DAMLAS Part 2
Spark Introduction
+ Homework for Tuesday, 7/13

James G. Shanahan [1,2]

[1]Church and Duncan Group, *[2]iSchool UC Berkeley, CA*

EMAIL: James_DOT_Shanahan_AT_gmail_DOT_com

Part 2 Lecture 2

July 13, 2016

Homework for 7/13/2016

- **Finish NaiveBayes stateless**
- **Write STAR proposal for any project**
  - * STAR (Situation, Task, Action, Result) - 100 words regarding your proposed projects
- **Install Spark on your local computers**
  - See section later in these slides
  - Please try the following notebook on your local Spark
    - http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/ya26nezbyz0d1ly/1-Spark-conf-test.ipynb
    - WordCount: https://www.dropbox.com/s/ul0l3q98w54dr8x/WordCount.ipynb?dl=0
    - In Spark word count example, play around by adding features. (For example, filter words out, etc.)
    - Familiarize with lambda functions
- **Read Chapters 2, 3, and 4 in the following text book**
  - https://www.dropbox.com/s/m4xxds3po5byyg6/Learning_Spark.pdf?dl=0

Part 1

- **Part 1: Introduction**
  - Welcome Survey
  - Install Spark
  - Background and Motivation
    - Big Data Science
    - Functional Programming
    - Poorman's Map-Reduce (to dividing and conquering)

Part 2: Spark Intro and Basics

- **Part 2: Spark Intro and basics**
  - Base RDD
  - Fault tolerance (and lineage)
  - Transformations and Actions
  - Persistence
  - Animated Example
  - Pair RDDs
  - Word count example

Part 3: Machine Learning in Spark

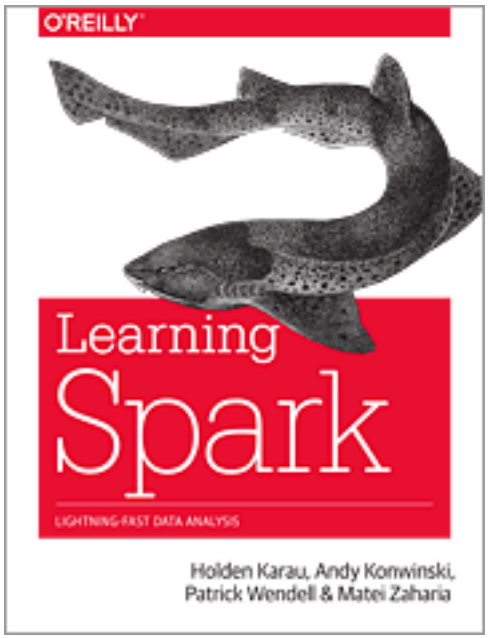
- **Data Frames**
- **MLLib**
- **Write your own algos**
  - Linear regression (Ridge and Lasso)
  - Logistic Regression
- **Pipelines**
- **R and Spark**
  - Linear Regression example
- **Graphs in Spark**
- **Case studies**
  - Flight delay
  - Mobile advertising
  - Social Networks

Tutorial Outline

- **Part 1: Introduction**
  - Welcome Survey
  - Install Spark
  - Background and motivation
- **Part 2: Spark Intro and basics**
  - fundamental Spark concepts, including Spark Core, data frames, the Spark Shell, Spark Streaming, Spark SQL and vertical libraries such as MLlib and GraphX;
- **Part 3: Machine learning in Spark**
  - will focus on hands-on algorithmic design and development with Spark developing algorithms from scratch such as linear regression, logistic regression, graph processing algorithms such as pagerank/shortest path, *etc.*
- **Part 4: Wrap up**
  - Spark 1.5 and beyond
  - Summary

Reference material

- **Book: Learning Spark: Lightning-Fast Big Data Analysis**
  - By Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia
  - Chapters 2, 3, and 4, and beyond



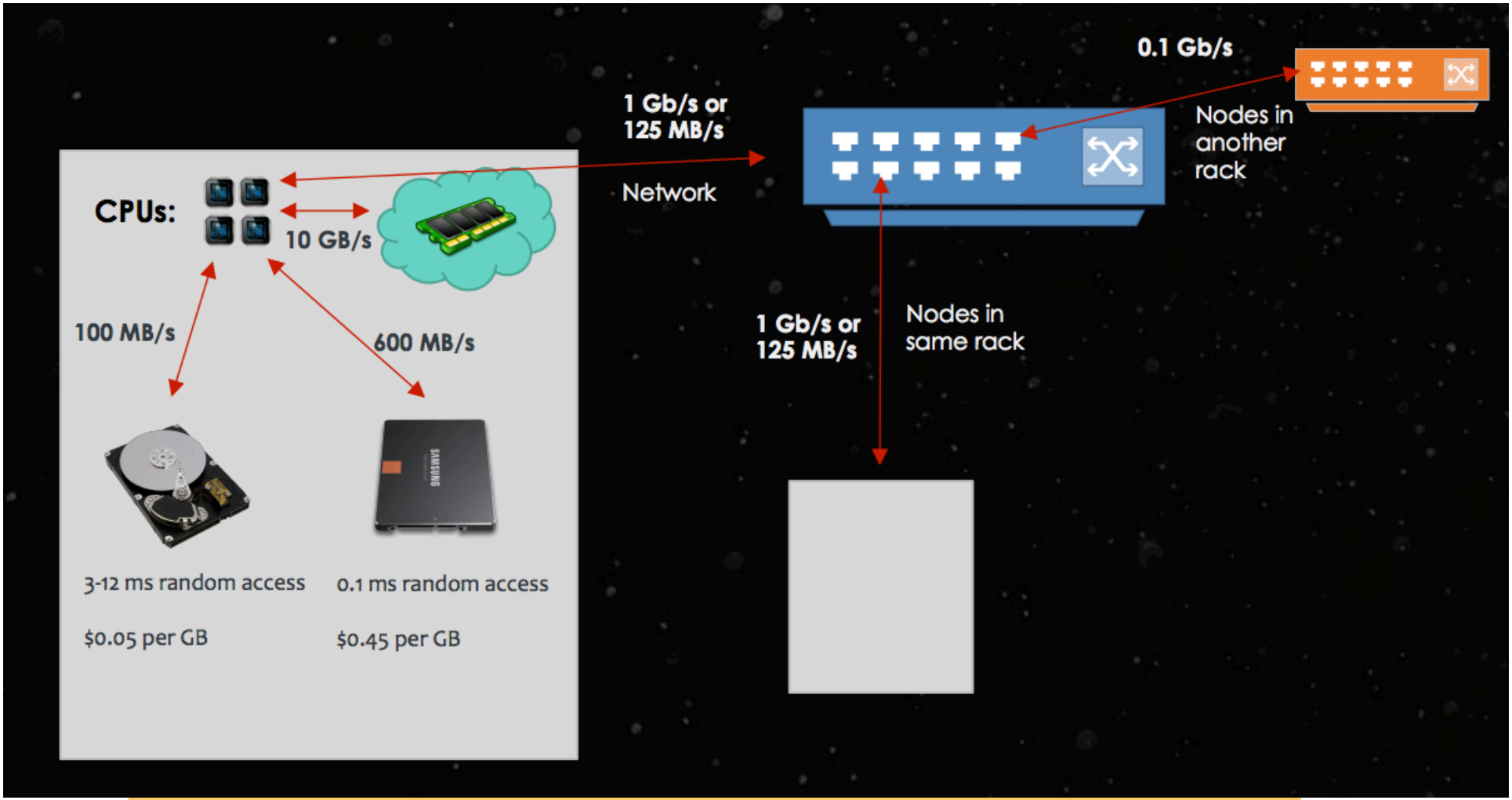
- **https://spark.apache.org/docs/latest/programming-guide.html**

Spark Online Resources

- http://spark.apache.org/research.html
- Spark 1.4.1 released **(Jul 15, 2015)**
- Spark Summit 2015 Videos Posted**(Jun 29, 2015)**

- **Reza Zadeh's presentation on Machine Learning in Spark @ Spark Summit 2015 in San Francisco (assumes strong understanding of programming in Spark)**

CPU ← → Memory 100X faster than HD
Across network: 10X to 100X

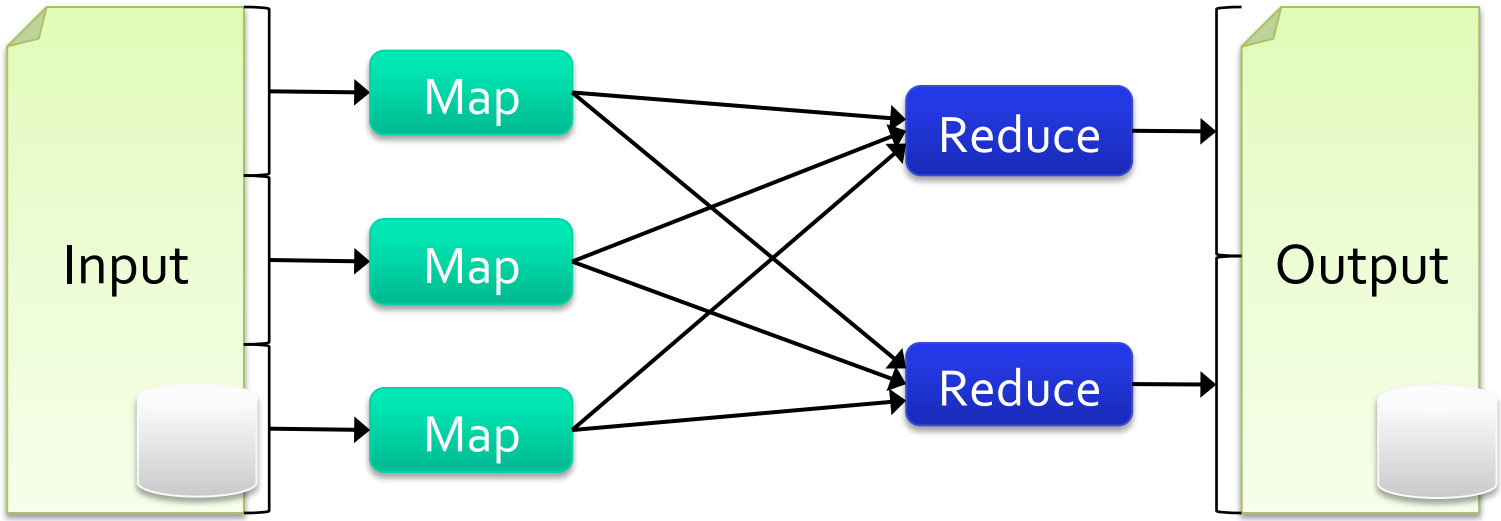


CPU ← → Memory 100X faster than HD
Across network: on same rack 10X; across the network 100X

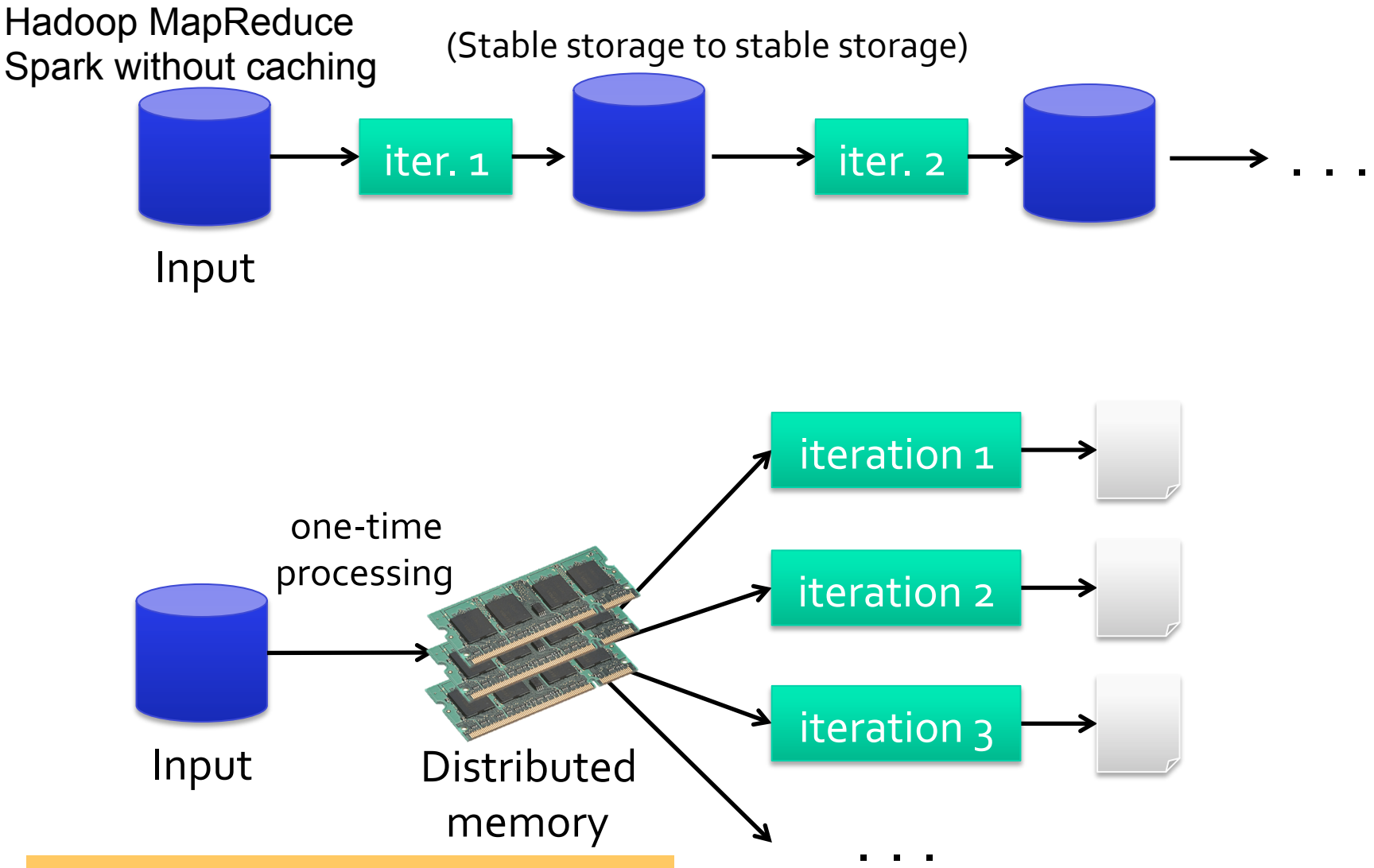
Motivation: Read from file; Process; Write to file

