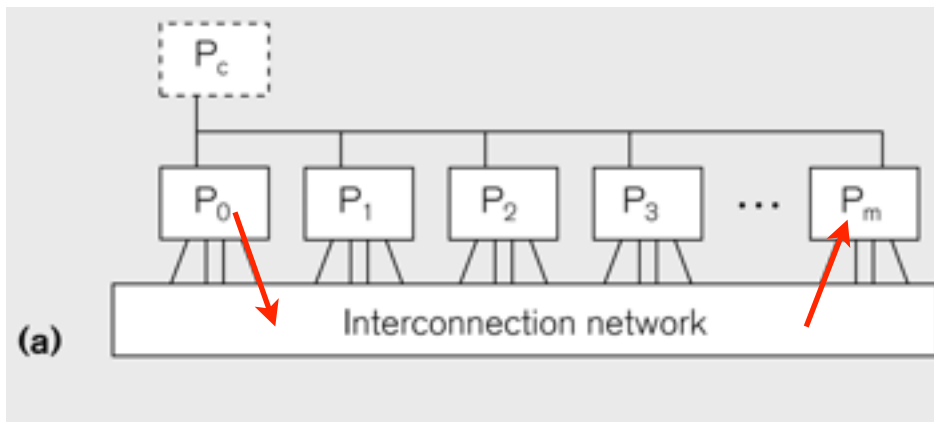
MapReduce Pattern (Recap from Lecture 8)

- Apply `Map` function `f` to user supplied record of key-value pairs

- Compute set of intermediate key/value pairs

- Apply `Reduce` operation `g` to all values that share same key to combine derived data properly
  - Often produces smaller set of values

- User supplies Map and Reduce operations in functional model so that the system can parallelize them, and also re-execute them for fault tolerance

- Distributed Map-Reduce frameworks (Hadoop, Spark) support the Map-Reduce pattern (with extensions) on a distributed-memory multiprocessor
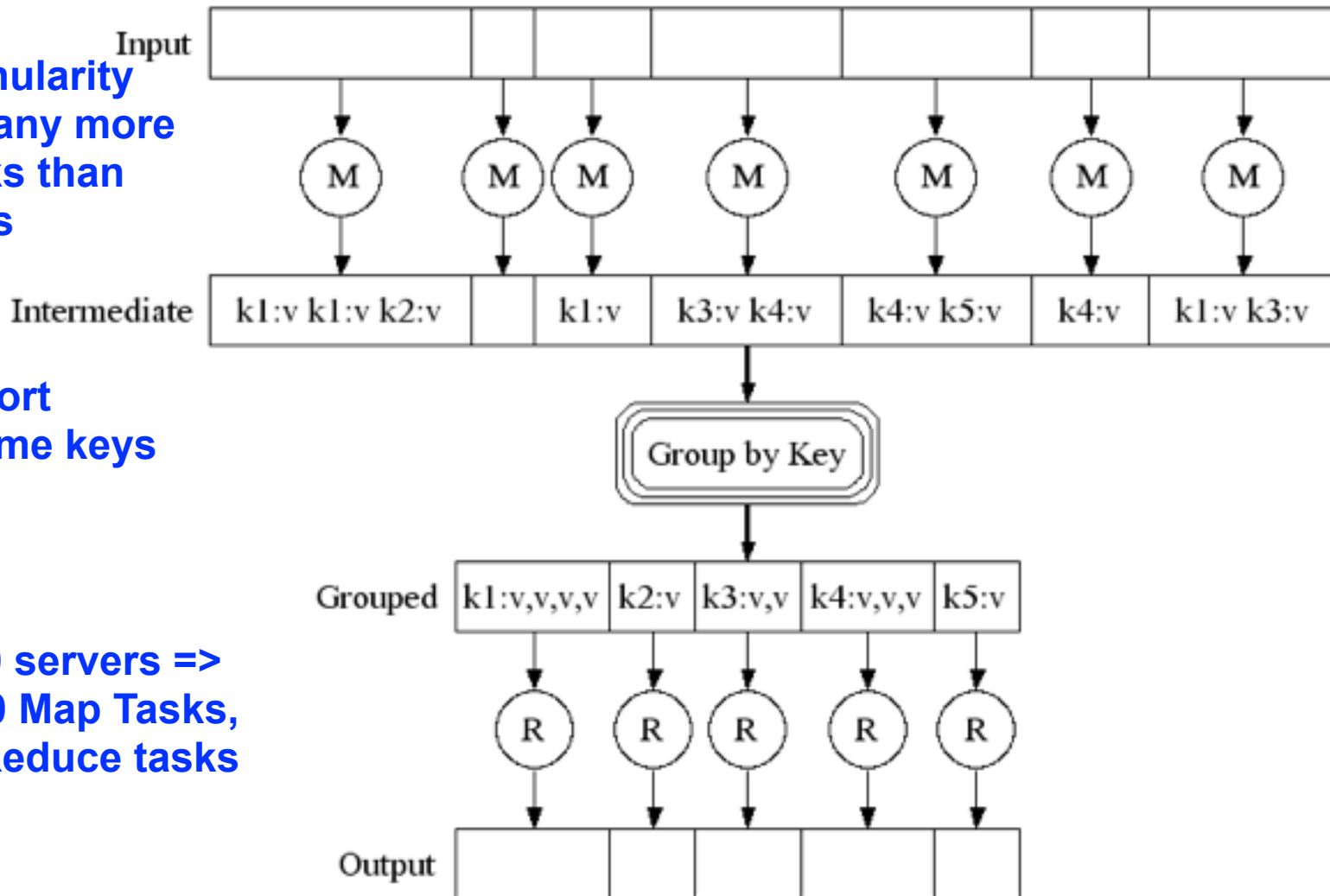
Distributed MapReduce Execution

