



Introduction to Apache Spark

LYU Zishen, YAO Fangjie

Contents

- Background
- Features & Designs
- Programming Spark

Background

The Big Data Problem

- Data growing faster than computation speeds
- Growing data sources
 - » Web, mobile, scientific, ...
- Storage getting cheaper
 - » Size doubling every 18 months
- But, stalling CPU speeds and storage bottlenecks

Big Data Examples

- Facebook's daily logs: 60 TB"
- 1,000 genomes project: 200 TB"
- Google web index: 10+ PB"
- Cost of 1 TB of disk: ~\$35"
- Time to read 1 TB from disk: 3 hours !(100 MB/s)"

Tackling Big Data Problems

- Scale out instead of scaling up
- Divide and conquer
- Move computations to data

First solution: Google MapReduce

Opensource version: Hadoop MapReduce

Features & Designs



What is Spark

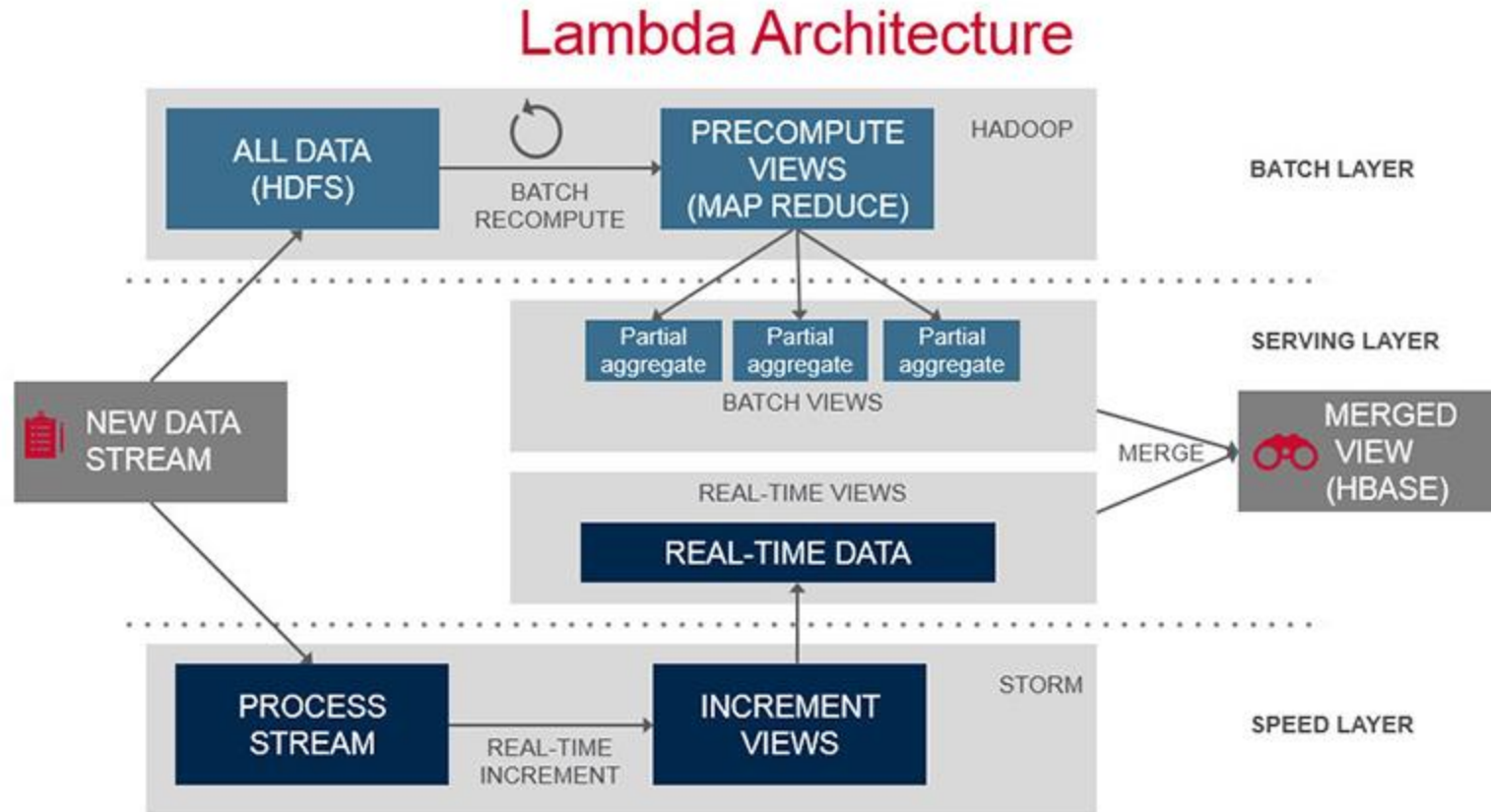
- Originally developed in 2009 in UC Berkeley's AMPLab, and open sourced in 2010 as an Apache project.
- A cluster computing platform designed to be ***fast*** and ***general purpose*** for big data processing.
- Written in Scala and runs on Java Virtual Machine (JVM) environment.
- Simple APIs in Python, Java, Scala, and SQL, and rich built-in libraries.
- Can run in Hadoop clusters and access any Hadoop data source.