Current popular programming models for clusters transform data flowing from stable storage to stable storage (fault tolerant storage)

e.g., MapReduce:



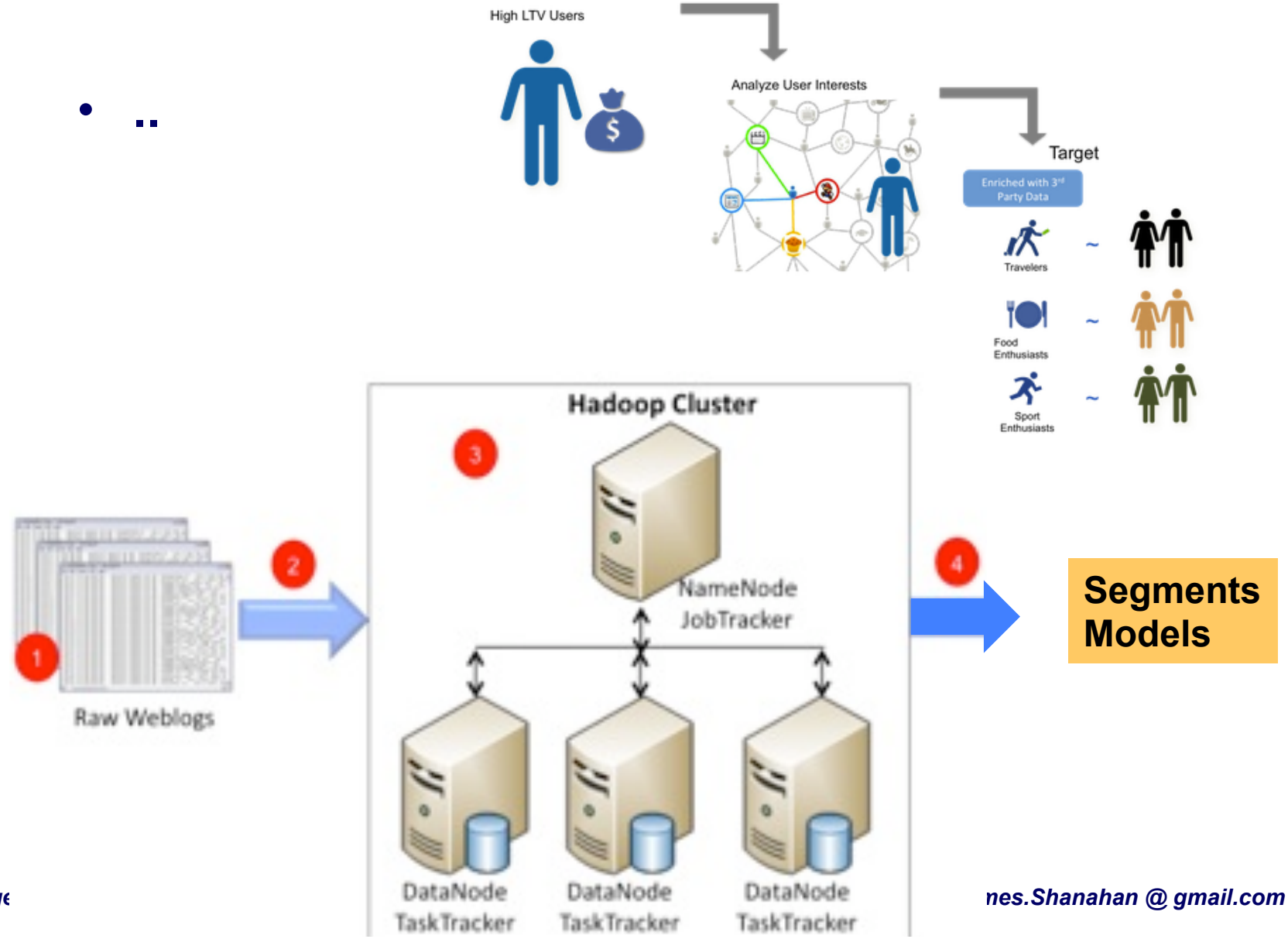
Goal: Keep Working Set in RAM



Spend 90% of time do I/O

MapReduce: Typical usecase

- ..



Hadoop is a black box and lacks REPL; Complex Jobs not possible

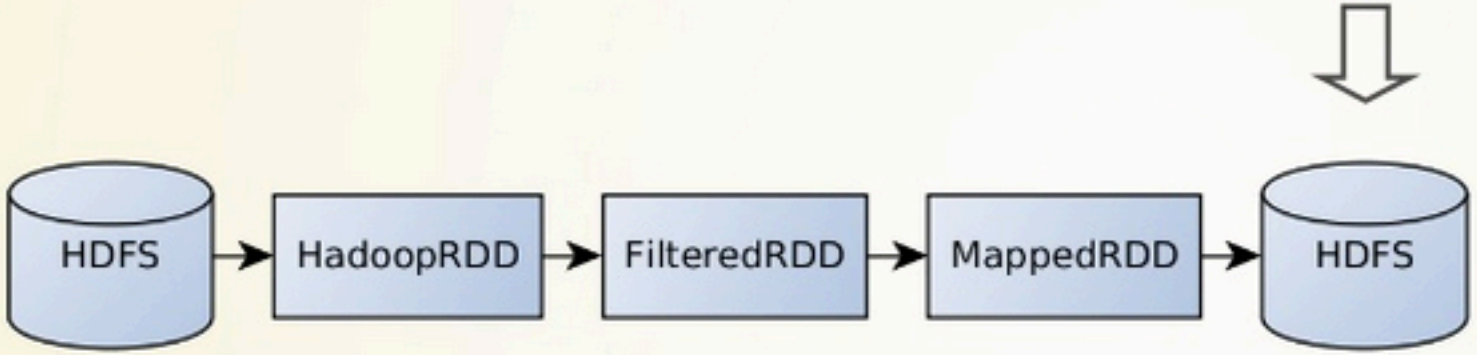
Read–eval–print loop

- **Run the WordCount example job written in Java.**
  - hduser@ubuntu:/usr/local/hadoop$ ***bin/hadoop jar hadoop*examples*.jar wordcount /user/hduser/gutenberg /user/hduser/gutenberg-output***
- **This command will read all the files in the HDFS directory /user/hduser/gutenberg, process it, and store the result in the HDFS directory /user/hduser/gutenberg-output.**
- **Hadoop is a black box**
  - provide input, process, get output
  - Difficult to manipulate/access the data in the stream

Hadoop not good with pipeline jobs

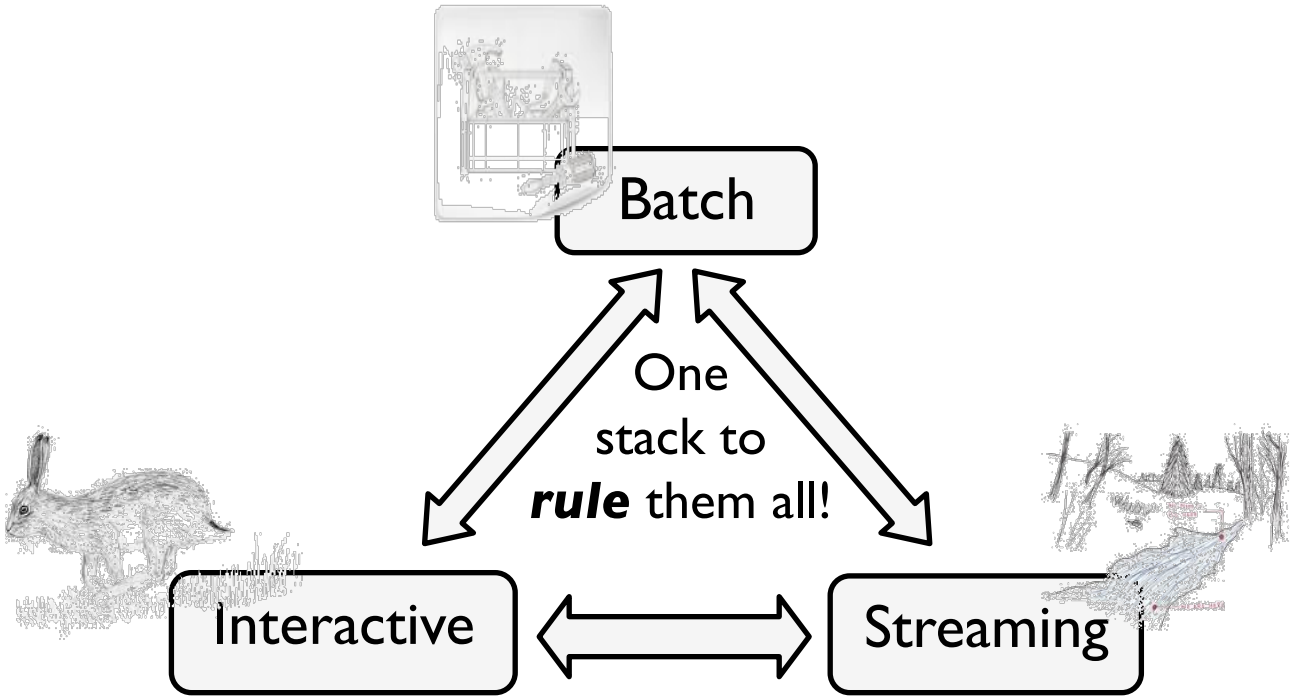
Step by Step

RDD: each row is a key, value pair



The RDD.saveAsTextFile() action triggers a job. Tasks are started on scheduled executors.

Goals for (Distributed) Systems for Big Data Science



- ***Easy*** to combine ***batch***, ***streaming***, and ***interactive*** computations
- ***Easy*** to develop ***sophisticated*** algorithms
- ***Compatible*** with existing open source ecosystem (Hadoop/HDFS)

Spark

In-Memory Cluster Computing for Iterative and Interactive Applications

[Zaharia, 2009, UC Berkeley]

Became an top level Apache project in 2013

What is Spark?

Fast and Expressive Cluster Computing System
Compatible with Apache Hadoop

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

2-5x less code

Efficient

- **General execution graphs**
- **In-memory storage**

Usable

- **Rich APIs in Java, Scala, Python, R (1.4)**
- **Interactive shell**

Spark Metrics

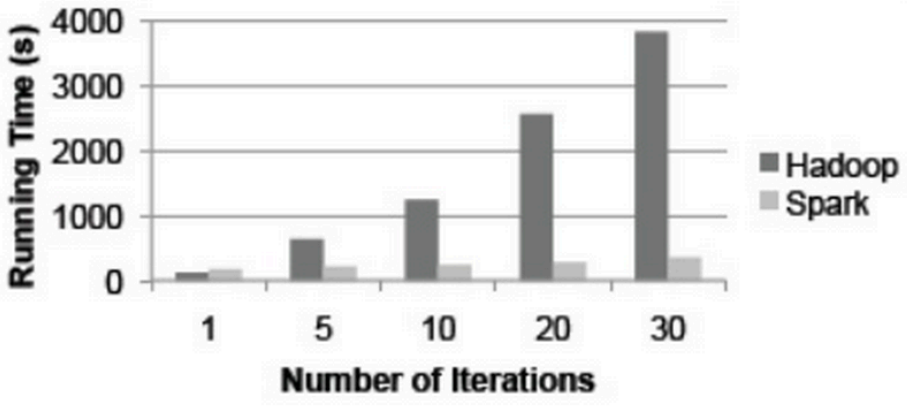
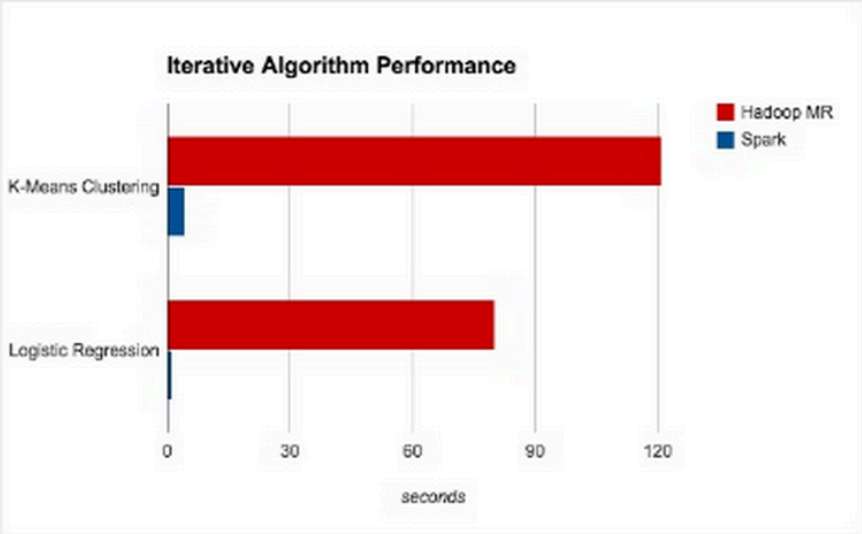
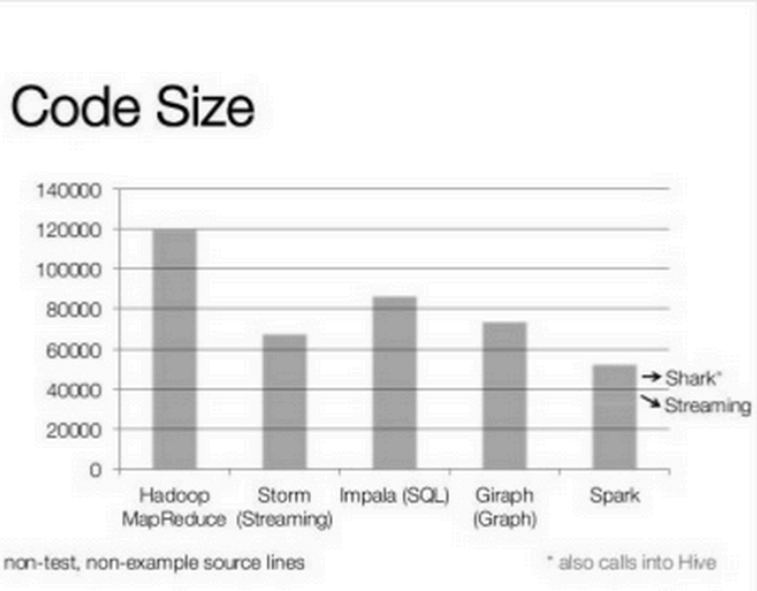


Figure 2: Logistic regression performance in Hadoop and Spark.





