



*Marilyn Waldman*

*@mdwaldman*

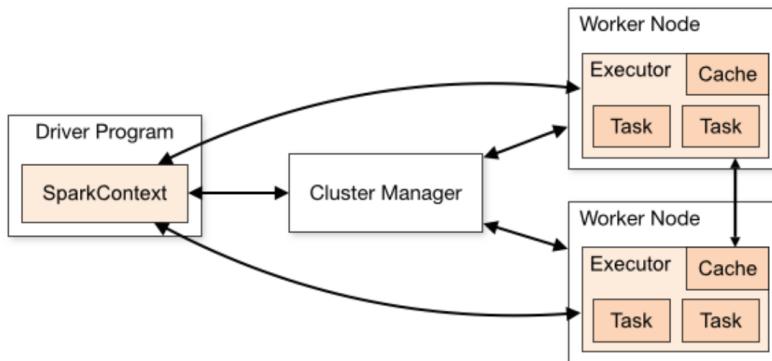
# *Agenda*

- **What is Spark - Why Spark**
- **lab 1 - Functional programming -map and reduce**
- **lab 2 - RDD's**
- **lab 3 - Word Count**
- **lab 4 - Spark SQL**
- **Conclusions**

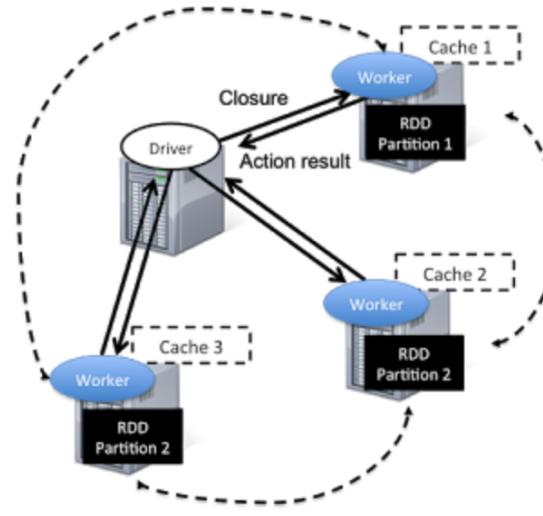




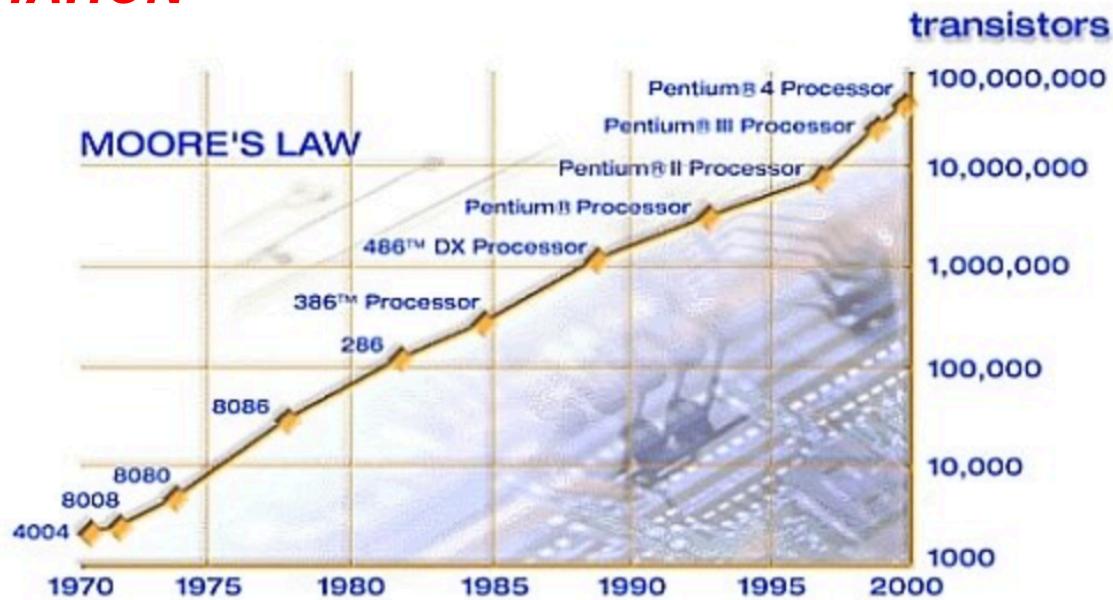
Cluster computing platform, a *distributed system*



- genomics
- regression - optimization
- real-time data
- anomaly detection
- fraud detection
- codeneuro



## **COMPUTATION**

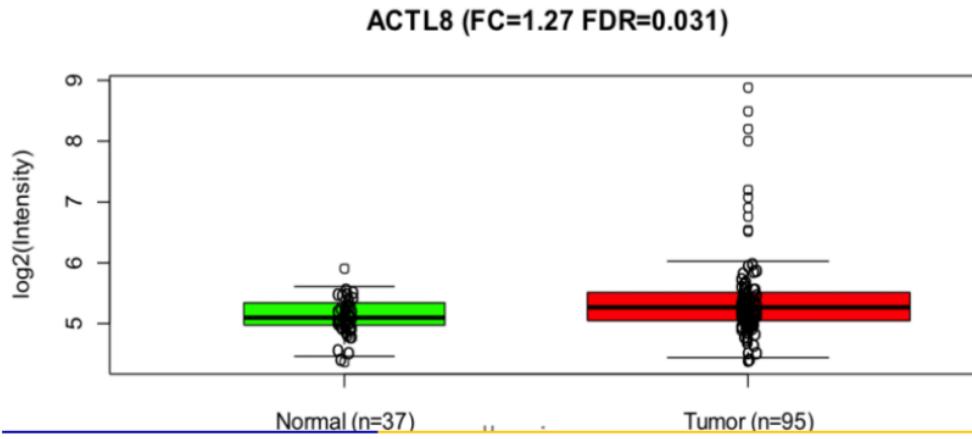


*Moore's Law* states that the number of transistors that can be placed on an integrated circuit for the same price will increase exponentially by a factor of 2 every 18 to 24 months. In other words, put simply *Moore's Law* claims that CPU processing power will double approximately every two years for the price of 1,000 dollars. (Graph copyright Ray Kurzweil)

# DATA

At its core, COPA is a technique for analyzing key-value pairs of gene expression data and to detect outliers, which are the candidates for cancer. A parallelized algorithm was necessary owing to the size of the data involved, says Parsian, who also teaches at the University of Santa Clara.