Fine granularity tasks: many more map tasks than machines

Bucket sort to get same keys together

E.g. 2000 servers =>
**≈ 200,000 Map Tasks,**
≈ 5,000 Reduce tasks

Apache Hadoop Project

- Hadoop: an open-source software framework that supports data-intensive distributed applications, licensed under the Apache v2 license.

- Goals / Requirements:
  - Abstract and facilitate the storage and processing of large and/or rapidly growing data sets
  - Simple programming models
  - High scalability and availability
  - Fault-tolerance
  - Move computation rather than data

- Distributed design, with some centralization
  - Main nodes of cluster are where most of the computational power and storage of the system lies
  - Main nodes run TaskTracker to accept and reply to MapReduce tasks, and also DataNode to store needed blocks closely as possible
  - Central control node runs NameNode to keep track of HDFS directories & files, and JobTracker to dispatch compute tasks to TaskTracker

- Written in Java, also supports Python and Ruby

- Acknowledgment: slides on Hadoop from UCI CS 237 course by Nalini Venkatasubramanian

Hadoop's Architecture
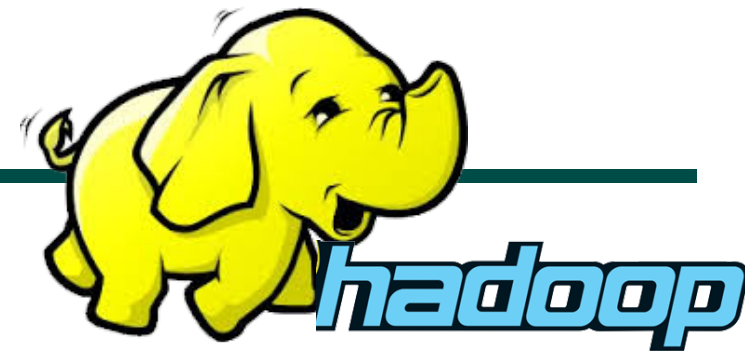
- NN = Name Node

- DN = Data Node

- TT = Task Tracker

- Acknowledgment: <u>slides on Hadoop</u> from UCI CS 237 course by Nalini Venkatasubramanian
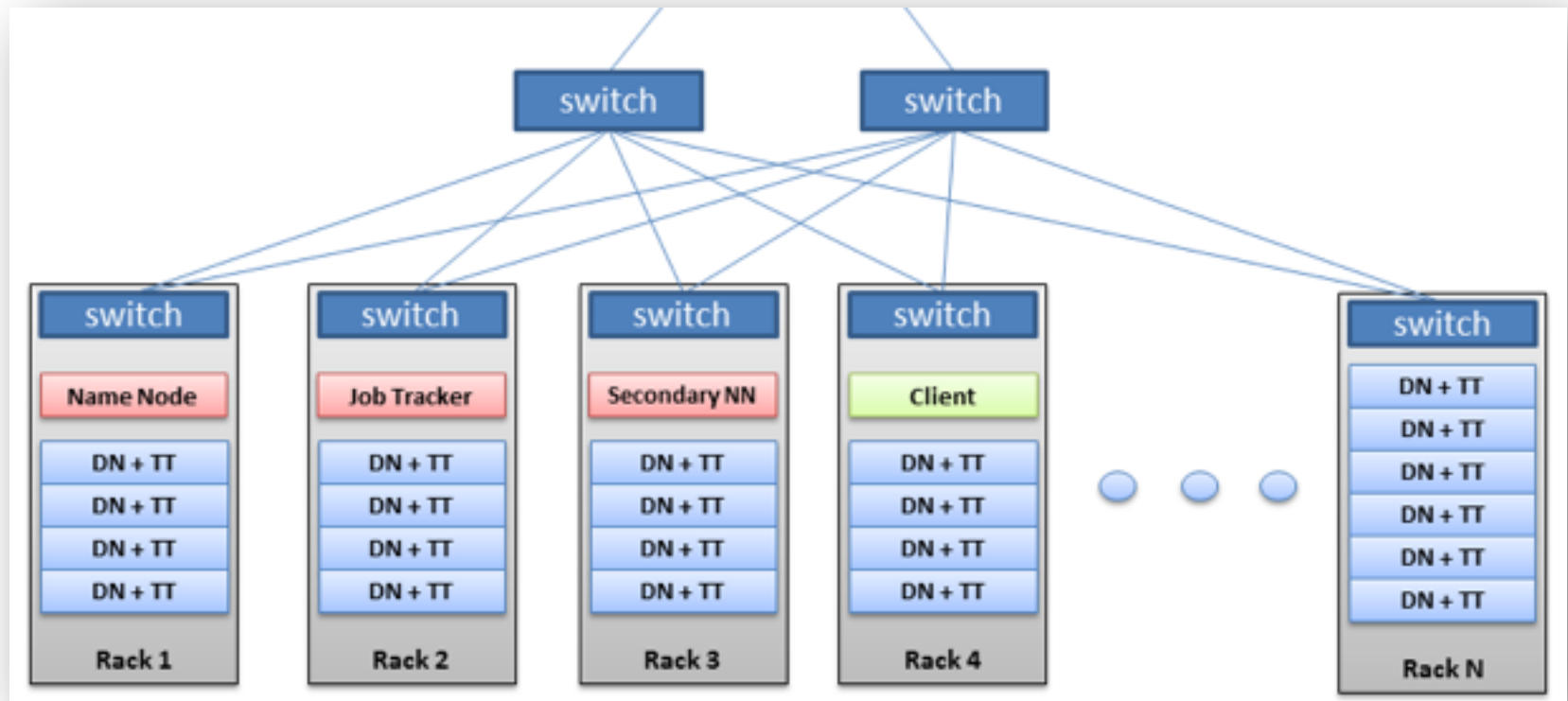
Spark and Iterative Map/Reduce

- **Apache Spark: General purpose functional programming over a cluster**

  — **Caches results of map/reduce operations in memory so they can be used on subsequent iterations**