Why use Spark

Hadoop MapReduce

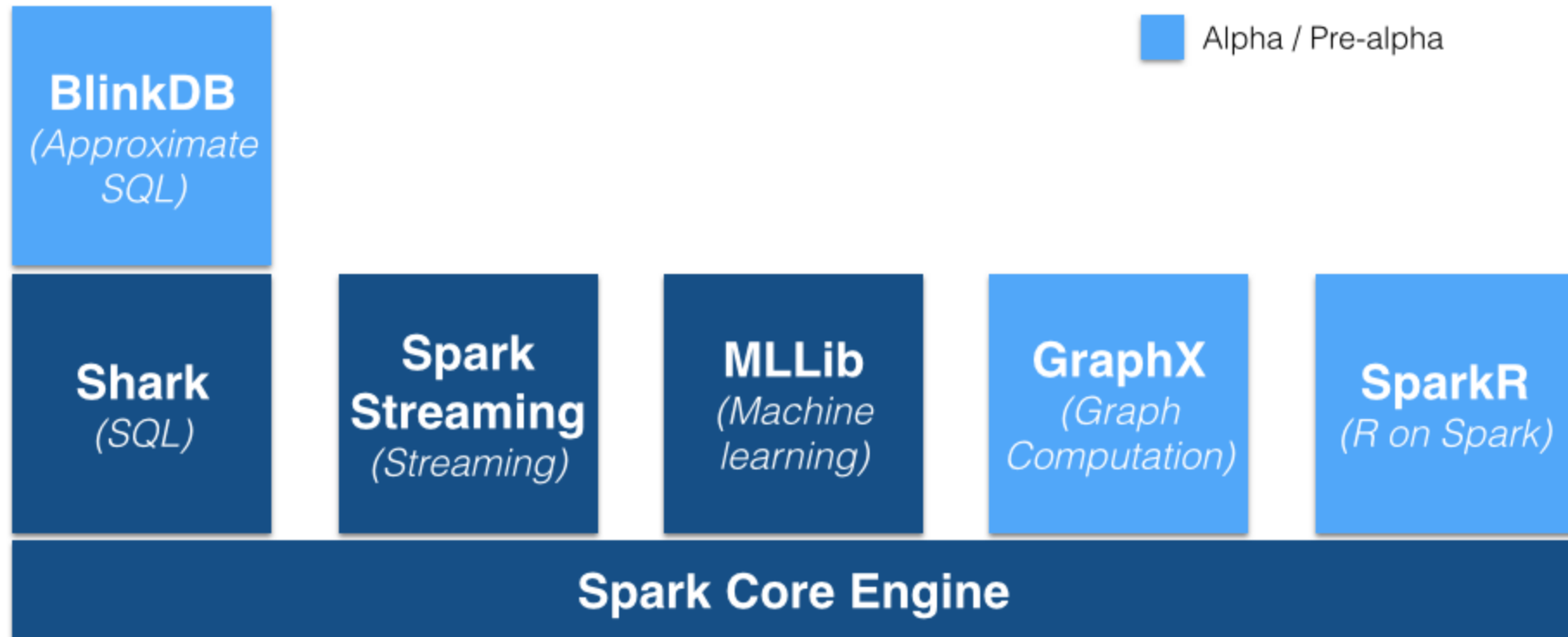
- The latency is very high, can not be used for real-time processing.
- One job has only two phases: map and reduce.
- The abstract level is low, it is not very easy to use.
- Reduce Tasks can start only after all the Map Tasks completed.
- The performance is not good for iterative data processing.