- COPA = Cancer Outlier Profile Analysis
- Statistics designed to identify outliers in cancer gene expression profile



*COPA has proven its value in detecting genetic mutations linked to prostate cancer, and is now being studies with other types of cancers*

"Of course, if the data size is small, you can detect it visually or by writing some sample programs," Parsian says. "But when you have terabytes of data you're analyzing, detecting mutation is impossible to do visually."

There are many ways to solve this problem, and any number of algorithms could do the

# Dijkstra - Cooperating Sequential Processes



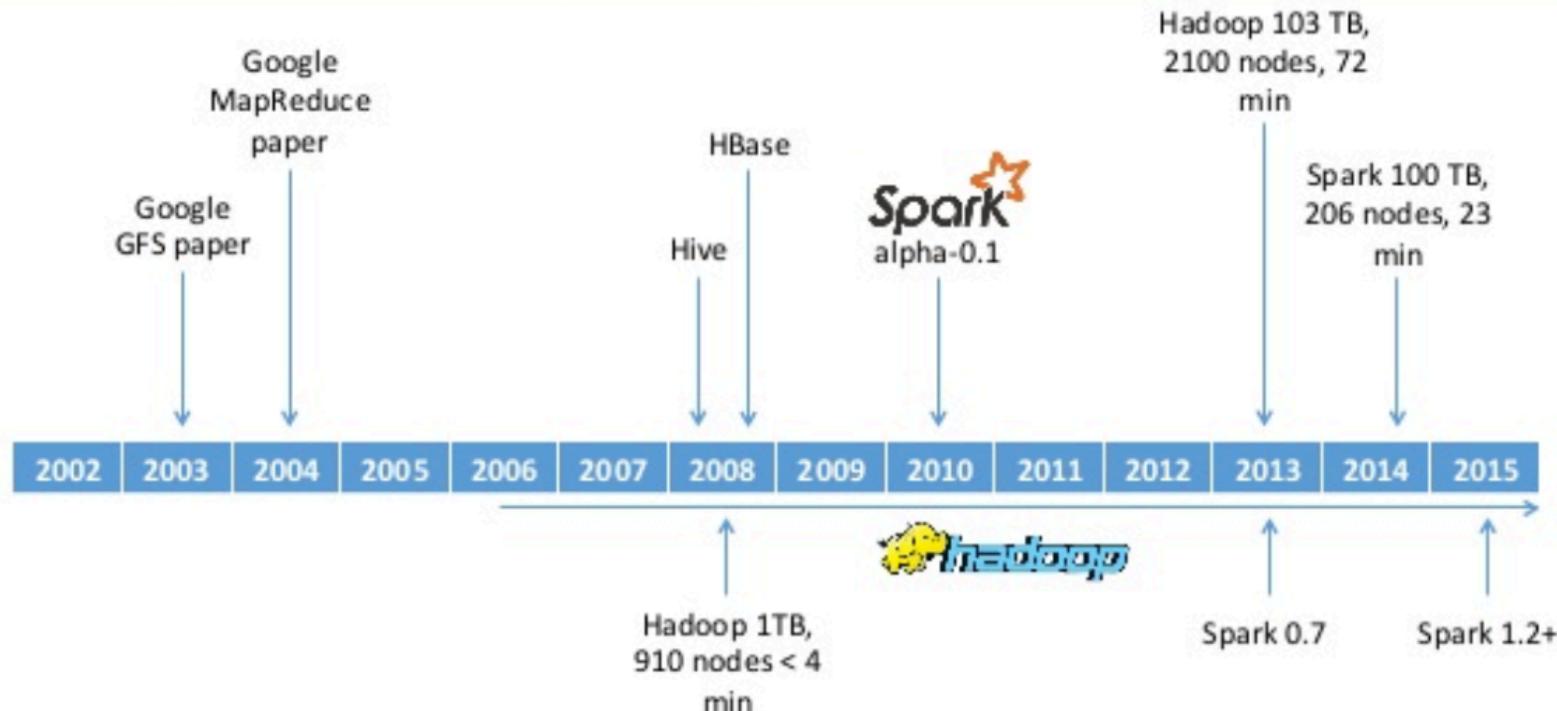
*communicate*

*cooperate*

*synchronize*

***Caution: Distributed Processing is a hard problem. Keeping track of all the moving parts is challenging.***

# Timeline



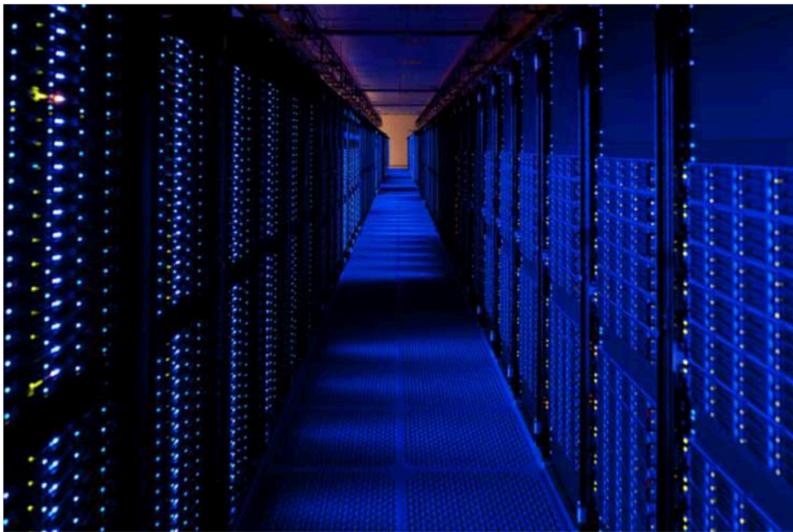
#t3chfest2015

STRATIO

9

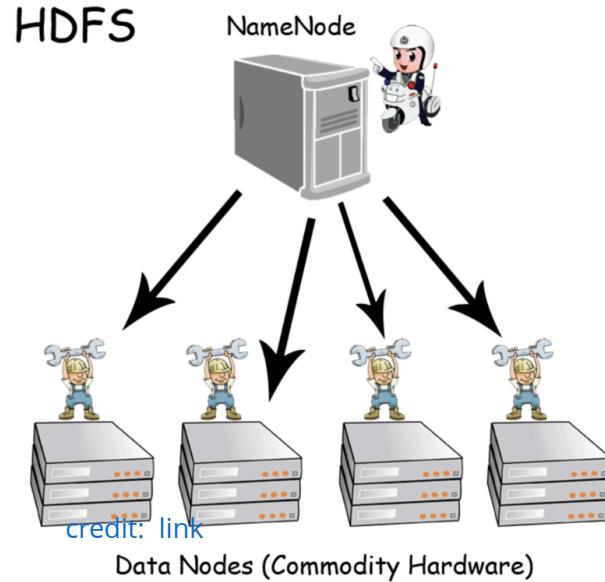
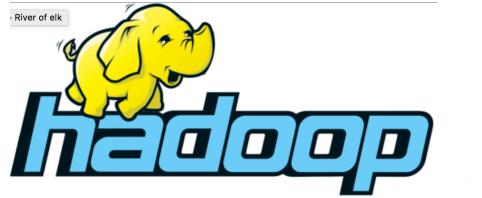
credit : Adios hadoop, Hola Spark! T3chfest 2015

9 of 26



The project includes these modules:

- **Hadoop Common:** The common utilities that support the other Hadoop modules.
- **Hadoop Distributed File System (HDFS™):** A distributed file system that provides high-throughput access to application data.