  — **Tends to be 10-100 times faster than Hadoop for many applications**

Apache Spark Project
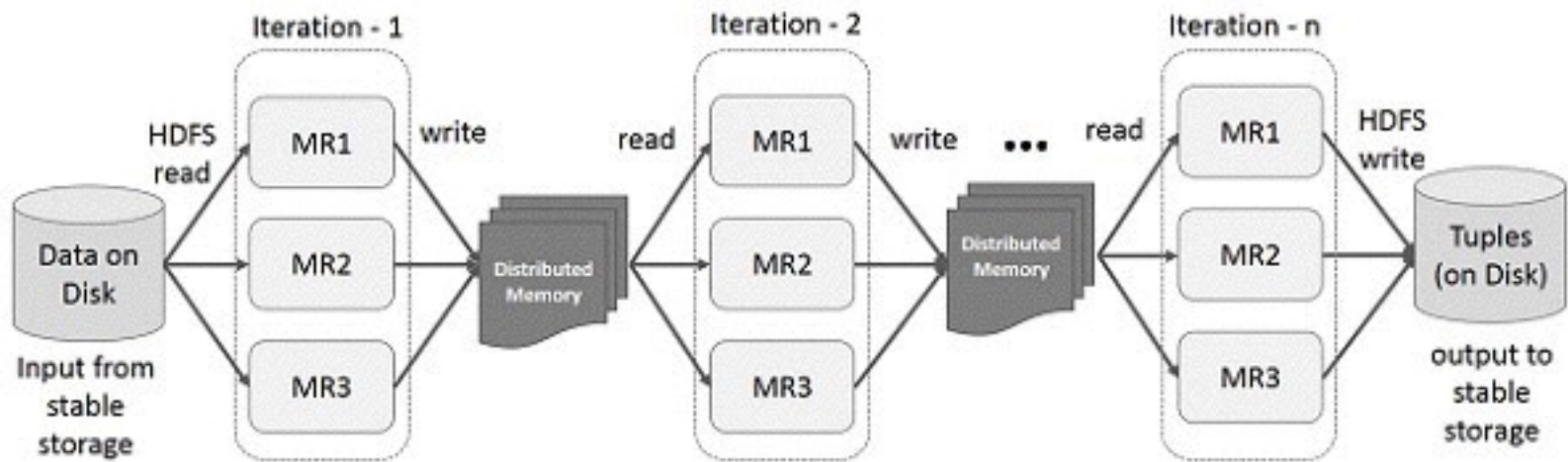
- **Spark is a data parallel processing framework, which means it will execute tasks as close to where the data lives as possible (i.e. minimize data transfer).**

- **Spark follows a paradigm of keeping as much data in-memory and spilling excess to disk rather than pulling data from disk when needed.**

- **Spark decomposes your program into tasks and handles dispatching and scheduling of these tasks on worker nodes in your cluster.**

- **Spark revolves around the concept of a resilient distributed datasets (RDD).**
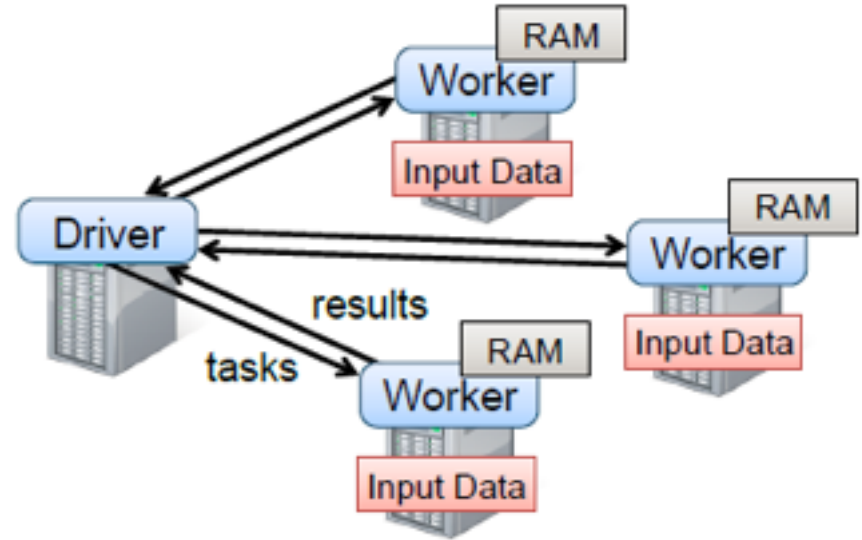
Resilient Distributed Datasets

- **The key construct in Spark is the Resilient Distributed Dataset (RDD)**

  — **RDDs can be though of as a collection of key-value pairs**

- **An RDD is a giant <u>immutable</u> collection, distributed in a redundant way over all the machines in a cluster**