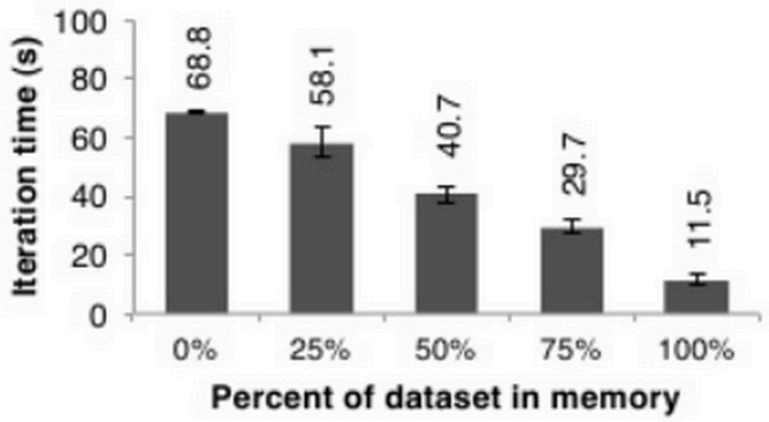


Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

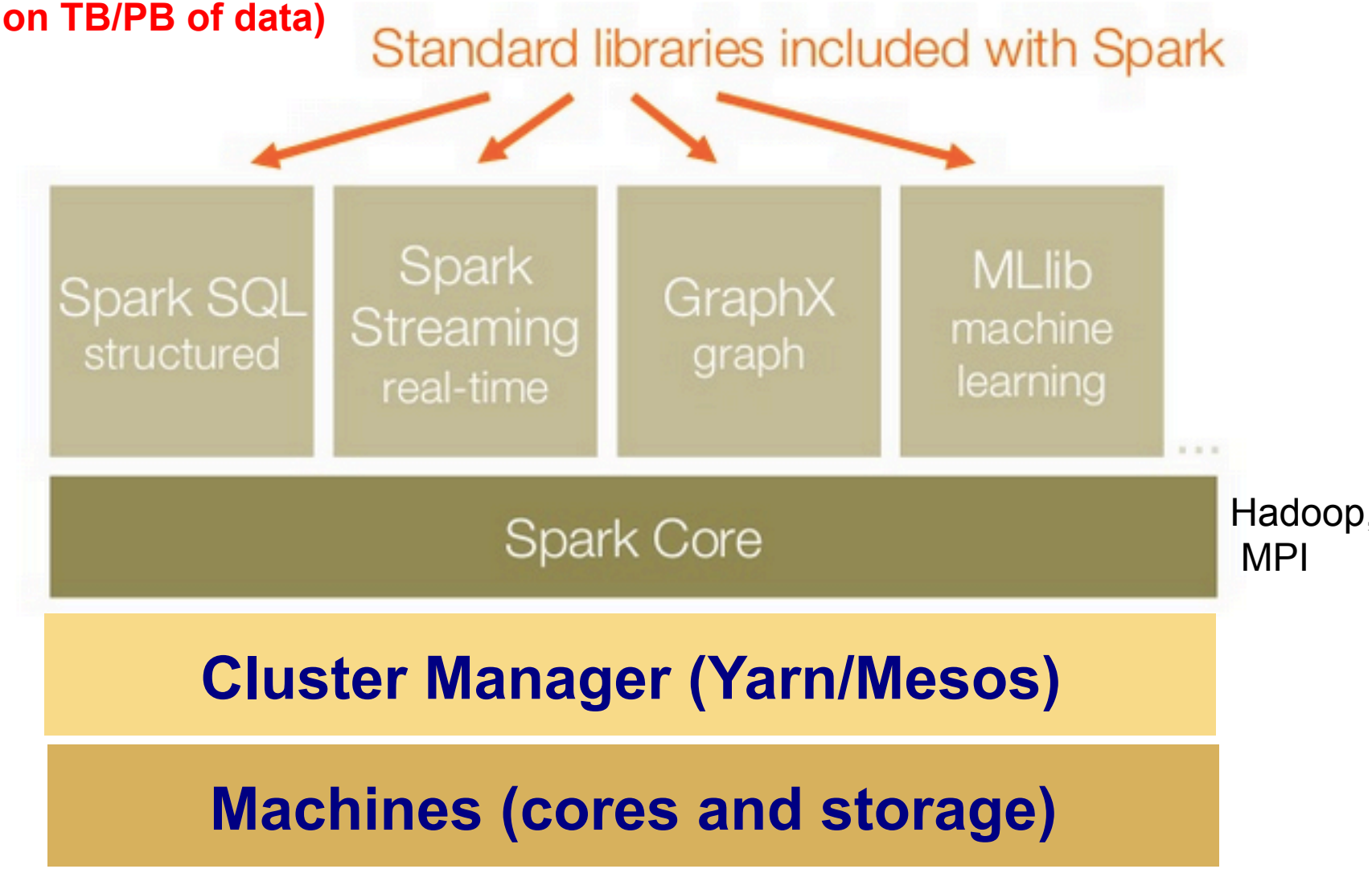
Spark

- **Data Management**
  - Split data
  - Ship data
  - Replicate data
  - Run tasks
- **Task management**
  - Distribute, track
- **Fault tolerance (nodes crash 1-5 in a 1000 per day)**
- **First class programming framework for distributing programming over huge volumes of data that leverages memory, disk and network resources and constraints**
- **REPL (Read–eval–print loop)**

Spark

Apache Spark is an open-source cluster computing framework for big data (works on TB/PB of data)

- .



Spark in local mode on a single computer or on a Spark Cluster

- **Apache Spark is an open-source cluster computing framework for big data (works on TB/PB of data)**

```
df= sc.textFile("hdfs://mydocs")
dfLen= df.map(length)
strLen= dfLen.reduce(sum)
dfLen.saveAsTextFile("strLens") #Action
```

INPUT	Key	Value
	d1	the quick brown fox
	d2	the fox ate the mouse
	d3	how now brown cow

OUTPUT	Key	Value
	d1	20
	d2	22
	d3	17

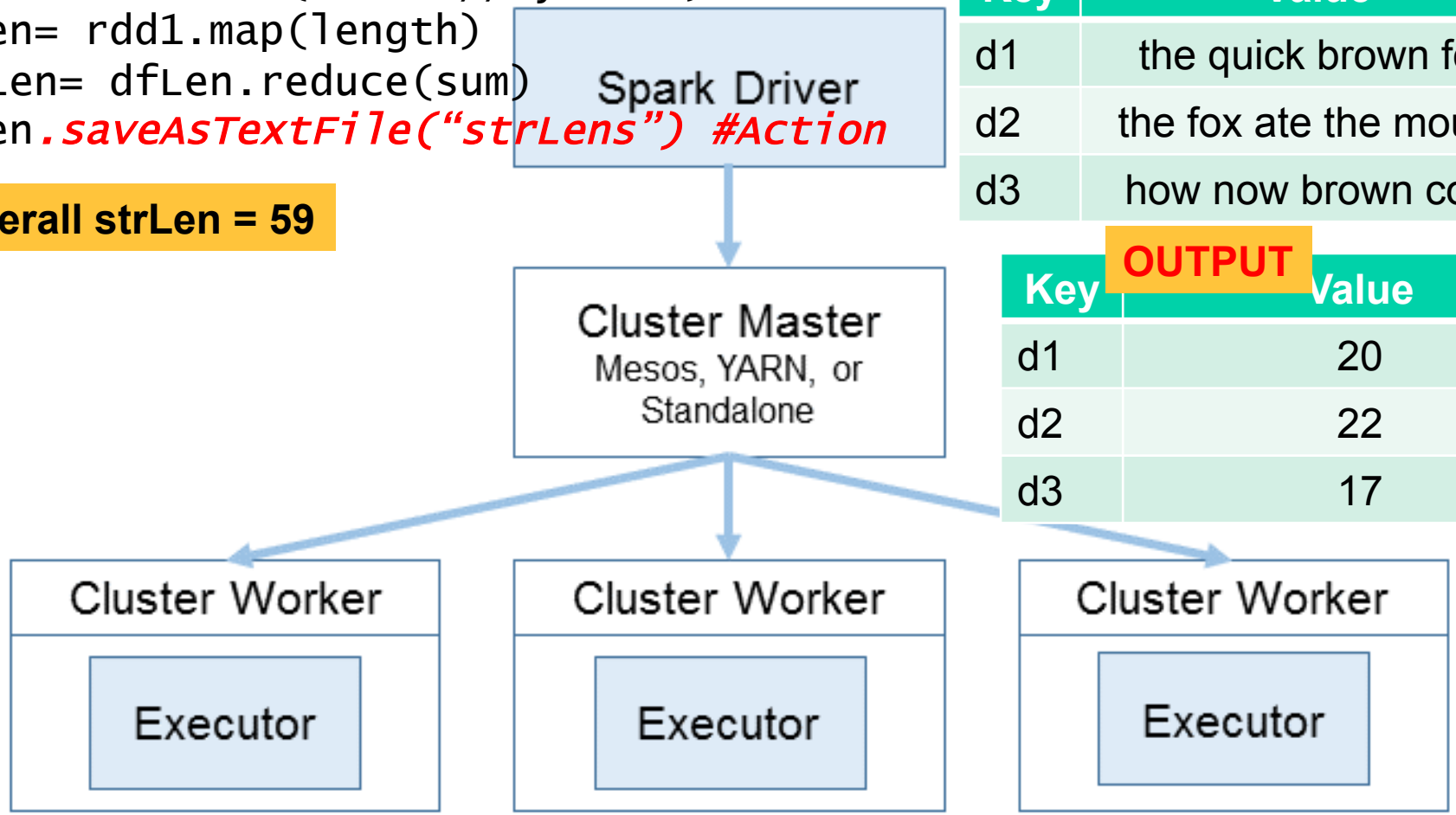
Overall strLen = 59

Spark in local mode on a single computer or on a Spark Cluster

- **Apache Spark is an open-source cluster computing framework for big data (works on TB/PB of data)**

```
df= sc.textFile("hdfs://mydocs")
dfLen= rdd1.map(length)
strLen= dfLen.reduce(sum)
dfLen.saveAsTextFile("strLens") #Action
```

Overall strLen = 59



INPUT

Key	Value
d1	the quick brown fox
d2	the fox ate the mouse
d3	how now brown cow

OUTPUT

Key	Value
d1	20
d2	22
d3	17

Life of a Spark Program

1) Create some input RDDs from external data or parallelize a collection in your driver program.

2) Lazily ***transform*** them to define new RDDs using transformations like `filter()` or `map()`

3) Ask Spark to `cache()` any intermediate RDDs that will need to be reused.

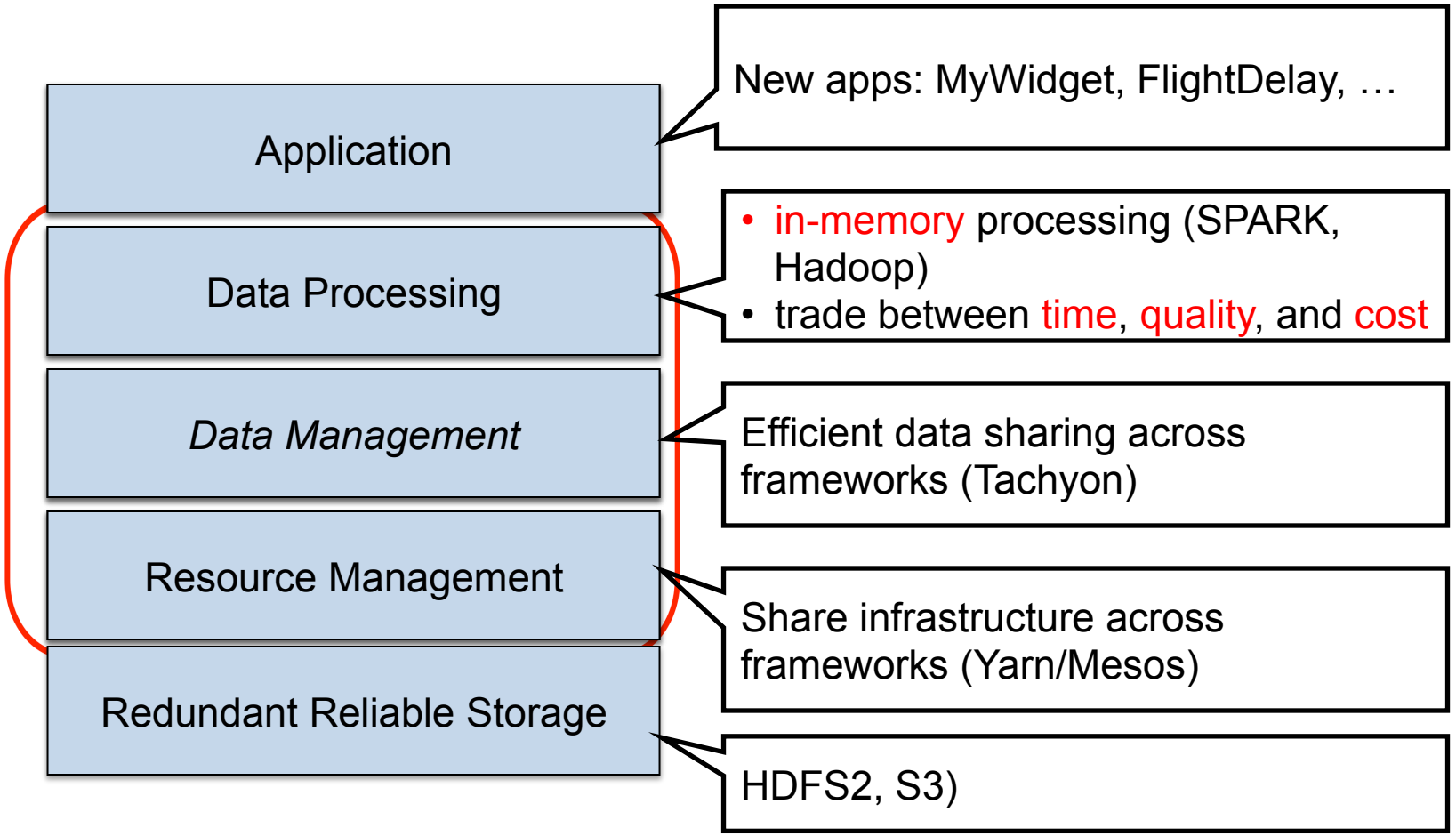
4) Launch ***actions*** such as `count()` and `collect()` to kick off a parallel computation, which is then optimized and executed by Spark.