- **Hadoop YARN:** A framework for job scheduling and cluster resource management.
- **Hadoop MapReduce:** A YARN-based system for parallel processing of large data sets.



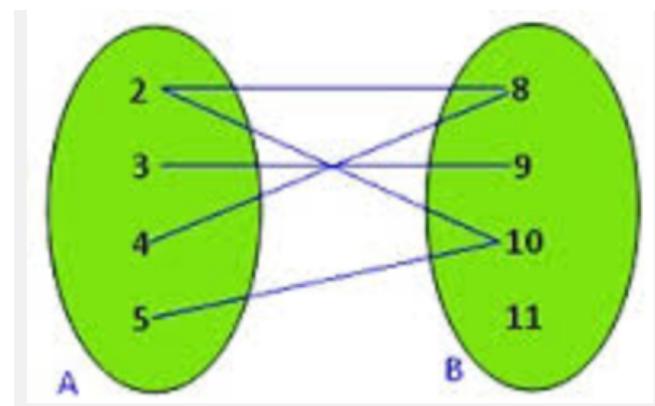


- Began at UC Berkeley in 2009
- Fast and general purpose cluster computing
- 10x faster on disk. 100x faster in-memory
- Integrates with Hadoop and can read existing data
- API's - Java, Python, Scala
- Deeply embraced due to *elegance* of use

# Functional Programming

vs

# Imperative Programming



## *Functional*



```
range = domain.map(lambda x : x*x)
```

*domain is immutable*

*range is a new fresh object*

*required for parallel processing*

## *Imperative*

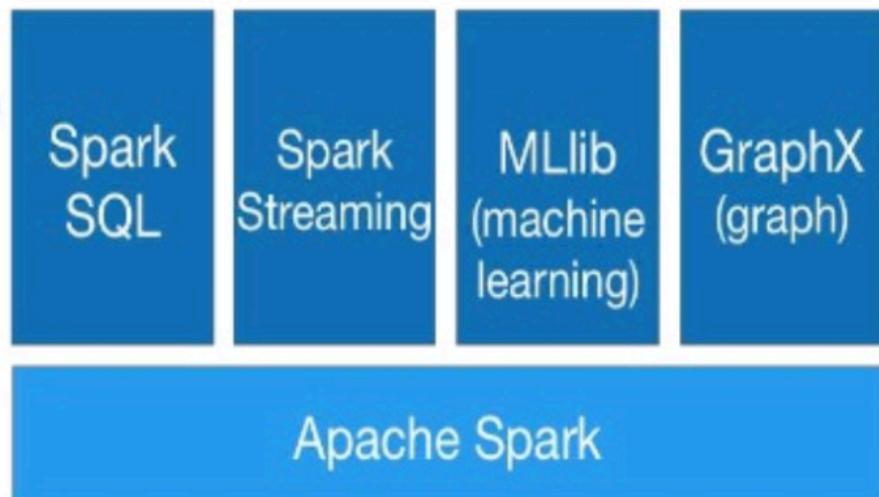
```
for i in domain:
```

```
    domain[i] = domain[i] * domain[i]
```

*danger: side effects, mutations*

# Spark Stack

- Spark SQL
  - For SQL and unstructured data processing
- MLlib
  - Machine Learning Algorithms
- GraphX
  - Graph Processing
- Spark Streaming
  - stream processing of live data streams



<http://spark.apache.org>

1. Most machine learning programs are iterative. Each iteration improves results
2. With MapReduce each iteration is written to disk. This is expensive.
3. Spark runs ***in memory*** using an abstraction known as an **RDD**