- **The types of the elements in the RDD can be arbitrary elements**

- **If the elements are pairs, then the RDD acts like a table**

- **Computations on an RDD (including Map/Reduce) can be expressed as functional programming operations**
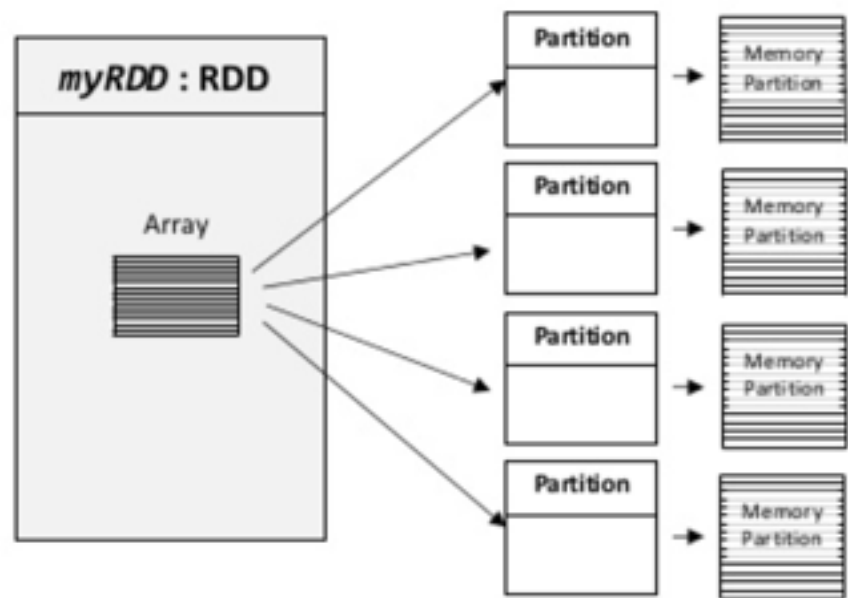
Copyright 2014 Tony Duarte

Working with RDDs

- There are two kinds of operations one can perform over an RDD.

- <u>Transformations</u>: Operations like `map`, `filter`, `join` etc. that just return another RDD. They are *lazy* operations.

- <u>Actions</u>: These are operations that *actually produce* results like `count`, `collect`, `save` etc. These operations don't return an RDD.

- Similar to *intermediate* and *terminal* operations in Java 8 Streams.


```
textFile = sc.textFile("SomeFile.txt")
```

Transformations

```
map(func)
flatMap(func)
filter(func)
groupByKey()
reduceByKey(func)
mapValues(func)
...
```

RDD → **Action** → **Value**

```
take(N)
count()
collect()
reduce(func)
takeOrdered(N)
top(N)
...
```
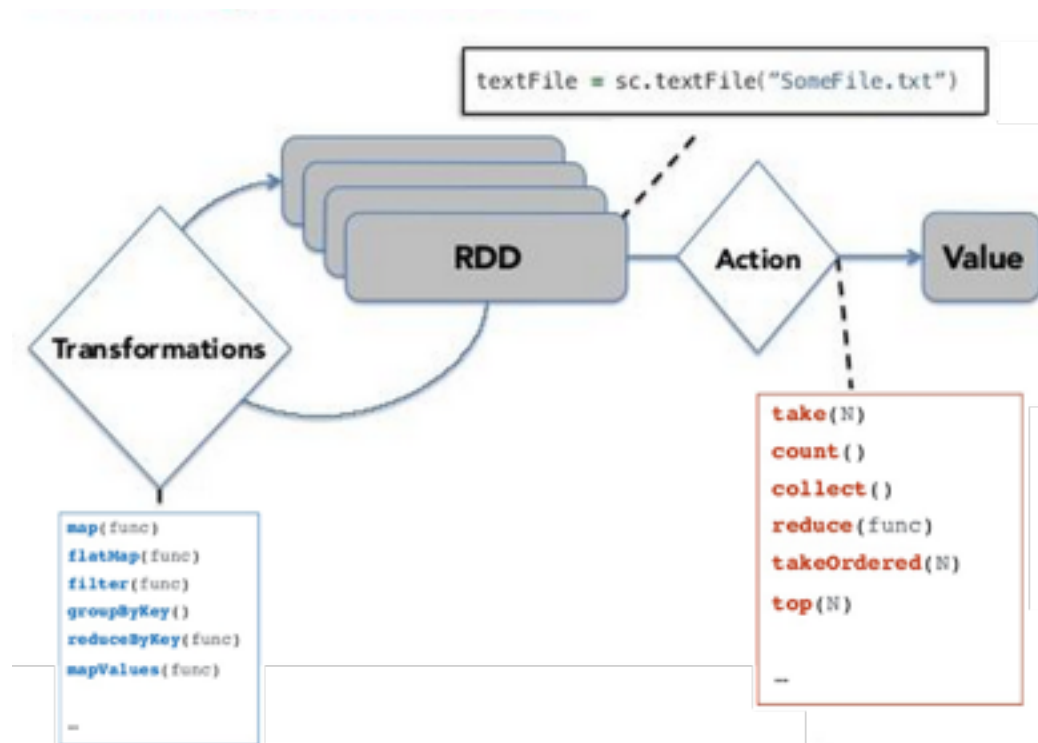
Advantages of Immutability