Lambda Architecture



<https://gigaom.com/2014/06/28/4-reasons-why-spark-could-jolt-hadoop-into-hyperdrive/>

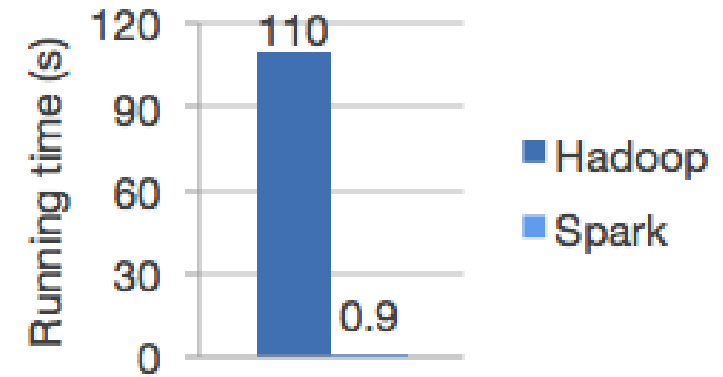
Spark Stack



<https://gigaom.com/2014/06/28/4-reasons-why-spark-could-jolt-hadoop-into-hyperdrive/>

Features

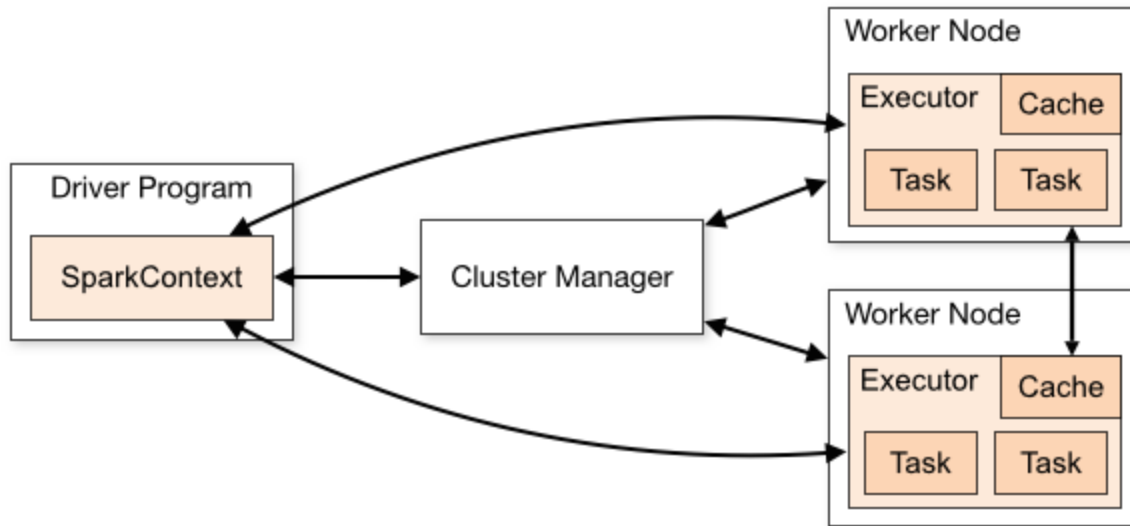
- Lightning Fast Processing
- Support for Sophisticated Analytics
- Real Time Stream Processing
- Ability to Integrate with Hadoop and Existing Hadoop Data
- Ease of use as it supports multiple languages
- Active and Expanding Community