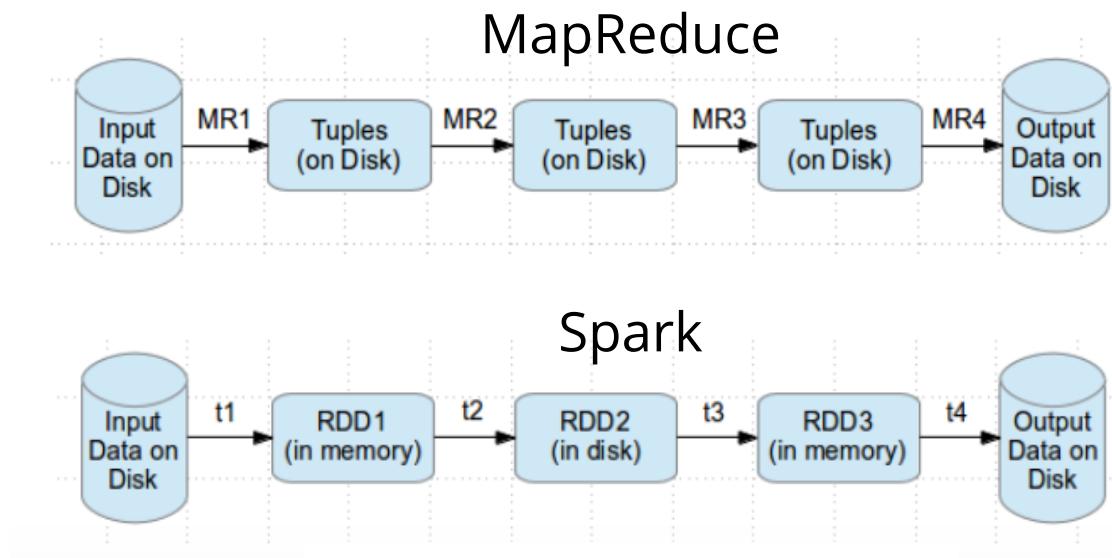
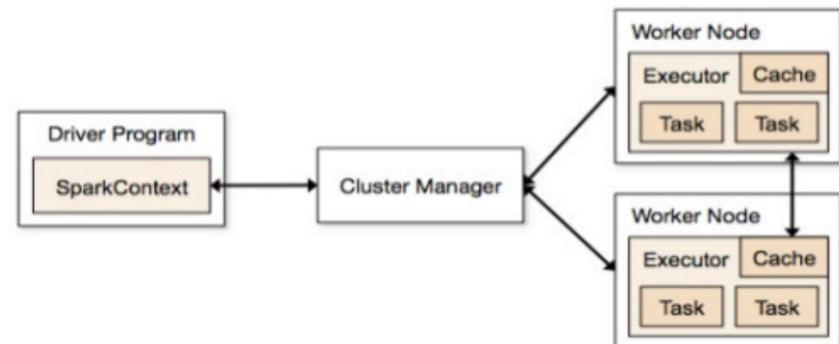
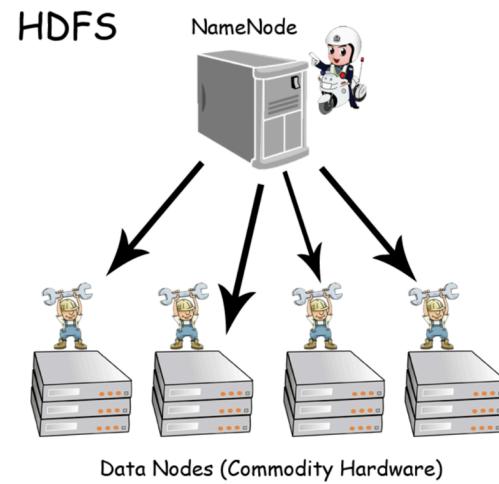


Image Credits: [Datatamasha.com](http://Datatamasha.com)

# ***Take the compute to the data***

## Execution Flow



<http://spark.apache.org/docs/latest/cluster-overview.html>

## Coding Exercise: WordCount

```
1 public class WordCount {
2     public static class TokenizerMapper
3         extends Mapper<Object, Text, Text, IntWritable>{
4
5     private final static IntWritable one = new IntWritable(1);
6     private Text word = new Text();
7
8     public void map(Object key, Text value, Context context
9                     ) throws IOException, InterruptedException {
10        StringTokenizer itr = new StringTokenizer(value.toString());
11        while (itr.hasMoreTokens()) {
12            word.set(itr.nextToken());
13            context.write(word, one);
14        }
15    }
16}
17
18 public static class IntSumReducer
19     extends Reducer<Text,IntWritable,Text,IntWritable> {
20     private IntWritable result = new IntWritable();
21
22     public void reduce(Text key, Iterable<IntWritable> values,
23                        Context context
24                        ) throws IOException, InterruptedException {
25
26         int sum = 0;
27         for (IntWritable val : values) {
28             sum += val.get();
29         }
30         result.set(sum);
31         context.write(key, result);
32     }
33}
34
35 public static void main(String[] args) throws Exception {
36     Configuration conf = new Configuration();
37     String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
38     if (otherArgs.length < 2) {
39         System.err.println("Usage: wordcount <in> [<in>... <out>]");
40         System.exit(2);
41     }
42     Job job = new Job(conf, "word count");
43     job.setJarByClass(WordCount.class);
44     job.setMapperClass(TokenizerMapper.class);
45     job.setCombinerClass(IntSumReducer.class);
46     job.setReducerClass(IntSumReducer.class);
47     job.setOutputKeyClass(Text.class);
48     job.setOutputValueClass(IntWritable.class);
49     for (int i = 0; i < otherArgs.length - 1; i++) {
50         FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
51     }
52     FileOutputFormat.setOutputPath(job,
53         new Path(otherArgs[otherArgs.length - 1]));
54     System.exit(job.waitForCompletion(true) ? 0 : 1);
55 }
```

```
1 val f = sc.textFile(inputPath)
2 val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
3 w.reduceByKey(_ + _).saveAsText(outputPath)
```

## WordCount in 3 lines of Spark

## WordCount in 50+ lines of Java MR

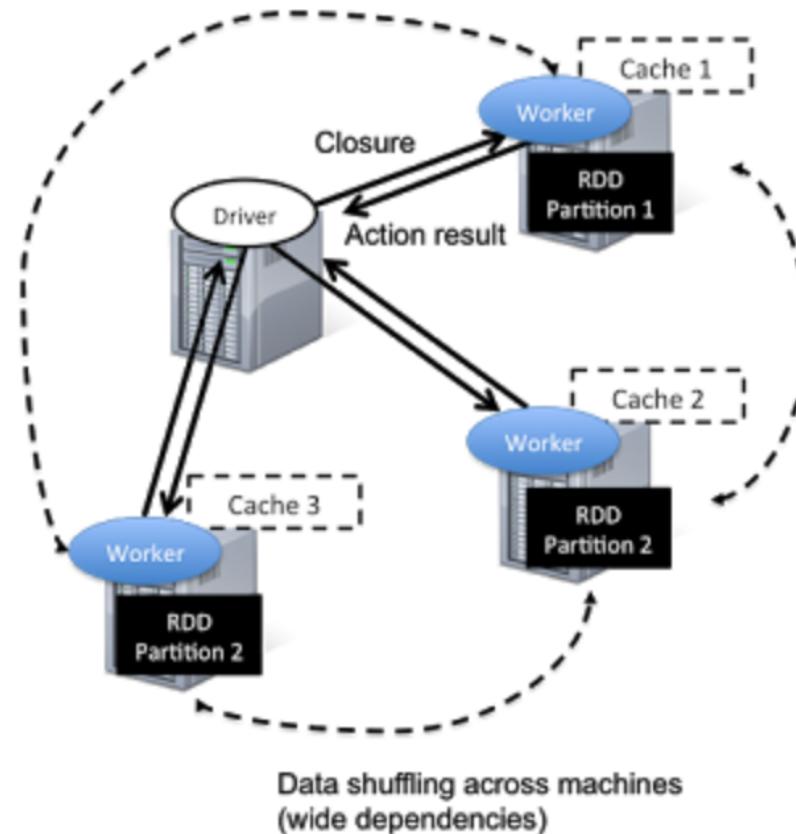
credit: Sparkcamp @ Strata CA: Intro to Apache Spark with Hands-on Tutorials