Spark APIs

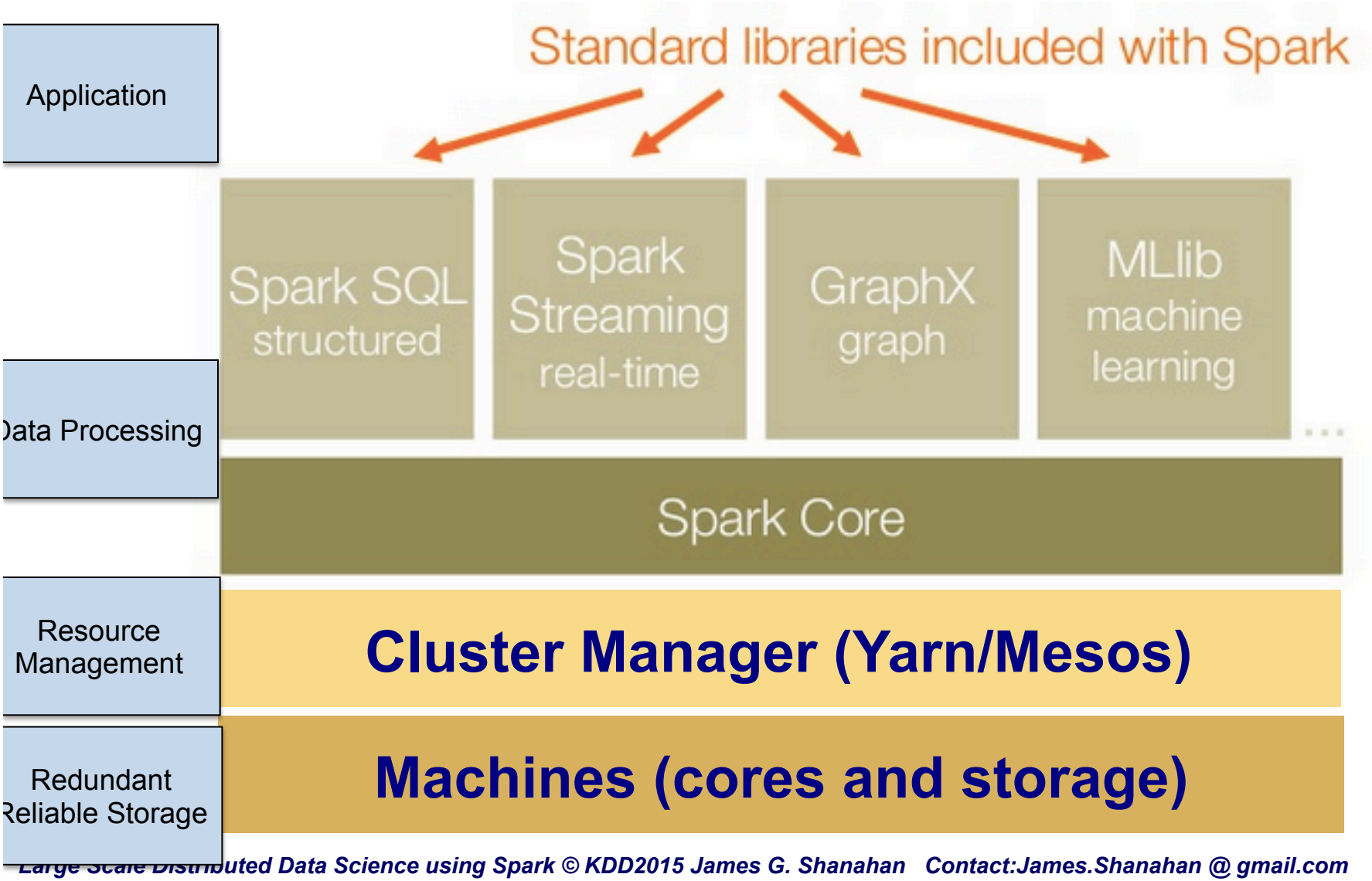
- **API**
  - In computing, a binding from a programming language to a library or operating system service is an application programming interface (API) providing glue code to use that library or service in a particular programming language.
- **Apache Spark Apache Spark is an alternative big data computing system which can run on Yarn/Mesos and provides**
  - An Elegant, Rich and Usable Core API
  - An Expansive set of ecosystem libraries built around the Core API
  - Hive compatibility via SparkSQL
  - Mature Python/R/Java/SQL/Scala support for both core APIs as well as the spark ecosystem
- **Spark has APIs for**
  - Scala, Java, Python, R, and SQL

(Ipython/Zeppelin Notebook as a Unified Data Science Interface to all of this)

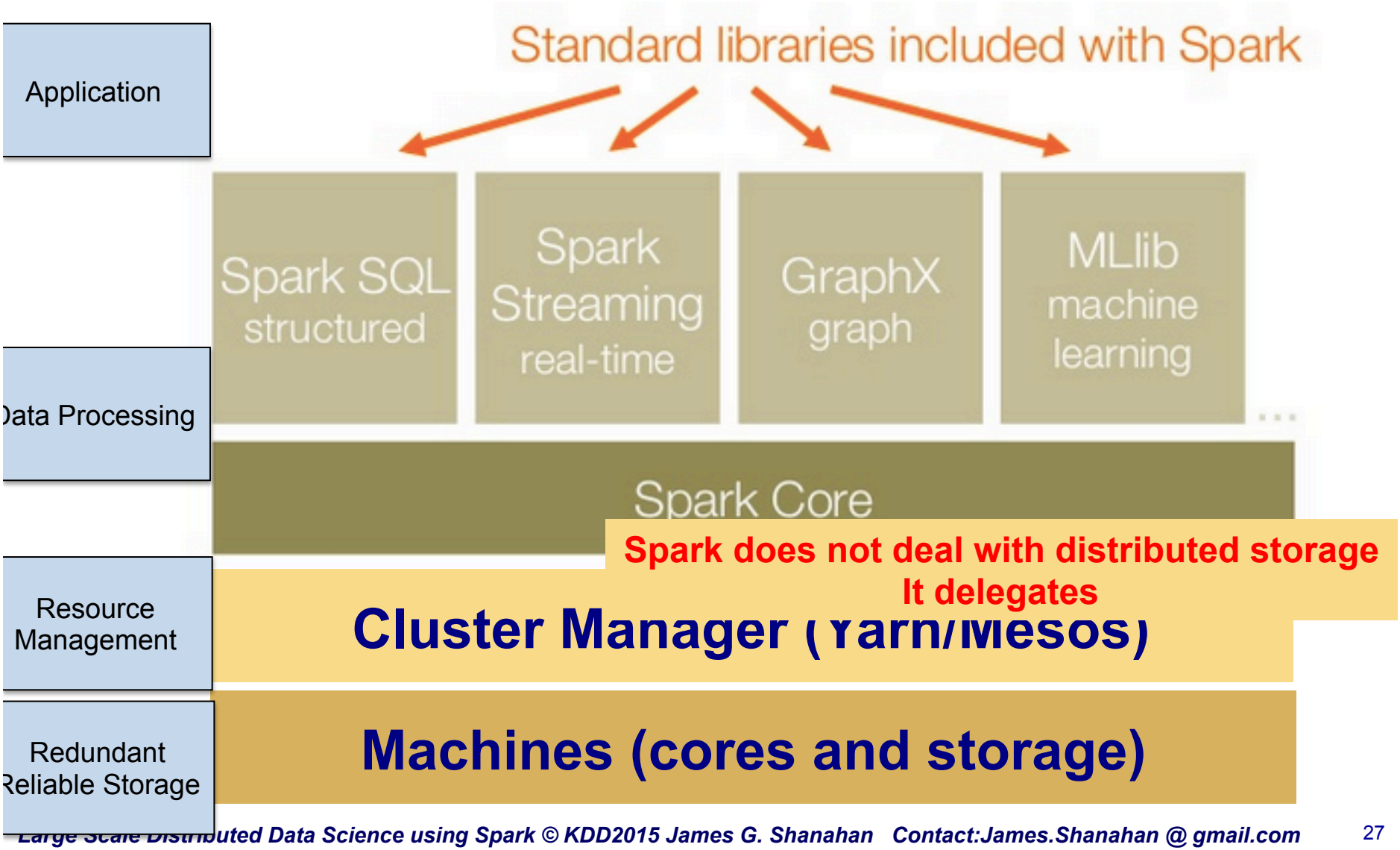
Berkeley Data Analytics Stack (BDAS)



Spark


Application
Data Processing
Resource Management
Redundant Reliable Storage
Standard libraries included with Spark
Spark SQL structured
Spark Streaming real-time
GraphX graph
MLlib machine learning
...
Spark Core
Cluster Manager (Yarn/Mesos)
Machines (cores and storage)

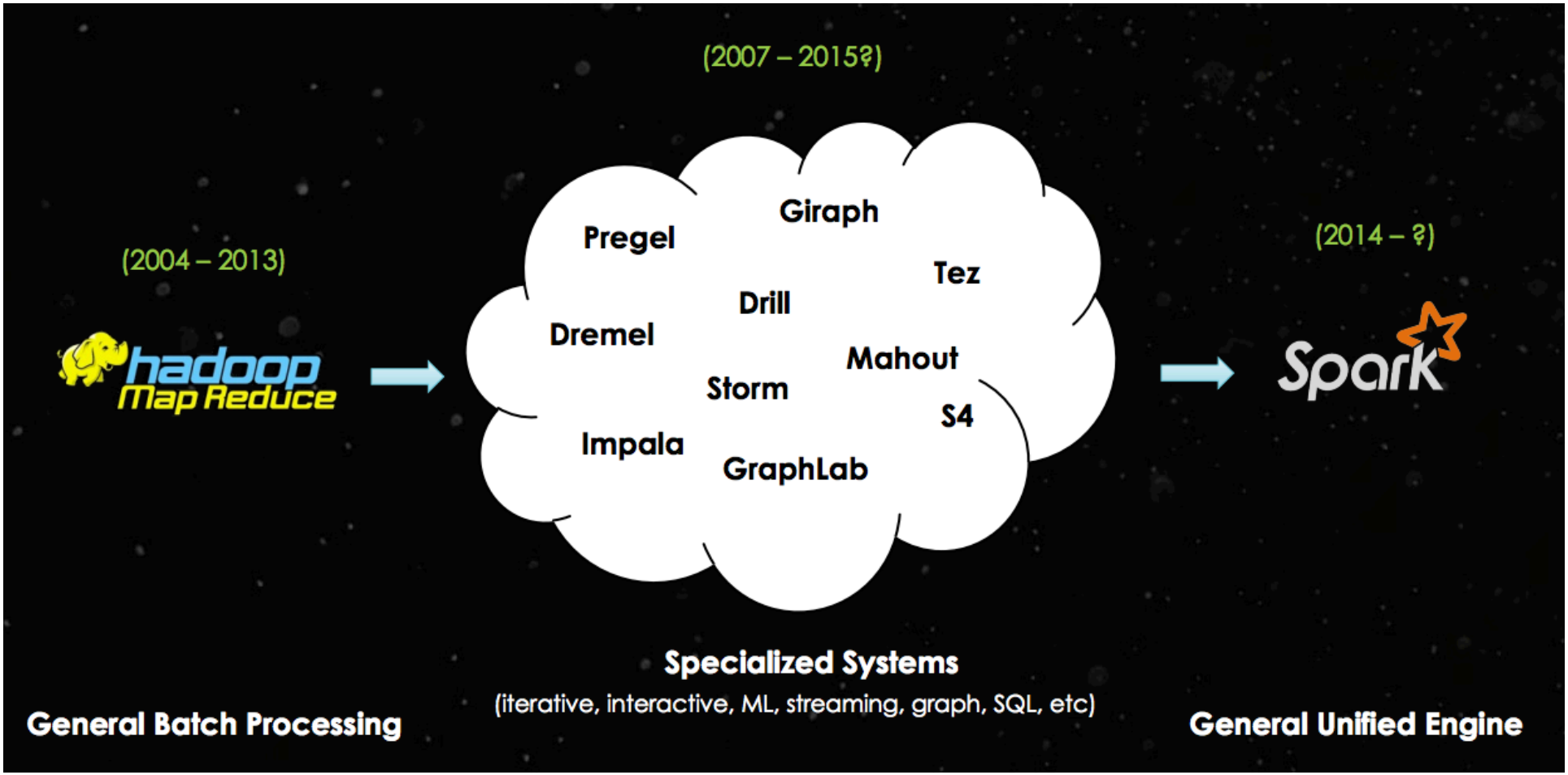
Spark


Application
Standard libraries included with Spark
Spark SQL
structured
Spark
Streaming
real-time
GraphX
graph
MLlib
machine
learning
...
Data Processing
Spark Core
Spark does not deal with distributed storage
It delegates
Resource
Management
Cluster Manager (Yarn/Mesos)
Redundant
Reliable Storage
Machines (cores and storage)