Logistic regression in Hadoop and Spark

<http://spark.apache.org/>

Spark and Cluster



- 1) Connects to a cluster manager which allocate resources across applications
- 2) Acquires executors on cluster nodes, i.e. worker processes to do computation and store data
- 3) Send application code to the executors
- 4) Send tasks for the executors to run

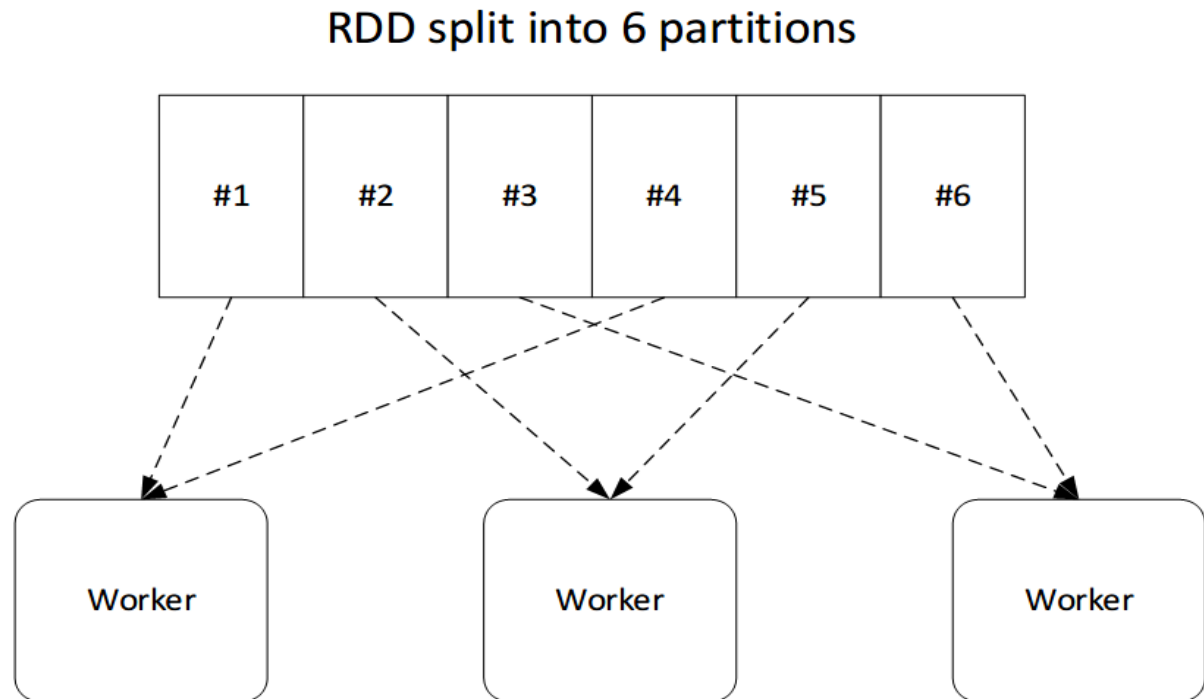
- Spark's own standalone cluster manager
- Yarn
- Mesos

Resilient Distributed Datasets (RDD)

- The primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel
- Designed to support in-memory data storage
- Immutable once constructed
- Track lineage information to efficiently re-compute lost data
- Two types of operations on RDDs
 - transformations: create a new RDD by changing the original through processes such as mapping, filtering
 - actions: measure but do not change the original data
- Can be persisted into storage in memory or on disk

RDD

- Split into multiple partitions, which can be computed on different nodes of the cluster.
- More partitions, more parallelism.