*The Magic of Spark*  
*RDD's*  
*Transformations*  
*Actions*

# Resilient Distributed Dataset (RDD)

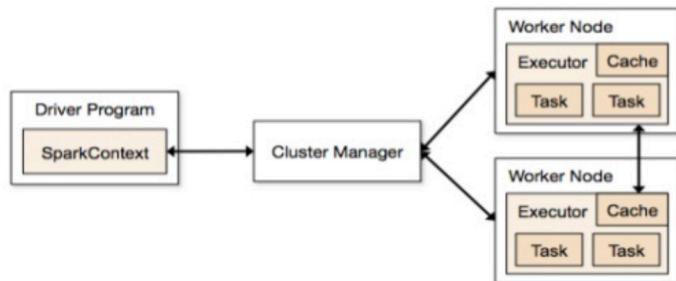
- **RDD** is a basic abstraction
- **Immutable**, partitioned collection of elements that can be operated in parallel
- Basic operations - map, filter, persist
- Multiple implementations - PairRDD <key,value> and Sequence Files



## How do I create an RDD?

1. Parallelized Collections
2. External Datasets
3. Streaming Data

### Execution Flow



```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

<http://spark.apache.org/docs/latest/cluster-overview.html>

```
>>> distFile = sc.textFile("data.txt")
```

## **What can I do with an RDD?**

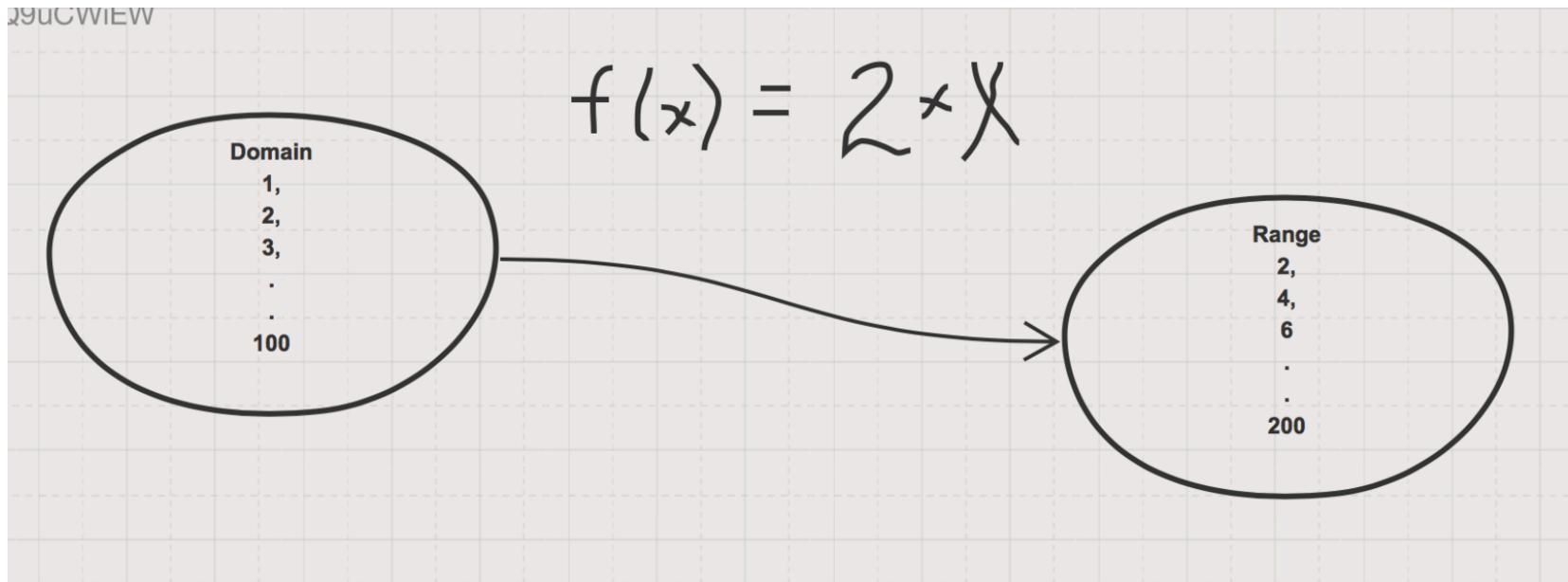
# RDD Operations

RDDs support two types of operations: *transformations*, which create a new dataset from an existing one, and *actions*, which return a value to the driver program after running a computation on the dataset.

*Map* is a transformation that passes each dataset element through a function and returns a new RDD representing the results.

*Reduce* is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program.

## *What functions? Monoids*



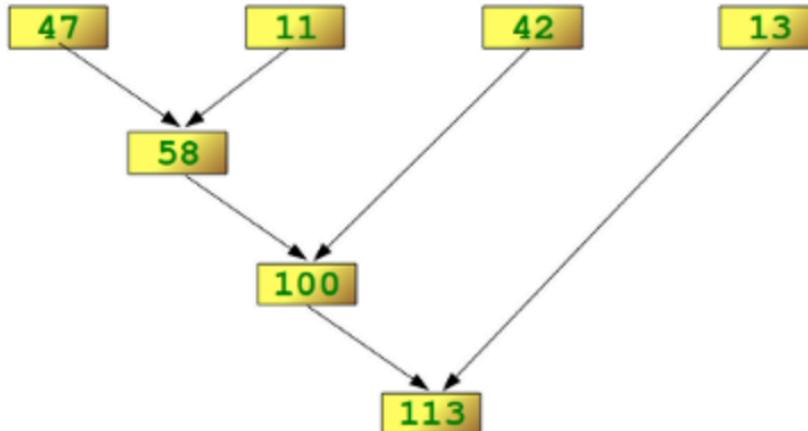
```
range = domain.map(lambda x : 2*x)
```