Divide and conquer with Closures

- **Decompose problems in non-overlapping sub-problems**
- **Spark's API relies heavily on passing functions in the driver program to run on the cluster. There are three recommended ways to do this:**
  - Lambda expressions, for simple functions that can be written as an expression. (Lambdas do not support multi-statement functions or statements that do not return a value.)
  - Local defs inside the function calling into Spark, for longer code.
  - Top-level functions in a module.
- **Lazy evaluation! Optimize execution graphs**

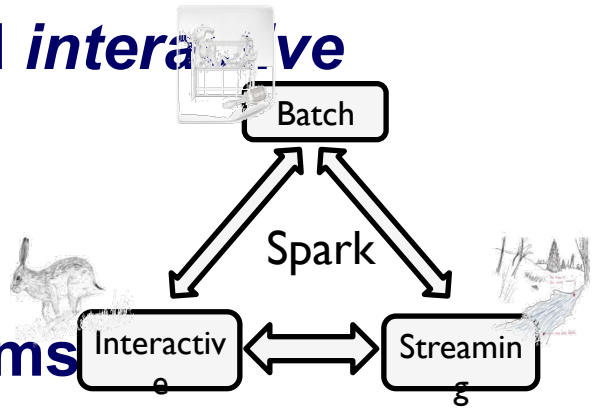
Spark builds on …



Google, Yahoo, Facebook, Twitter, MetaMarkets, DataBricks and many more

Spark Summary

- **Support *interactive* and *streaming* computations**
  - In-memory, fault-tolerant storage abstraction, low-latency scheduling,...
- ***Easy* to combine *batch*, *streaming*, and *interactive* computations**
  - Spark execution engine supports all comp. models
- ***Easy* to develop *sophisticated* algorithms**
  - Scala interface, APIs for Java, Python, R, SQL, Hive QL, …
  - New frameworks targeted to graph based and ML algorithms
- ***Compatible* with existing open source ecosystem**
- ***Open source* (Apache) and fully committed to release *high quality* software**



- **End section**

Part 1

- **Part 1: Introduction**
  - Welcome Survey
  - Install Spark
  - Background and Motivation
    - Big Data Science
    - Functional Programming
    - Poorman's Map-Reduce (to dividing and conquering)

Install the following:

Oracle Java JDK
Spark
Anaconda Python (+Jupyter notebooks)

Tutorial at WSDM 2016

http://www.wsdm-conference.org/2016/tutorials.html

Large Scale Distributed Data Science using Apache Spark

Monday, February 22, 2016
14:00 - 17:30
James Shanahan and Liang Dai
http://wsdm2016-sparktutorial.droppages.com/

Click here

Apache Spark is an open-source cluster computing framework. It has emerged as the next generation big data processing engine, overtaking Hadoop MapReduce which helped ignite the big data revolution. Spark maintains MapReduce's linear scalability and fault tolerance, but extends it in a few important ways: it is much faster (100 times faster for certain applications), much easier to program in due to its rich APIs in Python, Java, Scala and R, and its core data abstraction, the distributed data frame. In addition, it goes far beyond batch applications to support a variety of compute-intensive tasks, including interactive queries, streaming, machine learning, and graph processing.

This tutorial will provide an accessible introduction to large scale distributed machine learning and data mining, and to Spark and its potential to revolutionize academic and commercial data science practices. It is divided into two parts: the first part will cover fundamental Spark concepts, including Spark Core, data frames, the Spark Shell, Spark Streaming, Spark SQL, MLlib, and more; the second part will focus on hands-on algorithmic design and development with Spark (developing algorithms from scratch such as decision tree learning, association rule mining (aPriori), graph processing algorithms such as pagerank/shortest path, gradient descent algorithms such as support vectors machines and matrix factorization. Industrial applications and deployments of Spark will also be presented. Example code will be made available in python (pySpark) notebooks.

http://wsdm2016-sparktutorial.droppages.com/

http://www.wsdm-conference.org/2016/tutorials.html

Apps nbviewer.ipython.org Bookmarks (2) MIDS-MLS-2015- Stanford Machine Le » Other Bookmarks

Large Scale Distributed Data Science using Apache Spark

A tutorial by James Shanahan and Liang Dai

Home | Bios | Downloads | Preparation

Abstract

Apache Spark is an open-source cluster computing framework. It has emerged as the next generation big data processing engine, overtaking Hadoop MapReduce which helped ignite the big data revolution. Spark maintains MapReduce's linear scalability and fault tolerance, but extends it in a few important ways: it is much faster (100 times faster for certain applications), much easier to program in due to its rich APIs in Python, Java, Scala (and R), and its core data abstraction, the distributed data frame, and it goes far beyond batch applications to support a variety of compute-intensive tasks, including interactive queries, streaming, machine learning, and graph processing.

This tutorial will provide an accessible introduction to those not already familiar with Spark and its potential to revolutionize academic and commercial data science practices. It is divided into two parts: the first part will introduce fundamental Spark concepts, including

CONTACT US

emails:

james.shanahan_at_gmail.com

liangdai16_at_gmail.com

HOME

RELATED LINKS

Large Scale Distributed Data Science using Apache Spark

A tutorial by James Shanahan and Liang Dai

Home | Bios | Downloads | Preparation

Preparation Tab

Step 1: Install JDK

Because Apache spark depends on Java run time environment, you need to have JDK installed in your machine. Download

Java SE Development Kit 8u51

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.

Product / File Description	File Size	Download
Linux x86	146.9 MB	jdk-8u51-linux-i586.rpm
Linux x86	166.95 MB	jdk-8u51-linux-i586.tar.gz
Linux x64	145.19 MB	jdk-8u51-linux-x64.rpm
Linux x64	165.25 MB	jdk-8u51-linux-x64.tar.gz
Mac OS X x64	222.09 MB	jdk-8u51-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	139.36 MB	jdk-8u51-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	98.8 MB	jdk-8u51-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	139.79 MB	jdk-8u51-solaris-x64.tar.Z

CONTACT US

emails:

james.shanahan_at_gmail.com

liangdai16_at_gmail.com

PREPARATION

RELATED LINKS

CIKM 2015
Apache Spark

Installing Hadoop

- **Windows:**
  - Hadoop 1.0
  - http://saphanatutorial.com/hadoop-installation-on-windows-7-using-cygwin/
  - Hadoop 2.0 (Hortonworks Data Platform 2.0 for Windows)
  - http://hortonworks.com/blog/install-hadoop-windows-hortonworks-data-platform-2-0/
- **Mac:**
  - http://amodernstory.com/2014/09/23/installing-hadoop-on-mac-osx-yosemite/
  - This link is for haddoop 2.6. I follow the instructions and easily get hadoop installed.
- **Linux (Ubuntu):**
  - http://www.bogotobogo.com/Hadoop/BigData_hadoop_Install_on_ubuntu_single_node_cluster.php

On Mac install HomeBrew

- **Install HomeBrew**
  - Download it from the website at http://brew.sh/ or simply paste the script inside the terminal
  - $ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

On Windows install CygWin

- **Cygwin is:**
  - a large collection of GNU and Open Source tools which provide functionality similar to a Linux distribution on Windows.

Current Cygwin DLL version

The most recent version of the Cygwin DLL is 2.2.1. Install it by running setup-x86.exe (32-bit installation) or setup-x86_64.exe (64-bit installation).

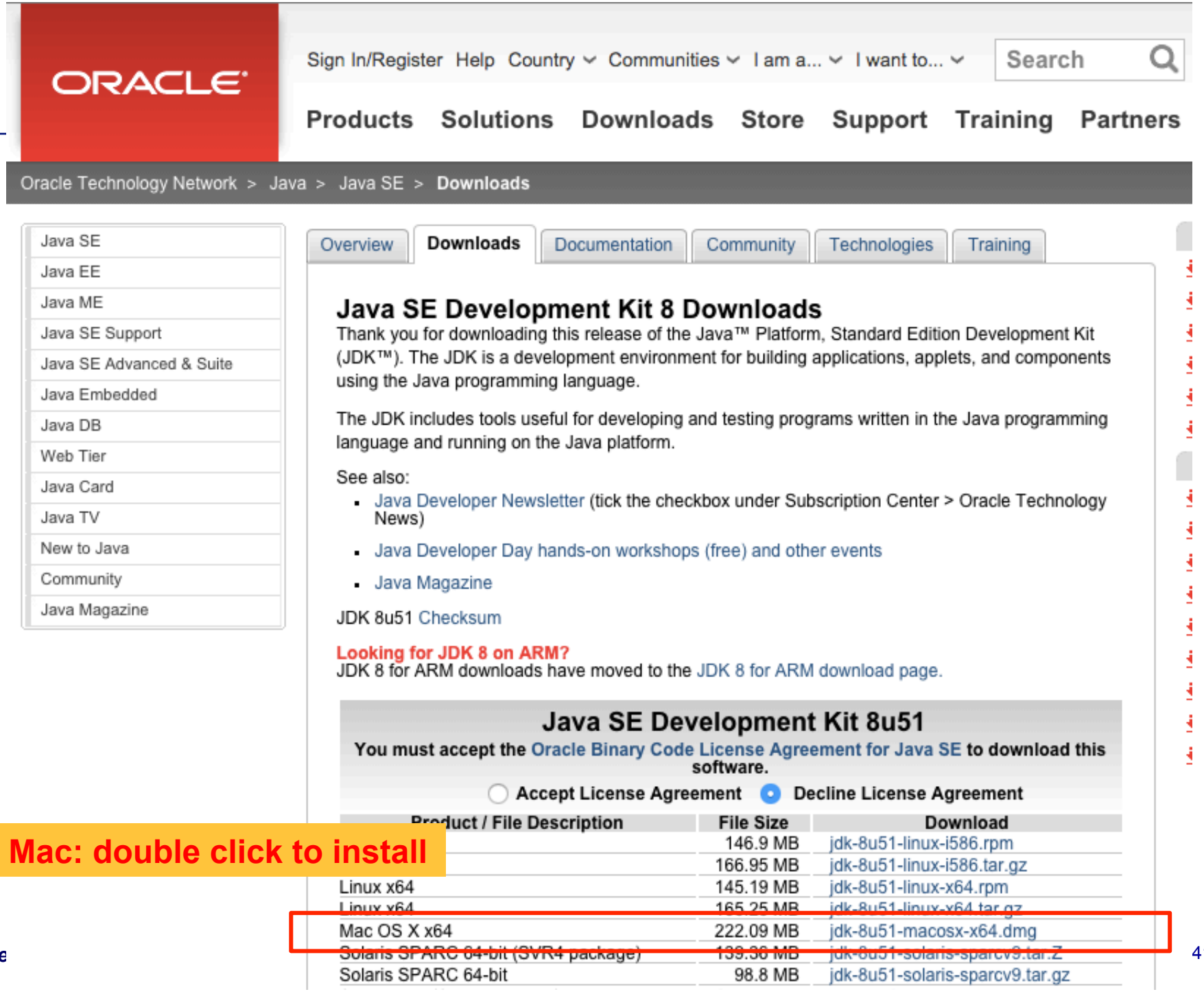
Use the setup program to perform a fresh install or to update an existing installation.

Note that individual packages in the distribution are updated separately from the DLL so the Cygwin DLL version is not useful as a general Cygwin release number.

Make sure Java JDK is installed

- **Click here:**
  http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

- **On windows machine set up $JAVA_HOME**
  - Right-click the My Computer icon on your desktop and select Properties
  - Click the Advanced tab
  - Click the Environment Variables button
  - Under System Variables, click New
  - Enter the variable name as JAVA_HOME
  - Enter the variable value as the installation path for the Java Development Kit



On Mac: double click to install

Anaconda Install

Download python+Notebook
Please ignore this if you have Jupter installed already

http://docs.continuum.io/anaconda/install

Contents

Download the Anaconda installer and double click it.