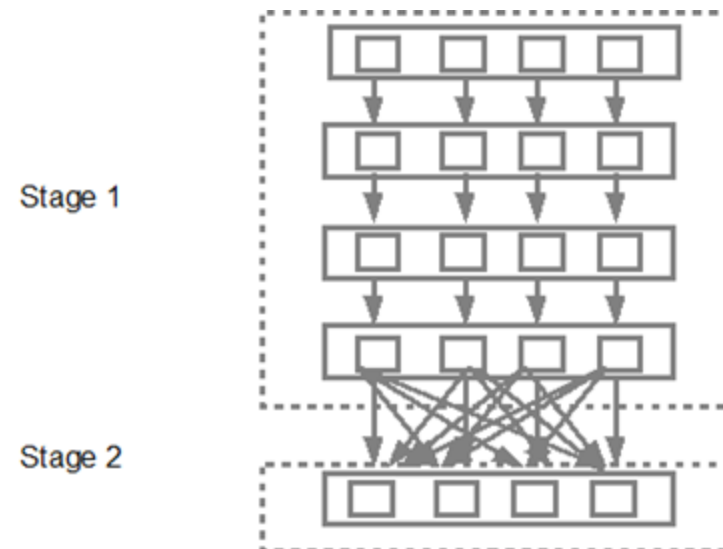
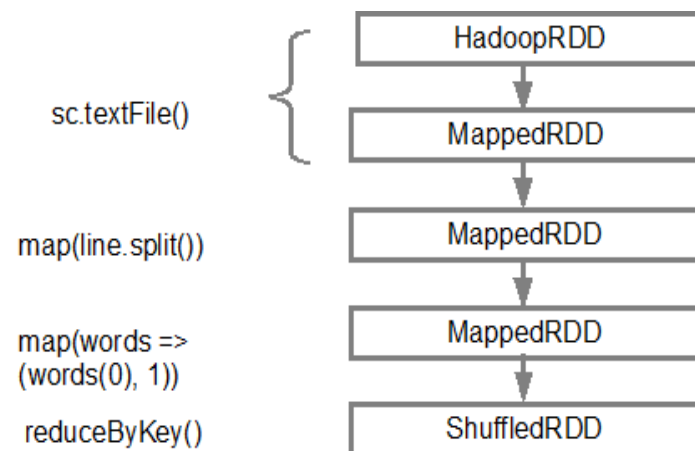
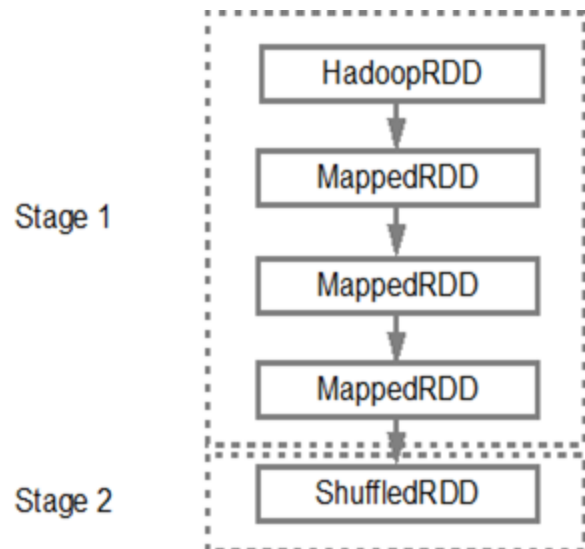


Directed Acyclic Graph (DAG)

- The sequence of commands implicitly defines a DAG of RDD objects (RDD lineage) that will be used later when an action is called.
- Each RDD maintains a pointer to one or more parents along with the metadata about what type of relationship it has with the parent.
- Once the DAG is built, the Spark scheduler creates a physical execution plan. The DAG scheduler splits the graph into multiple stages, the stages are created based on the transformations.
- The DAG scheduler will then submit the stages into the task scheduler. The number of tasks submitted depends on the number of partitions.

DAG Example

```
val input = sc.textFile("log.txt")  
val splitedLines = input.map(line => line.split(" "))  
                        .map(words => (words(0), 1))  
                        .reduceByKey{(a,b) => a + b}
```



General Work Procedure

1. Create some input RDDs from external data.
2. Transform them to define new RDDs through transformations.
3. Persist any intermediate RDDs that will need to be reused.
4. Launch actions to kick off a parallel computation, which is then optimized and executed by Spark.