```
sum = domain.reduce(lambda x,y : x+y)
```

We illustrate this process in the following example:

```
>>> reduce(lambda x,y: x+y, [47,11,42,13])  
113
```

The following diagram shows the intermediate steps of the calculation:



*credit: <http://www.python-course.eu/lambda.php>*

```
In [17]: reduced = mappedRdd.reduce(lambda x, y: x+y)  
print type(reduced)  
print reduced
```

```
<type 'int'>  
328350
```

# *The Map and Reduce Abstraction*

## *Lab 1*

```
In [19]: nums = range(10)
print nums

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [20]: #Python only

map(lambda x: x*x, nums)
```

```
Out[20]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
In [27]: #Spark - push nums list onto five executors
sparkNums = sc.parallelize(nums, 5)
#map - this is a transformation
squares = sparkNums.map(lambda x: x*x)
#print result - this is an action - push results back to the driver
print squares.collect()
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

RDD's are a collection of records

```
rdd = sc.parallelize(range(1000), 5)
```

Transformations create new RDD's from  
existing ones

```
errors = rdd.filter(lambda line: "ERROR" in line)
```

Actions materialize a value in the user  
program

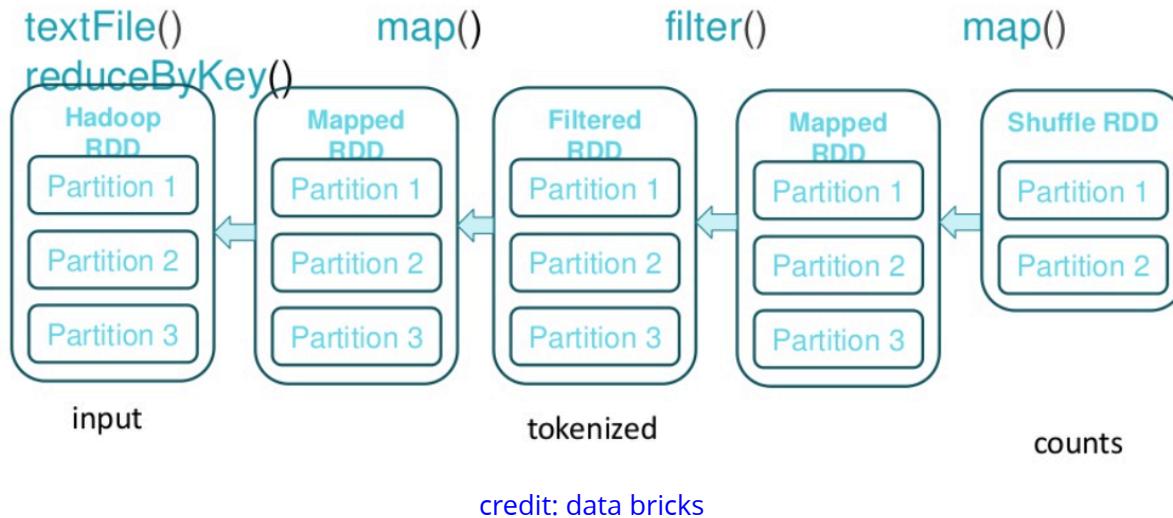
```
size = errors.count()
```

# THE DAG

## (directed acyclic graph)

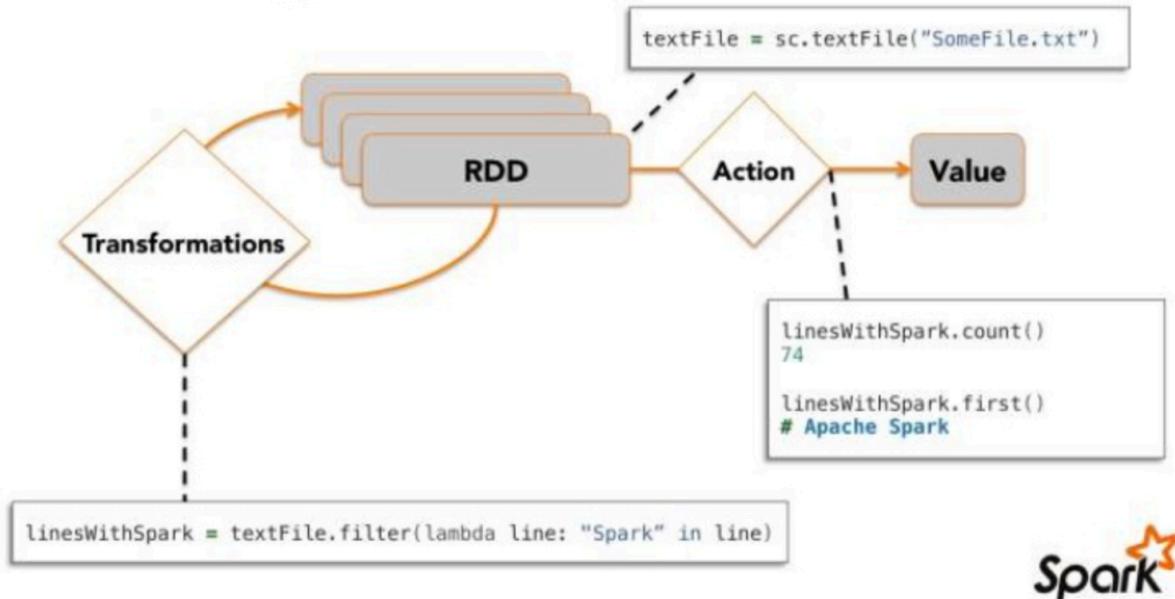
```
sc.textFile().map().filter().map().reduceByKey()
```