- **The distributed nature of RDDs is not evident in the programming model.**

- **RDD elements can be replicated for fault tolerance.**

- **Purely functional operations can be easily defined on RDDs**

- **Because RDDs are immutable, all the operations from purely functional programming can be applied and parallelized in a straightforward way.**

- **The runtime has great flexibility in scheduling operations on RDDs and executing them in parallel on partitions.**

- **Partitions of RDDs can be recomputed from their lineage.**

Word Count in Apache Spark

```
JavaRDD<String> file = context.textFile(inputFile);

JavaPairRDD<String, Integer> counter =
  file.flatMap(s -> Arrays.asList(s.split(" ")))
      .mapToPair(s -> new Tuple2<>(s, 1))
      .reduceByKey((a, b) -> a + b);

counter.collect().forEach(System.out::println);

// Definition of "flatMap"
// x.flatMap(f) = x.map(f).flatten()

// Definition of "reduceByKey"
// x.reduceByKey(f) = x.groupByKey()
                       .map(xs -> xs.reduce(f))
```

Word Count in Apache Spark

```
["this is a line",
 "this is another line",
 "this is yet another line"]
.map(s -> Arrays.asList(s.split(" ")))
.flatten()
-->
[["this", "is", "a", "line"],
 ["this", "is", "another", "line"],
 ["this", "is", "yet", "another", "line"]]
.flatten()
-->
["this", "is", "a", "line", "this", "is",
 "another", "line", "this", "is", "yet",
 "another", "line"]
```

Word Count in Apache Spark

```
["this", "is", "a", "line", "this", "is",
 "another", "line", "this", "is", "yet",
 "another", "line"]
.map(s -> new Tuple2<>(s, 1))


—>


[["this",1], ["is",1], ["a",1], ["line",1],
 ["this",1], ["is",1], ["another",1],
 ["line",1], ["this",1], ["is",1],
 ["yet",1], ["another",1], ["line",1]]
```

Word Count in Apache Spark

```
[["this",1], ["is",1], ["a",1], ["line",1],
 ["this",1], ["is",1], ["another",1],
 ["line",1], ["this",1], ["is",1],
 ["yet",1],["another",1], ["line",1]]
.groupByKey().map(xs -> xs.reduce(
                          (a,b) -> a + b))

—>
[["this", [1,1,1]],
 ["is", [1,1,1]],
 ["a", [1]],
 ["line", [1,1,1]],
 ["another", [1,1]],
 ["yet", [1]]].map(xs -> xs.reduce((a,b) -> a + b))
```

Word Count in Apache Spark

```
[["this", [1,1,1]],
 ["is", [1,1,1]],
 ["a", [1]],
 ["line", [1,1,1]],
 ["another", [1,1]],
 ["yet", [1]]].map(xs -> xs.reduce(
                         (a,b) -> a + b)
—>
[["this", [1,1,1]].reduce((a,b) -> a + b),
 ["is", [1,1,1]].reduce((a,b) -> a + b),
 ["a", [1]].reduce((a,b) -> a + b),
 ["line", [1,1,1]].reduce((a,b) -> a + b),
 ["another", [1,1]].reduce((a,b) -> a + b),
 ["yet", [1]].reduce((a,b) -> a + b))]
```

Word Count in Apache Spark

```
[["this", [1,1,1]].reduce((a,b) -> a + b),
 ["is", [1,1,1]].reduce((a,b) -> a + b),
 ["a", [1]].reduce((a,b) -> a + b),
 ["line", [1,1,1]].reduce((a,b) -> a + b),
 ["another", [1,1]].reduce((a,b) -> a + b),
 ["yet", [1]].reduce((a,b) -> a + b))]


—>


[["this", 3], ["is", 3], ["a", 1],
 ["line", 3], ["another", 2], ["yet", 1]]
```