Programming Spark in Python

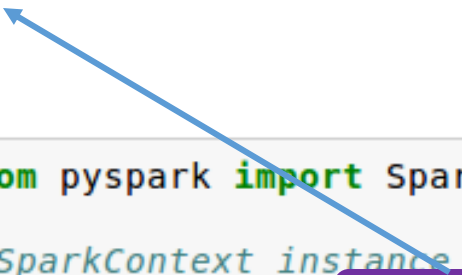
Installation

- 1) Install Java JDK
- 2) Download Spark and Unzip
- 3) Setup environment variables
- 4) Run it
 - Spark shell (for Scala): `./bin/spark-shell`
 - Spark shell for Python: `./bin/pyspark`
 - IPython Notebook

Spark Context

A Spark program first create a SparkContext object

The **master parameter** determines which cluster to use



```
In [1]: from pyspark import SparkContext
# SparkContext instance
sc = SparkContext('local[2]', 'pyspark')
print sc
```

```
<pyspark.context.SparkContext object at 0x7f1f746c49d0>
```

Construct RDDs

- By parallelizing existing Python collections
- By transforming existing RDDs
- From any file stored in HDFS or other storage systems supported by Hadoop

In [2]: *# create RDDs from collection*

```
data = range(1, 6)
```

```
print data
```

```
# no computation occurs here
```

```
# because Spark just simply records how to create the RDD
```

```
rangeRDD = sc.parallelize(data, 2)
```

```
rangeRDD
```

```
[1, 2, 3, 4, 5]
```

Out[2]: ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:423

In [3]: *# create RDDs from a file*

```
txtFile = sc.textFile('test.txt')
```

```
txtFile
```

Out[3]: MapPartitionsRDD[2] at textFile at NativeMethodAccessorImpl.java:-2

Transformations

- Create a new RDD from an existing one
- Lazy (not computed immediately), the transformed RDD gets computed when an action is run on it (default)
 - Help optimize the computations
 - Help recover from failures and stragglers

Some transformations

Transformation	Description
<code>map(<i>func</i>)</code>	return a new distributed dataset formed by passing each element of the source through a function <i>func</i>
<code>filter(<i>func</i>)</code>	return a new dataset formed by selecting those elements of the source on which <i>func</i> returns true
<code>distinct([<i>numTasks</i>]))</code>	return a new dataset that contains the distinct elements of the source dataset
<code>flatMap(<i>func</i>)</code>	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)

Transformation Examples

```
In [5]: # map
timesthreeRDD = rangeRDD.map(lambda x: x * 3)
# collect is an action which triggers computation
print timesthreeRDD.collect()

[3, 6, 9, 12, 15]
```

```
In [6]: # filter
evenRDD = rangeRDD.filter(lambda x: x % 2 == 0)
print evenRDD.collect()

[2, 4]
```

```
In [8]: # flatMap
selfandtimestwoRDD = rangeRDD.flatMap(lambda x: [x, x*2])
print selfandtimestwoRDD.collect()

[1, 2, 2, 4, 3, 6, 4, 8, 5, 10]
```

Actions

Trigger the actual computations and return results

Action	Description
<code>reduce(func)</code>	aggregate dataset's elements using function <i>func</i> . <i>func</i> takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
<code>take(n)</code>	return an array with the first <i>n</i> elements
<code>collect()</code>	return all the elements as an array WARNING: make sure will fit in driver program
<code>takeOrdered(n, key=func)</code>	return <i>n</i> elements ordered in ascending order or as specified by the optional key function

Action Example

```
In [8]: # take  
print rangeRDD.take(3)
```

```
[1, 2, 3]
```

```
In [9]: # takeOrdered  
print rangeRDD.takeOrdered(3, lambda x: -1 * x)
```

```
[5, 4, 3]
```

pySpark Closures

- Spark automatically creates closures for
 - Functions that run on RDDs at workers
 - Any global variables used by those workers
- One closure per worker
 - Sent for every task
 - No communication between workers
 - Changes to global variables at workers are not sent to driver

Caching

- Save RDDs for reuse
- Programmer-specified storage level

Storage level	Meaning
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. (Default).
DISK_ONLY	Store the RDD partitions only on disk.
...	