### DAG View of RDD's



# *Lazy Evaluation*

## Working With RDDs



*Code runs only upon encountering an action*

credit: link

# Persistence layers for Spark

## Distributed

- Hadoop (HDFS)
- Local file system
- Cassandra
- Amazon S3
- Hive
- Base

## File formats

- Text - CSV, Plain Txt
- Sequence File
- AvRO
- Parquet

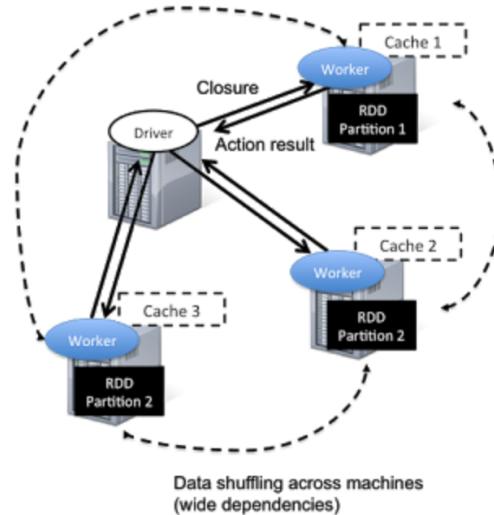
## WHEN IS the DAG EXECUTED?

```
import sys
import os

logFile = os.path.join('data', 'logfile')

""" Read and parse log file """
parsed_logs = (sc
    .textFile(logFile)
    .filter(lambda line: 'GET' in line))

print parsed_logs.count()
```



```
in24.inetnebr.com - - 01/Aug/1995:00:00:01 -0400 "GET /shuttle/missions/sts-6.txt HTTP/1.0" 200 1839-
uplherc.upl.com - - 01/Aug/1995:00:00:07 -0400 "GET / HTTP/1.0" 304 0-
uplherc.upl.com - - 01/Aug/1995:00:00:08 -0400 "GET /images/ksc.gif HTTP/1.0" 304 0-
uplherc.upl.com - - 01/Aug/1995:00:00:08 -0400 "GET /images/MOSAIC.gif HTTP/1.0" 304 0-
```

## Transformations

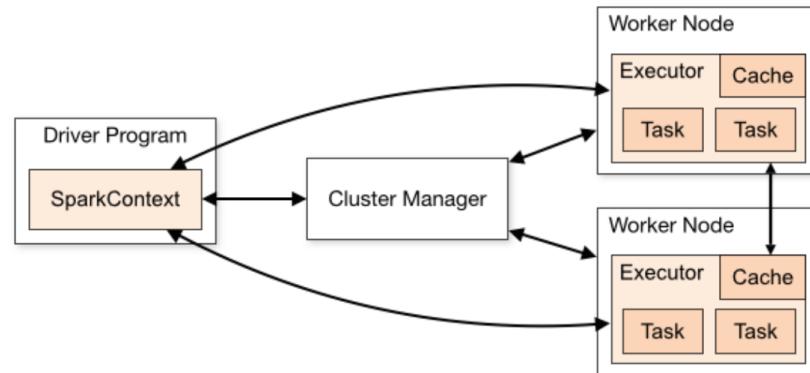
<code>map(func)</code>	<code>reduceByKey(func, [numTasks])</code>
<code>filter(func)</code>	<code>aggregateByKey(zeroValue)(seqOp,</code>
<code>flatMap(func)</code>	<code>combOp, [numTasks])</code>
<code>mapPartitions(func)</code>	<code>join(otherDataset, [numTasks])</code>
<code>mapPartitionsWithIndex(func)</code>	<code>cogroup(otherDataset, [numTasks])</code>
<code>union(otherDataset)</code>	<code>cartesian(otherDataset)</code>
<code>intersection(otherDataset)</code>	<code>pipe(command, [envVars])</code>
<code>distinct([numTasks]))</code>	<code>coalesce(numPartitions)</code>
<code>groupByKey([numTasks])</code>	<code>sample(withReplacement, fraction, seed)</code>
<code>sortByKey([ascending], [numTasks])</code>	<code>repartition(numPartitions)</code>

## Actions

<code>reduce(func)</code>	<code>take(n)</code>
<code>collect()</code>	<code>takeSample(withReplacement, num, [seed])</code>
<code>count()</code>	<code>takeOrdered(n, [ordering])</code>
<code>first()</code>	<code>saveAsTextFile(path)</code>
<code>countByKey()</code>	<code>saveAsSequenceFile(path)</code>
<code>foreach(func)</code>	<code>saveAsObjectFile(path)</code> (Only Java and Scala)

## Lab1: sparkRDDs

*how many executors?*



```
#range(start, end=None, step=1, numSlices=None)

rdd = sc.parallelize(xrange(0, 100, 1), 5)
print rdd.collect()
```

Text

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31
, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 9
0, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

What does this look like?