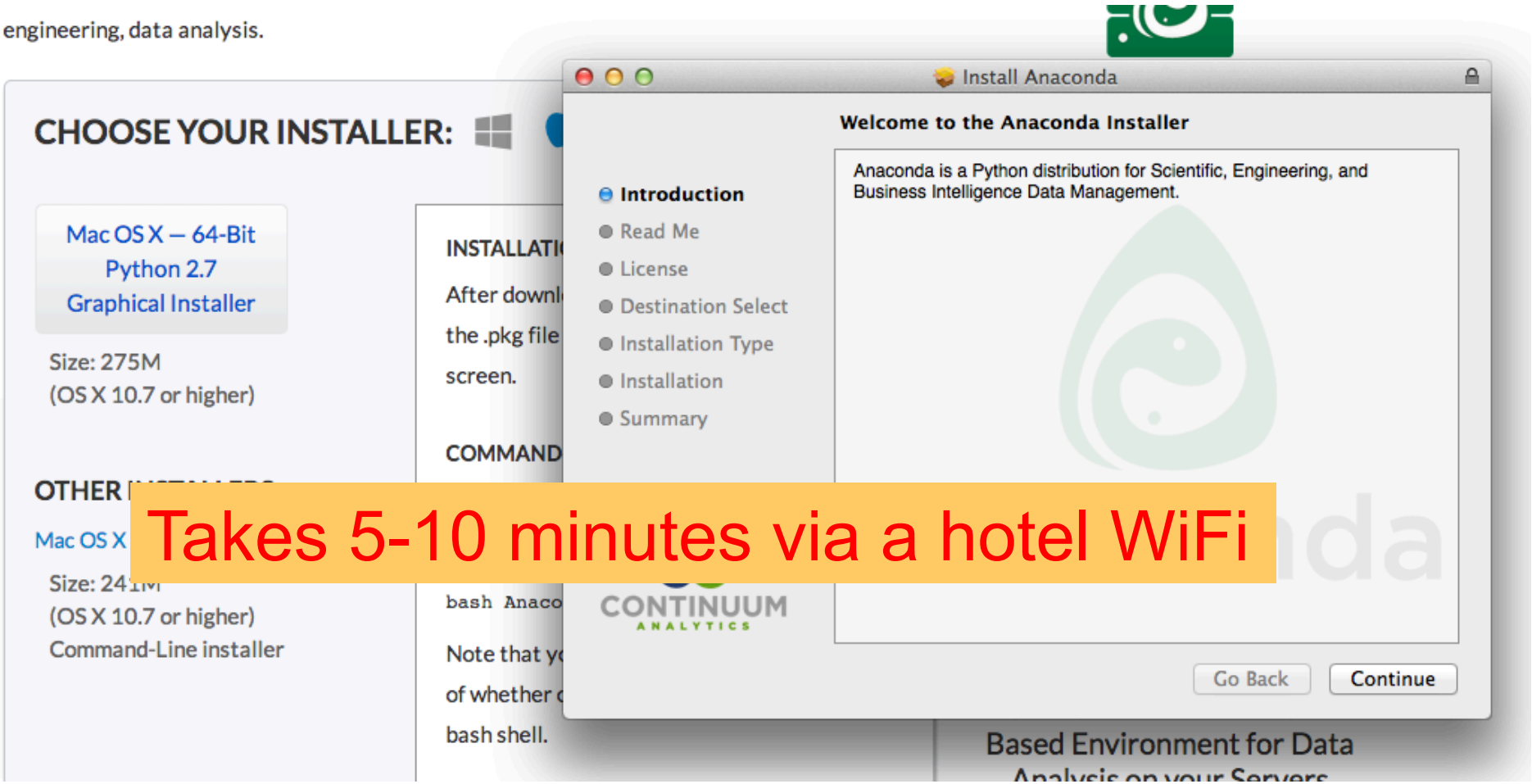
OS X Install

Download the Anaconda installer and double click it.

NOTE: You may see a screen that says "You cannot install Anaconda in this location. The Anaconda

Takes 5-10 minutes via a hotel WiFi

Anaconda Installer



conda update --prefix /Users/jshanahan/anaconda anaconda

conda update --prefix /Users/jshanahan/anaconda anaconda

To start the Notebook

/Users/jshanahan/anaconda/bin/ipython notebook&

- **/Users/jshanahan/anaconda/bin/ipython notebook&**

Apache Spark Download

- **Install 1.4.1 (or latest version) by clicking here**
  - https://dl.dropboxusercontent.com/u/27377155/spark-1.4.1-bin-hadoop2.6.tgz
- **For other versions click here**
  - http://spark.apache.org/downloads.html
  - http://spark.apache.org/docs/latest/programming-guide.html

Download latest version of Spark

- ..

Lightning-fast cluster computing

Download | Libraries | Documentation | Examples | Community | FAQ

Get latest version Spark 1.6.2 as of June, 2016

Download Spark

The latest release of Spark is Spark 1.5.1, released on October 2, 2015 (release notes) (git tag)

Step1 — 1. Choose a Spark release: 1.5.1 (Oct 02 2015)

Step2 — 2. Choose a package type: Pre-built for Hadoop 2.6 and later

3. Choose a download type: Select Apache Mirror

Step4 — 4. Download Spark: spark-1.5.1-bin-hadoop2.6.tgz

5. Verify this release using the 1.5.1 signatures and checksums.

Note: Scala 2.11 users should download the Spark source package and build with Scala 2.11 support.

Untar Spark; start pyspark interpreter

```
tar -xzvf spark-1.4.1-bin-hadoop2.6.tgz
cd spark-1.4.1-bin-hadoop2.6
./bin/pyspark
 #if everything goes well, you can see the PySpark interactive shell
./bin/pyspark
```

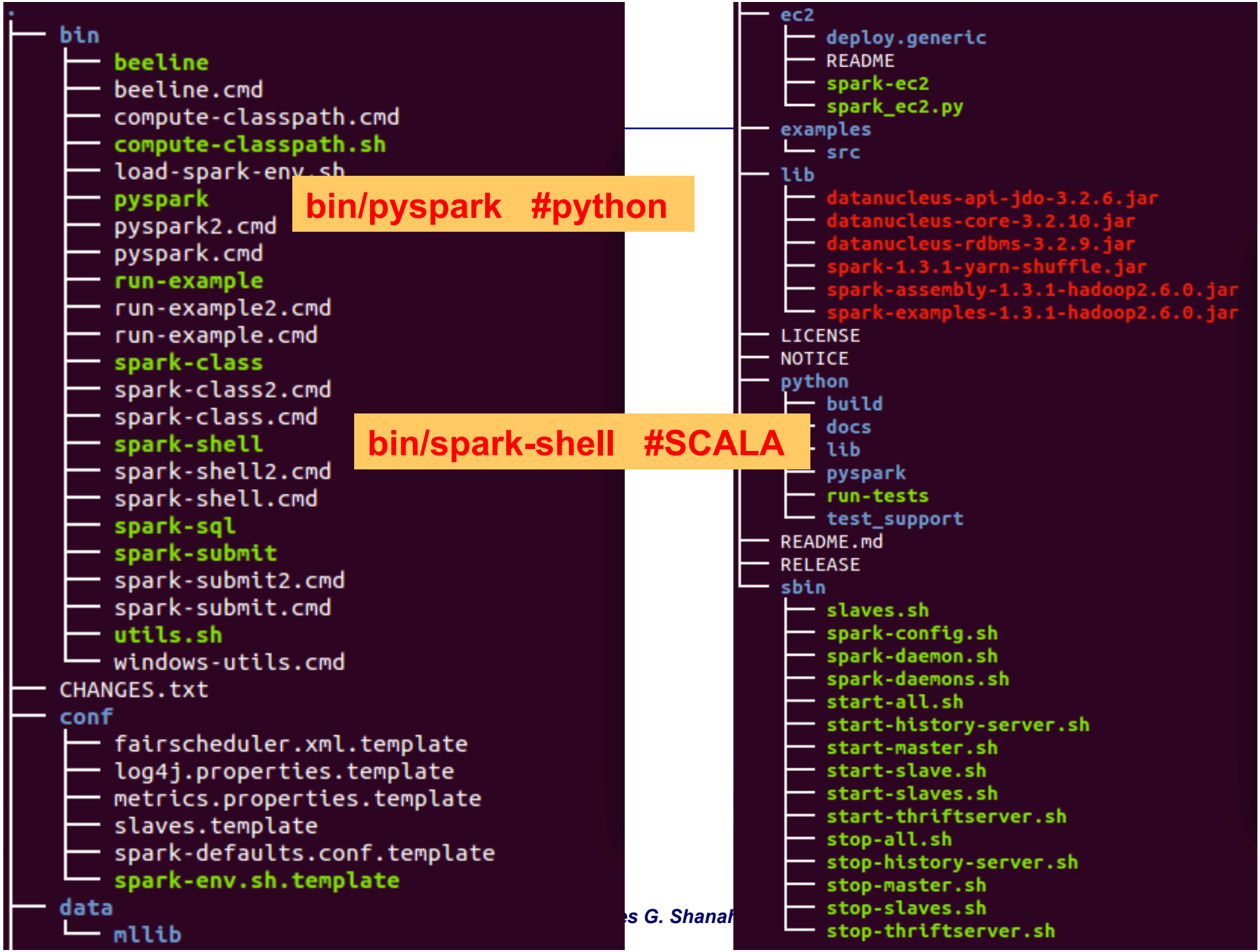
```
JAMES-SHANAHANs-Desktop-Pro:Software jshanahan$ ls spark-1.2.1-bin-hadoop2.4
LICENSE         README.md       bin             data            examples        python
NOTICE          RELEASE         conf            ec2             lib             sbin
JAMES-SHANAHANs-Desktop-Pro:Software jshanahan$ spark-1.2.1-bin-hadoop2.4/bin/pyspark
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr  9 2012, 20:52:43)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Spark assembly has been built with Hive, including Datanucleus jars on classpath
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
15/02/11 16:57:02 INFO SecurityManager: Changing view acls to: jshanahan
15/02/11 16:57:02 INFO SecurityManager: Changing modify acls to: jshanahan
```

Untar the Spark; start pyspark interpreter

- tar -xzvf spark-1.2.1-bin-hadoop2.4.tgz
- #Start pyspark
- spark-1.2.1-bin-hadoop2.4/bin/pyspark

```
JAMES-SHANAHANs-Desktop-Pro:Software jshanahan$ ls spark-1.2.1-bin-hadoop2.4
LICENSE         README.md       bin             data            examples        python
NOTICE          RELEASE         conf            ec2             lib             sbin
JAMES-SHANAHANs-Desktop-Pro:Software jshanahan$ spark-1.2.1-bin-hadoop2.4/bin/pyspark
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr  9 2012, 20:52:43)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Spark assembly has been built with Hive, including Datanucleus jars on classpath
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
15/02/11 16:57:02 INFO SecurityManager: Changing view acls to: jshanahan
15/02/11 16:57:02 INFO SecurityManager: Changing modify acls to: jshanahan
15/02/11 16:57:02 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; u
w permissions: Set(jshanahan); users with modify permissions: Set(jshanahan)
15/02/11 16:57:02 INFO Slf4jLogger: Slf4jLogger started
15/02/11 16:57:02 INFO Remoting: Starting remoting
15/02/11 16:57:02 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkDriver@10
]
15/02/11 16:57:02 INFO Utils: Successfully started service 'sparkDriver' on port 54079.
15/02/11 16:57:02 INFO SparkEnv: Registering MapOutputTracker
15/02/11 16:57:02 INFO SparkEnv: Registering BlockManagerMaster
15/02/11 16:57:02 INFO DiskBlockManager: Created local directory at /var/folders/j4/95k348x940xcz40fk
/T/spark-5081f827-8087-436d-8cc2-bb05a24410a3/spark-c4be330b-173d-4f3f-8659-2a919c2f7bab
15/02/11 16:57:02 INFO MemoryStore: MemoryStore started with capacity 273.0 MB
2015-02-11 16:57:03.065 java[44783:ea03] Unable to load realm info from SCDynamicStore
15/02/11 16:57:03 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... us
ava classes where applicable
15/02/11 16:57:03 INFO HttpFileServer: HTTP File server directory is /var/folders/j4/95k348x940xcz40f
n/T/spark-d6caef39-c3bc-48d3-8a9e-f4b24e3e4a21/spark-1c7165c4-a18d-4f23-86ae-5d92519ff345
15/02/11 16:57:03 INFO HttpServer: Starting HTTP Server
15/02/11 16:57:03 INFO Utils: Successfully started service 'HTTP file server' on port 54080.
15/02/11 16:57:03 INFO Utils: Successfully started service 'SparkUI' on port 4040.
15/02/11 16:57:03 INFO SparkUI: Started SparkUI at http://10.0.0.93:4040
15/02/11 16:57:03 INFO Executor: Starting executor ID <driver> on host localhost
15/02/11 16:57:03 INFO AkkaUtils: Connecting to HeartbeatReceiver: akka.tcp://sparkDriver@10.0.0.93:5
rtbeatReceiver
15/02/11 16:57:03 INFO NettyBlockTransferService: Server created on 54081
15/02/11 16:57:03 INFO BlockManagerMaster: Trying to register BlockManager
15/02/11 16:57:03 INFO BlockManagerMasterActor: Registering block manager localhost:54081 with 273.0 I
ManagerId(<driver>, localhost, 54081)
15/02/11 16:57:03 INFO BlockManagerMaster: Registered BlockManager
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.2.1
      /_/

Using Python version 2.7.3 (v2.7.3:70274d53c1dd, Apr  9 2012 20:52:43)
SparkContext available as sc.
>>>
```


bin
beeline
beeline.cmd
compute-classpath.cmd
compute-classpath.sh
load-spark-env.sh
pyspark
pyspark2.cmd
pyspark.cmd
run-example
run-example2.cmd
run-example.cmd
spark-class
spark-class2.cmd
spark-class.cmd
spark-shell
spark-shell2.cmd
spark-shell.cmd
spark-sql
spark-submit
spark-submit2.cmd
spark-submit.cmd
utils.sh
windows-utils.cmd
CHANGES.txt
conf
fairscheduler.xml.template
log4j.properties.template
metrics.properties.template
slaves.template
spark-defaults.conf.template
spark-env.sh.template
data
mllib
bin/pyspark #python
bin/spark-shell #SCALA
ec2
deploy.generic
README
spark-ec2
spark_ec2.py
examples
src
lib
datanucleus-api-jdo-3.2.6.jar
datanucleus-core-3.2.10.jar
datanucleus-rdbms-3.2.9.jar
spark-1.3.1-yarn-shuffle.jar
spark-assembly-1.3.1-hadoop2.6.0.jar
spark-examples-1.3.1-hadoop2.6.0.jar
LICENSE
NOTICE
python
build
docs
lib
pyspark
run-tests
test_support
README.md
RELEASE
sbin
slaves.sh
spark-config.sh
spark-daemon.sh
spark-daemons.sh
start-all.sh
start-history-server.sh
start-master.sh
start-slave.sh
start-slaves.sh
start-thriftserver.sh
stop-all.sh
stop-history-server.sh
stop-master.sh
stop-slaves.sh
stop-thriftserver.sh