<http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence>

Shared Variables

Extremely useful for iterative jobs and counting events

- Broadcast variables
 - Read-only variable cached on each worker rather than shipping a copy of it with tasks
 - Broadcast a value: `scval = sc.broadcast([1,2,3])`
 - Use the value of the broadcast variable: `scval.value`
- Accumulators
 - Variables that are only “added” to through an associative operation, usually used to implement counters or sums
 - Create a accumulator: `acval = sc.accumulator(0)`
 - Add a value: `acval.add(3)`
 - Read the value: `acval.value`

WordCount

Example Data:

Big data is a buzzword nowadays, many companies use big data processing platforms to tackle big data problems.

Apache Spark has quite some advantages compared to others, and is rapidly becoming the compute engine of choice for big data.

This talk covers Apache Spark's architecture, programming model and how to write basic application with its commonly used APIs in Python.

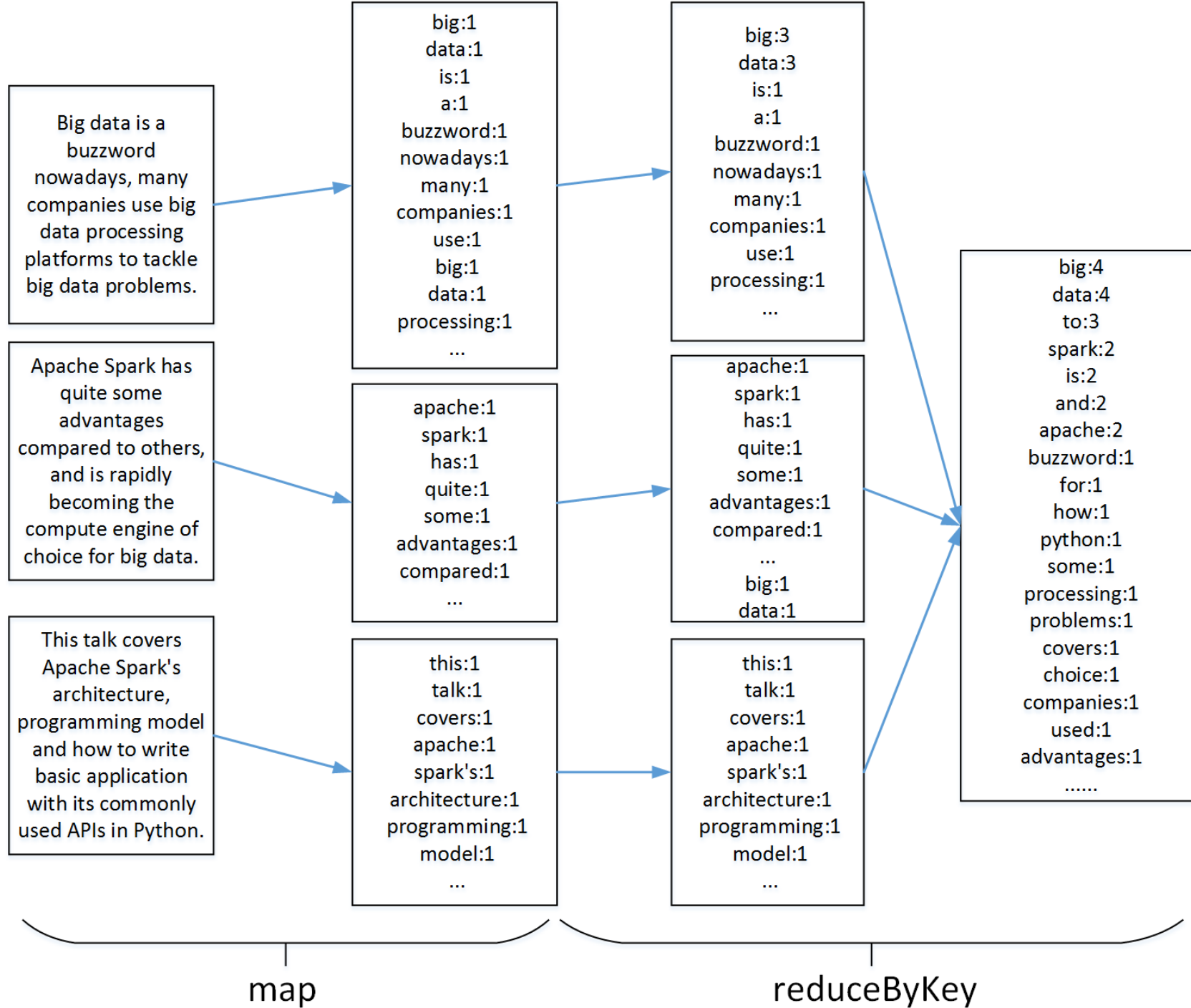
Non-distributed approach

- Create a dictionary object, word as key, count as value
- Loop over the input text, for each word
 - If not in the dictionary, set its value as 1
 - If in the dictionary, increment the value by 1

Equivalent to processing a bunch of (word, 1) tuples

Not scalable!

Partition the data, process these partition simultaneously and combine results together



WordCount

```
In [10]: from operator import add
import re