- `glom`: Returns an RDD list from each partition of an RDD.
- `collect`: Returns a list from all elements of an RDD.

```
for x in rdd.glom().collect():
    print x
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
[40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59]
[60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79]
[80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

# Key Value Pairs

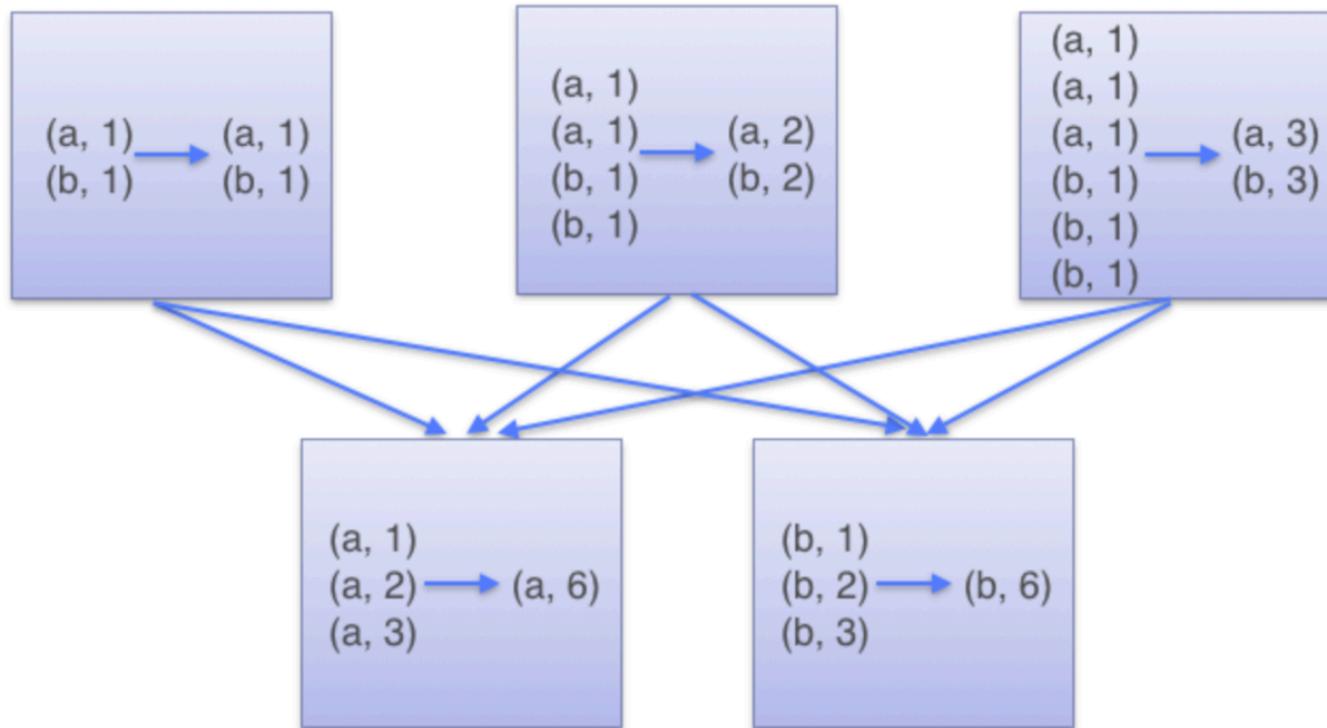
## Transformations

- rdd.reduceByKey(func)
- rdd.groupByKey()
- rdd.mapValues(fund)
- rdd.keys()
- rdd.values()
- rdd.sortByKey()

## Actions

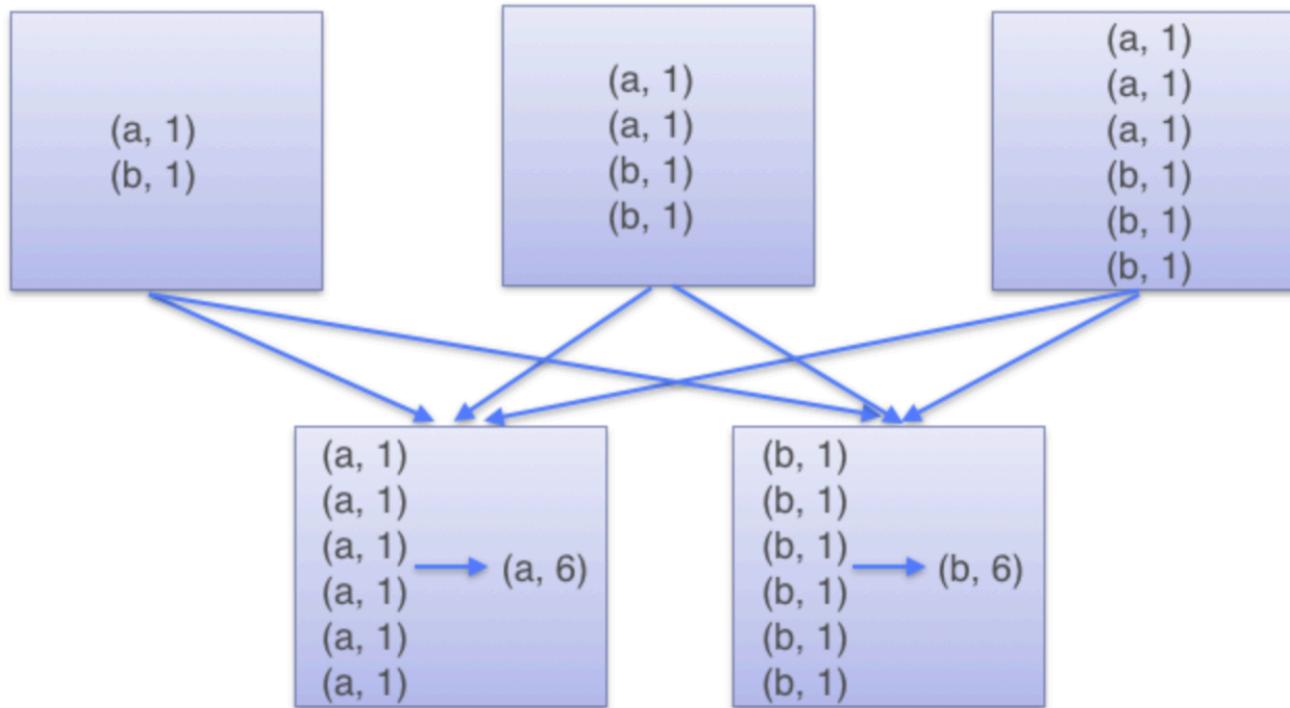
- rdd.countByKey()
- rdd.collectAsMap()
- rdd.lookup(key)

# ReduceByKey



credit: [https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best\\_practices/prefer\\_reducebykey\\_over\\_groupbykey.html](https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html)

# GroupByKey



credit: [https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best\\_practices/prefer\\_reducebykey\\_over\\_groupbykey.html](https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html)

```
val lines = sc.textFile("input.txt")
val words = lines.flatMap(line => line.split(" "))
val ones = words.map(s => (s,1))
val count = ones.reduceByKey((a,b) => a + b)
val result = count.collectAsMap()
```

**RDD lineage DAG built on driver side**  
**data source RDD**  
**transformation RDD, transformation**  
**action, action RDD**

# Lab 3 WordCount

# Cluster Deployment

- Standalone Deploy Mode
  - simplest way to deploy Spark on a private cluster
- Amazon EC2
  - EC2 scripts are available
  - Very quick launching a new cluster
- Apache Mesos
- Hadoop YARN