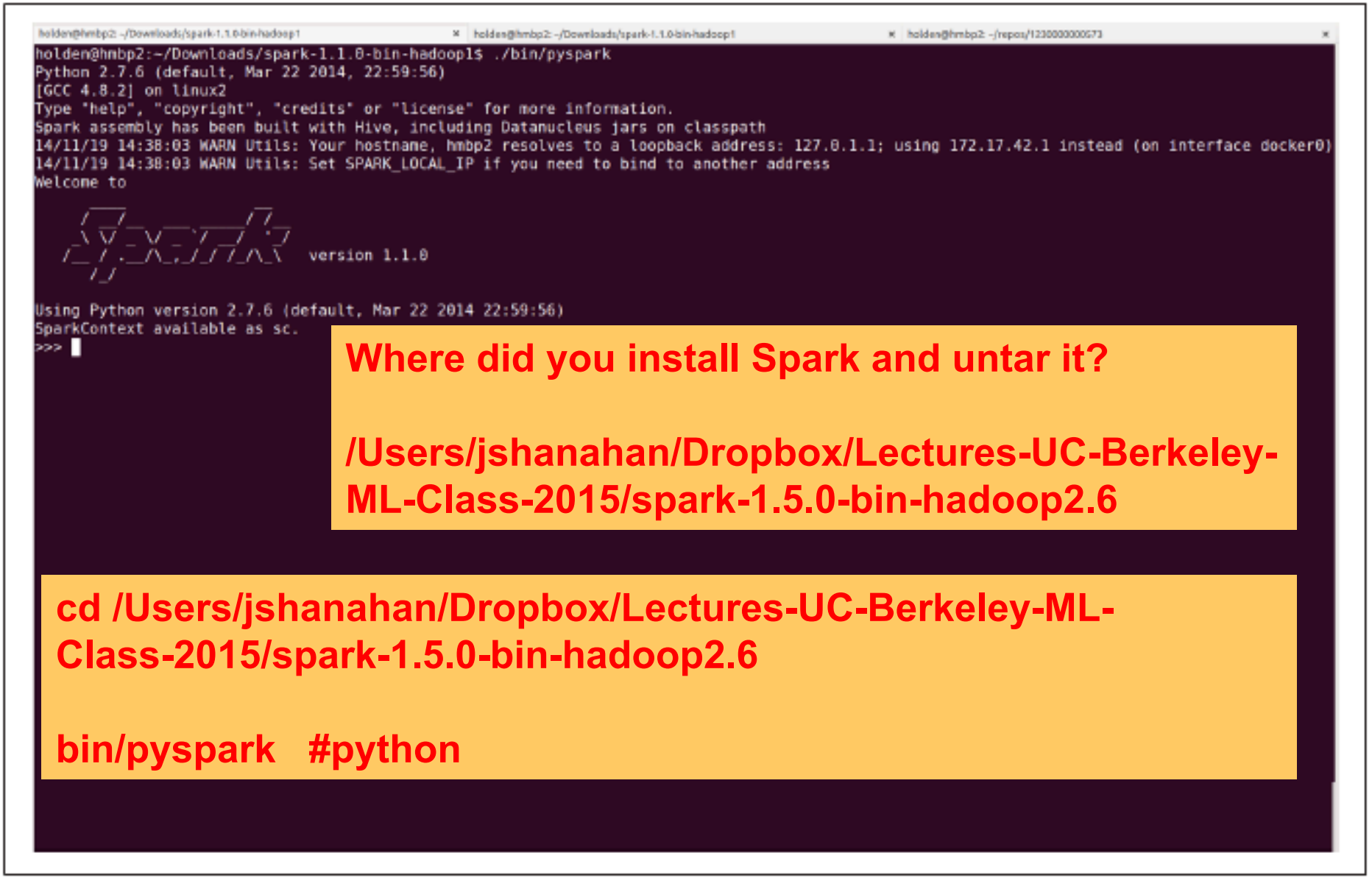


Figure 2-2. The PySpark shell with less logging output

bin/sparkR

```
JAMES-SHANAHANs-Desktop-Pro:KDD-Notebooks jshanahan$ cd /Users/jshanahan/Dropbox/Lectures-UC-Berkeley-ML-Class-2015/spark-1.4.0-bin-hadoop2.6
JAMES-SHANAHANs-Desktop-Pro:spark-1.4.0-bin-hadoop2.6 jshanahan$ bin/sparkR
```

Where did you install Spark and untar it?

/Users/jshanahan/Dropbox/Lectures-UC-Berkeley-ML-Class-2015/spark-1.5.0-bin-hadoop2.6

#To run sparkR
/Users/jshanahan/Dropbox/Lectures-UC-Berkeley-ML-Class-2015/spark-1.5.0-bin-hadoop2.6/bin/sparkR

```
Launching java with spark-submit command /Users/jshanahan/Dropbox/Lectures-UC-Berkeley-ML-Class-2015/spark-1.4.0-bin-hadoop2.6/bin/spark-submit  "sparkr-shell" /var/folders/j4/95k348x940xcz40fkdmgy_n40000gn/T//RtmpjQpRRC/backend_port14725284a4ad
log4j:WARN No appenders could be found for logger (io.netty.util.internal.logging.InternalLoggerFactory).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
15/08/10 13:52:17 INFO SparkContext: Running Spark version 1.4.0
15/08/10 13:52:19 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
15/08/10 13:52:19 INFO SecurityManager: Changing view acls to: jshanahan
15/08/10 13:52:19 INFO SecurityManager: Changing modify acls to: jshanahan
15/08/10 13:52:19 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(jshanahan); users with modify permissions: Set(jshanahan)
```

Commands in R: iris data

- ?iris
- ?data

iris package:datasets R Documentation

Edgar Anderson's Iris Data

Description:

This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are _Iris setosa_, _versicolor_, and _virginica_.

Usage:

```
iris
iris3
```

Format:

'iris' is a data frame with 150 cases (rows) and 5 variables (columns) named 'Sepal.Length', 'Sepal.Width', 'Petal.Length', 'Petal.Width', and 'Species'.

'iris3' gives the same data arranged as a 3-dimensional array of size 50 by 4 by 3, as represented by S-PLUS. The first dimension gives the case number within the species subsample, the second the measurements with names 'Sepal L.', 'Sepal W.', 'Petal L.', and 'Petal W.', and the third the species.

```
> head(iris)
  Sepal.Length Sepal.Width Pe
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

```
Data sets in package 'datasets':

AirPassengers           Monthly Airline Passenger Numbers 1949-1960
BJsales                 Sales Data with Leading Indicator
BJsales.lead (BJsales)
                        Sales Data with Leading Indicator
BOD                     Biochemical Oxygen Demand
CO2                     Carbon Dioxide Uptake in Grass Plants
ChickWeight             Weight versus age of chicks on different diets
DNase                   Elisa assay of DNase
EuStockMarkets          Daily Closing Prices of Major European Stock
                        Indices, 1991-1998
Formaldehyde            Determination of Formaldehyde
HairEyeColor            Hair and Eye Color of Statistics Students
Harman23.cor            Harman Example 2.3
Harman74.cor            Harman Example 7.4
Indometh                Pharmacokinetics of Indomethacin
InsectSprays            Effectiveness of Insect Sprays
JohnsonJohnson          Quarterly Earnings per Johnson & Johnson Share
LakeHuron               Level of Lake Huron 1875-1972
LifeCycleSavings        Intercountry Life-Cycle Savings Data
Loblolly                Growth of Loblolly pine trees
Nile                    Flow of the River Nile
Orange                  Growth of Orange Trees
OrchardSprays           Potency of Orchard Sprays
PlantGrowth             Results from an Experiment on Plant Growth
Puromycin               Reaction Velocity of an Enzymatic Reaction
Seatbelts               Road Casualties in Great Britain 1969-84
Theoph                  Pharmacokinetics of Theophylline
Titanic                 Survival of passengers on the Titanic
ToothGrowth             The Effect of Vitamin C on Tooth Growth in
                        Guinea Pigs
UCBAdmissions           Student Admissions at UC Berkeley
UKDriverDeaths          Road Casualties in Great Britain 1969-84
UKgas                   UK Quarterly Gas Consumption
USAccDeaths             Accidental Deaths in the US 1973-1978
USArrests               Violent Crime Rates by US State
USJudgeRatings          Lawyers' Ratings of State Judges in the US
                        Superior Court
USPersonalExpenditure   Personal Expenditure Data
VADeaths                Death Rates in Virginia (1940)
WWWusage                Internet Usage per Minute
WorldPhones             The World's Telephones
ability.cov             Ability and Intelligence Tests
airmiles                Passenger Miles on Commercial US Airlines,
                        1937-1960
:
```

>data() #lists all datasets

```
airquality              New York Air Quality Measurements
anscombe                Anscombe's Quartet of 'Identical' S
                        Regressions
attenu                  The Joyner-Boore Attenuation Data
attitude                The Chatterjee-Price Attitude Data
austres                 Quarterly Time Series of the Number
                        Australian Residents
beaver1 (beavers)       Body Temperature Series of Two Beav
beaver2 (beavers)       Body Temperature Series of Two Beav
cars                    Speed and Stopping Distances of Car
chickwts                Chicken Weights by Feed Type
co2                     Mauna Loa Atmospheric CO2 Concentra
crimtab                 Student's 3000 Criminals Data
discoveries             Yearly Numbers of Important Discove
esoph                   Smoking, Alcohol and (O)esophageal
euro                    Conversion Rates of Euro Currencies
euro.cross (euro)       Conversion Rates of Euro Currencies
eurodist                Distances Between European Cities
faithful                Old Faithful Geyser Data
fdeaths (UKLungDeaths)
                        Monthly Deaths from Lung Diseases i
freeny                  Freeny's Revenue Data
freeny.x (freeny)       Freeny's Revenue Data
freeny.y (freeny)       Freeny's Revenue Data
infert                  Infertility after Spontaneous and I
                        Abortion
iris                    Edgar Anderson's Iris Data
iris3                   Edgar Anderson's Iris Data
islands                 Areas of the World's Major Landmass
ldeaths (UKLungDeaths)
                        Monthly Deaths from Lung Diseases i
lh                      Luteinizing Hormone in Blood Sample
longley                 Longley's Economic Regression Data
lynx                    Annual Canadian Lynx trappings 1821
mdeaths (UKLungDeaths)
                        Monthly Deaths from Lung Diseases i
morley                  Michelson Speed of Light Data
mtcars                  Motor Trend Car Road Tests
nhtemp                  Average Yearly Temperatures in New
nottem                  Average Monthly Temperatures at Not
                        1920-1939
npk                     Classical N, P, K Factorial Experim
occupationalStatus      Occupational Status of Fathers and
precip                  Annual Precipitation in US Cities
presidents              Quarterly Approval Ratings of US Pr
pressure                Vapor Pressure of Mercury as a Func
:
```

```
morley                  Michelson Speed of Light Data
mtcars                  Motor Trend Car Road Tests
nhtemp                  Average Yearly Temperatures in New Haven
nottem                  Average Monthly Temperatures at Nottingham,
                        1920-1939
npk                     Classical N, P, K Factorial Experiment
occupationalStatus      Occupational Status of Fathers and their Sons
precip                  Annual Precipitation in US Cities
presidents              Quarterly Approval Ratings of US Presidents
pressure                Vapor Pressure of Mercury as a Function of
                        Temperature
quakes                  Locations of Earthquakes off Fiji
randu                   Random Numbers from Congruential Generator
                        RANDU
rivers                  Lengths of Major North American Rivers
rock                    Measurements on Petroleum Rock Samples
sleep                   Student's Sleep Data
stack.loss (stackloss)
                        Brownlee's Stack Loss Plant Data
stack.x (stackloss)     Brownlee's Stack Loss Plant Data
stackloss               Brownlee's Stack Loss Plant Data
state.abb (state)       US State Facts and Figures
state.area (state)      US State Facts and Figures
state.center (state)    US State Facts and Figures
state.division (state)
                        US State Facts and Figures
state.name (state)      US State Facts and Figures
state.region (state)    US State Facts and Figures
state.x77 (state)       US State Facts and Figures
sunspot.month           Monthly Sunspot Data, from 1749 to "Present"
sunspot.year            Yearly Sunspot Data, 1700-1988
sunspots                Monthly Sunspot Numbers, 1749-1983
swiss                   Swiss Fertility and Socioeconomic Indicators
                        (1888) Data
treering                Yearly Treering Data, -6000-1979
trees                   Girth, Height and Volume for Black Cherry Trees
uspop                   Populations Recorded by the US Census
volcano                 Topographic Information on Auckland's Maunga
                        Whau Volcano
warpbreaks              The Number of Breaks in Yarn during Weaving
women                   Average Heights and Weights for American Women

Use 'data(package = .packages(all.available = TRUE))'
to list the data sets in all *available* packages.