def tokenize(text):
    # [^\w] matches anything that's not alphanumeric or underscore
    return re.sub(r'[^\w]', ' ', text).lower().split()

words = txtFile.flatMap(tokenize)

wcpairs = words.map(lambda x: (x,1))

counts = wcpairs.reduceByKey(add)
# print counts

# apply 'collect' action
wresult = counts.collect()
# sort the result for output
resultsorted = sorted(wresult, key=(lambda (w,c): c), reverse=True)

print resultsorted
```

Simple Community Detection

Online social networks

- Symmetric (e.g. Facebook)
 - A is a friend of B, then B is also a friend of A.
- Asymmetric (e.g. Twitter)
 - A is a friend of B, then B may not be a friend of A.
 - A is called the follower, B the followee.



Common requirement to detect communities among users.

Useful for recommending new friends to user.

Simple Community Detection

People belong to the same community when they have high similarity.

Here similarity is measured by the number of common followees.

Goal: find K most similar persons for every user

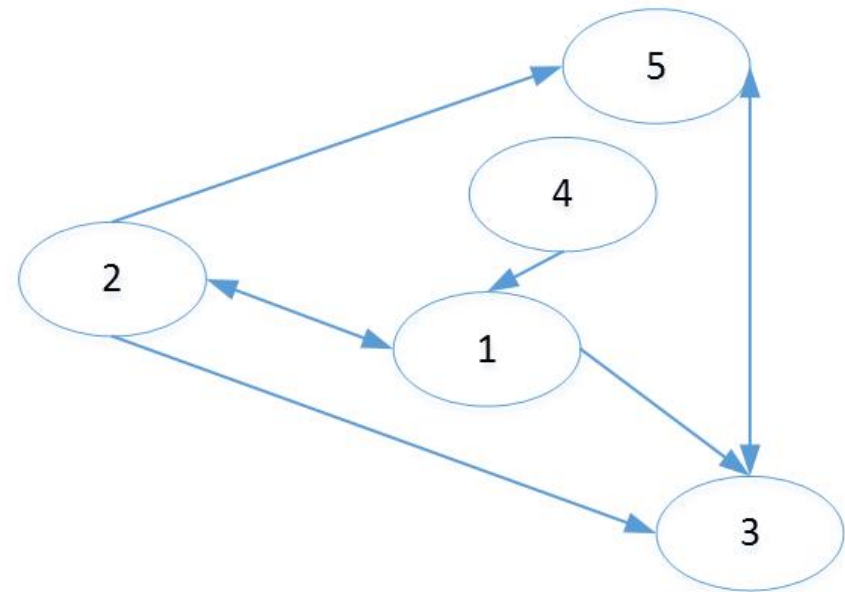
Sample Data

1: 2,4

2: 1

3: 1,2,5

5: 1,2,3



Simple Community Detection

For a user (A)

- Find all persons who has followed the same person with A
- Calculate their similarity with A
- Sort by similarity, get the top K persons



Check the code in IPython notebook

Thank you!

Acknowledgement

Several slides are adapted from the lecture notes of BerkeleyX's CS100.1x Introduction to Big Data with Apache Spark on <https://www.edx.org/> .