(END)
```

Iris data check

- df = createDataFrame(sqlContext, iris)
- head(df)

```
> df
DataFrame[Sepal_Length:double, Sepal_Width:double, Petal_Length:double, Petal_Width:double, Species:string]
> head(df)
15/08/10 13:52:51 INFO SparkContext: Starting job: dfToCols at NativeMethodAccessorImpl.java:-2
15/08/10 13:52:51 INFO DAGScheduler: Got job 1 (dfToCols at NativeMethodAccessorImpl.java:-2) with 1 output partitions (allowLoc
al=false)
15/08/10 13:52:51 INFO DAGScheduler: Final stage: ResultStage 1(dfToCols at NativeMethodAccessorImpl.java:-2)
15/08/10 13:52:51 INFO DAGScheduler: Parents of final stage: List()
15/08/10 13:52:51 INFO DAGScheduler: Missing parents: List()
15/08/10 13:52:51 INFO DAGScheduler: Submitting ResultStage 1 (MapPartitionsRDD[4] at dfToCols at NativeMethodAccessorImpl.java:
-2), which has no missing parents
15/08/10 13:52:51 INFO MemoryStore: ensureFreeSpace(8776) called with curMem=2134, maxMem=278019440
15/08/10 13:52:51 INFO MemoryStore: Block broadcast_1 stored as values in memory (estimated size 8.6 KB, free 265.1 MB)
15/08/10 13:52:51 INFO MemoryStore: ensureFreeSpace(3621) called with curMem=10910, maxMem=278019440
15/08/10 13:52:51 INFO MemoryStore: Block broadcast_1_piece0 stored as bytes in memory (estimated size 3.5 KB, free 265.1 MB)
15/08/10 13:52:51 INFO BlockManagerInfo: Added broadcast_1_piece0 in memory on localhost:54179 (size: 3.5 KB, free: 265.1 MB)
15/08/10 13:52:51 INFO SparkContext: Created broadcast 1 from broadcast at DAGScheduler.scala:874
15/08/10 13:52:51 INFO DAGScheduler: Submitting 1 missing tasks from ResultStage 1 (MapPartitionsRDD[4] at dfToCols at NativeMet
hodAccessorImpl.java:-2)
15/08/10 13:52:51 INFO TaskSchedulerImpl: Adding task set 1.0 with 1 tasks
15/08/10 13:52:51 INFO TaskSetManager: Starting task 0.0 in stage 1.0 (TID 1, localhost, PROCESS_LOCAL, 15837 bytes)
15/08/10 13:52:51 INFO Executor: Running task 0.0 in stage 1.0 (TID 1)
15/08/10 13:52:52 INFO Executor: Finished task 0.0 in stage 1.0 (TID 1). 2502 bytes result sent to driver
15/08/10 13:52:52 INFO TaskSetManager: Finished task 0.0 in stage 1.0 (TID 1) in 637 ms on localhost (1/1)
15/08/10 13:52:52 INFO DAGScheduler: ResultStage 1 (dfToCols at NativeMethodAccessorImpl.java:-2) finished in 0.637 s
15/08/10 13:52:52 INFO TaskSchedulerImpl: Removed TaskSet 1.0, whose tasks have all completed, from pool
15/08/10 13:52:52 INFO DAGScheduler: Job 1 finished: dfToCols at NativeMethodAccessorImpl.java:-2, took 0.654923 s
  Sepal_Length Sepal_Width Petal_Length Petal_Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
```

Wordcount Notebook

- **Download the following notebook**
  - https://www.dropbox.com/s/ul0l3q98w54dr8x/WordCount.ipynb?dl=0
- **Cd to the directory that contains the wordcount notebook**
  - cd /Users/jshanahan/Dropbox/NativeX-Internal/Publications/WSDM-2016
- **Launch Notebook**
  - /Users/jshanahan/anaconda/bin/ipython notebook&

localhost:8888/notebooks/Notebooks/WordCount/WordCount.ipynb#

jupyter WordCount (autosaved)

File Edit View Insert Cell Kernel Help Python 2

Code Cell Toolbar: None

In [2]:
```
import os
import sys
#spark_home = os.environ['SPARK_HOME'] = '/Users/liang/Downloads/spark-1.4.1-bin-hadoop2.6/'
spark_home = os.environ['SPARK_HOME'] = '/Users/jshanahan/Dropbox/Lectures-UC-Berkeley-ML-Class-2015/spark-1.5.0-bin-hadoop2.6'


if not spark_home:
    raise ValueError('SPARK_HOME enviroment variable is not set')
sys.path.insert(0,os.path.join(spark_home,'python'))
sys.path.insert(0,os.path.join(spark_home,'python/lib/py4j-0.8.2.1-src.zip'))
execfile(os.path.join(spark_home,'python/pyspark/shell.py'))
```

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.5.0
      /_/

Using Python version 2.7.11 (default, Dec  6 2015 18:57:58)
SparkContext available as sc, HiveContext available as sqlContext.
```

In [3]:
```
%%writefile wordcount.txt
hello hi hi hallo
bonjour hola hi ciao
nihao konnichiwa ola
hola nihao hello
```

```
Writing wordcount.txt
```

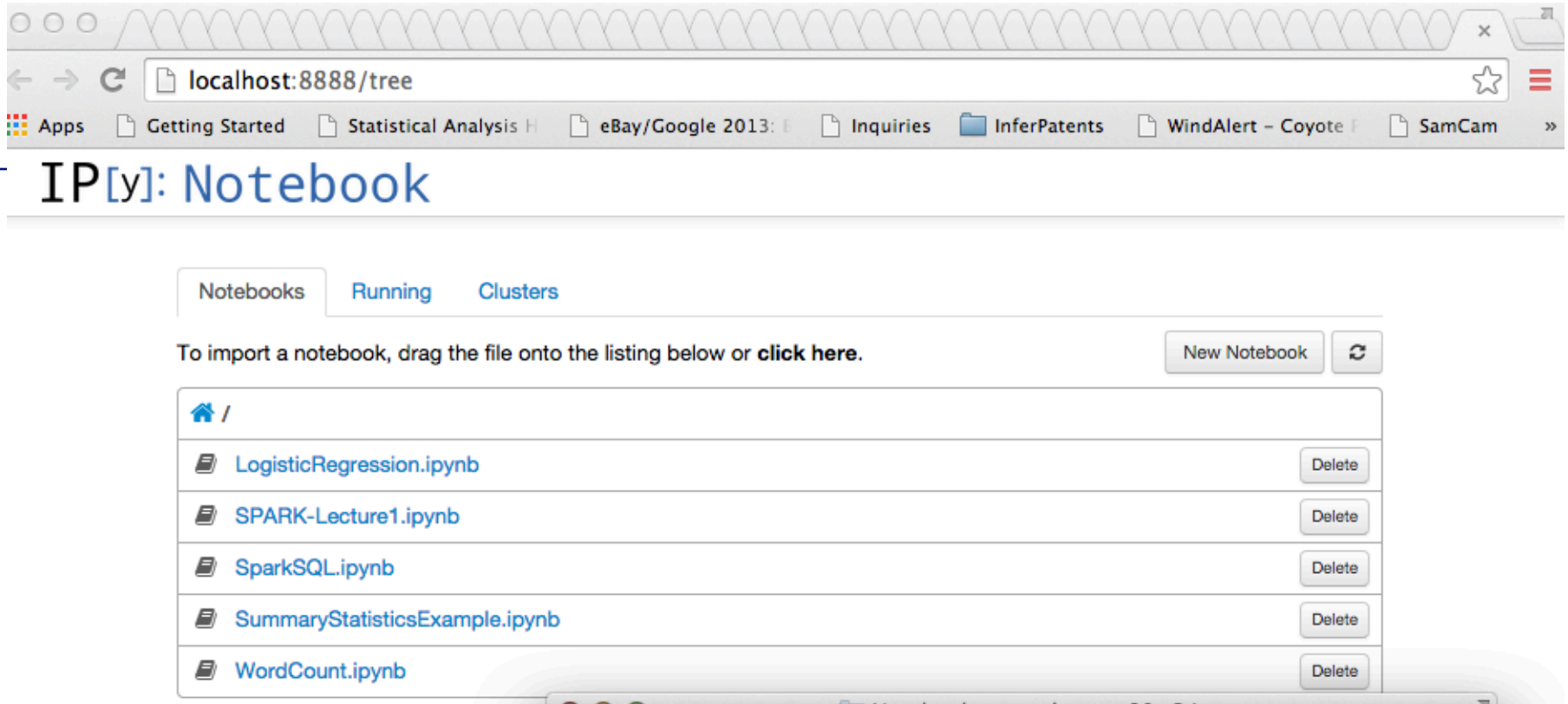
In [4]:
```
cat wordcount.txt
```

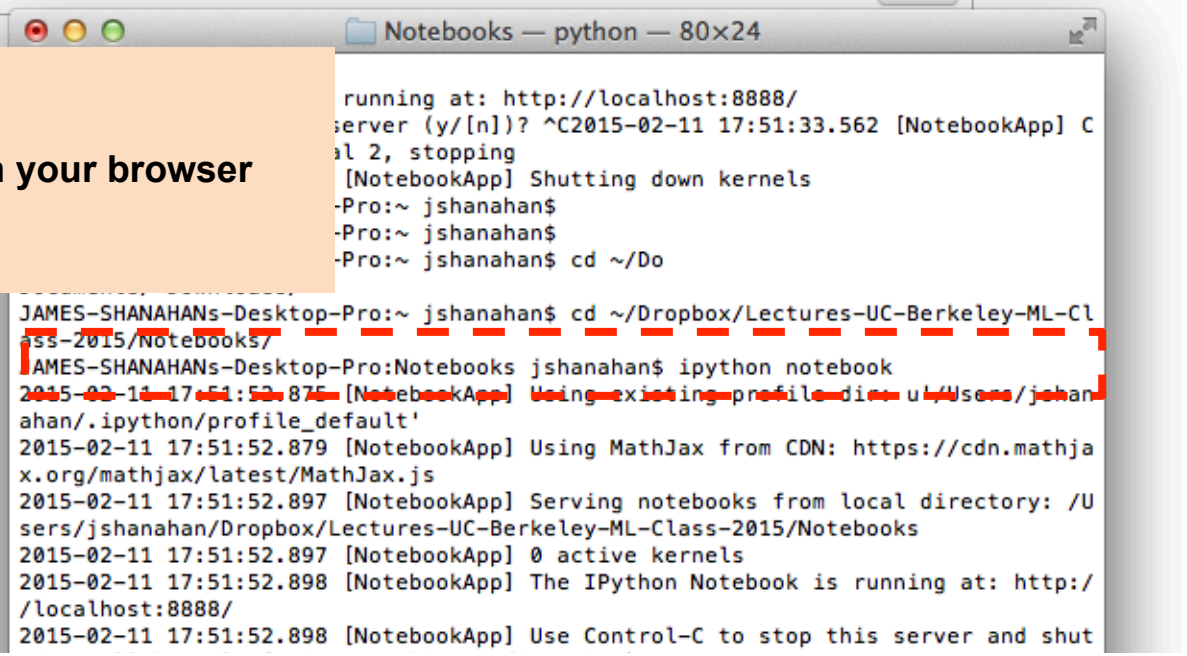
```
hello hi hi hallo
bonjour hola hi ciao
nihao konnichiwa ola
hola nihao hello
```

In [ ]:

In [3]:
```
#Count words in README.md
logFileNAME = 'wordcount.txt'
text_file = sc.textFile(logFileNAME)
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
for v in counts.collect():
    print v
```

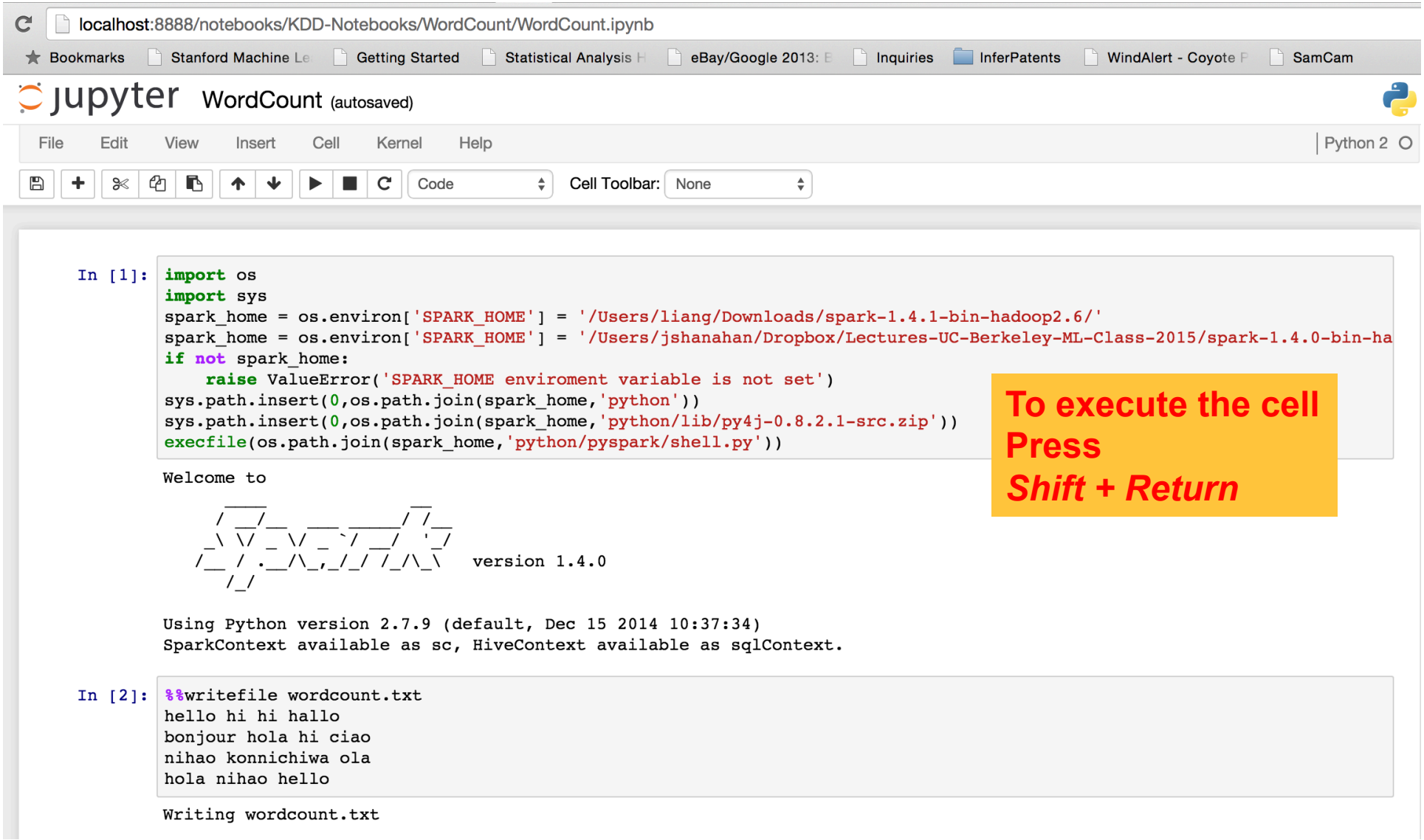
```
(u'ciao', 1)
(u'bonjour', 1)
```





mkdir SparkTutorial
cd SparkTutorial
#start the python server and notebook in your browser
 # from the shell/terminal command line
$ ipython notebook &

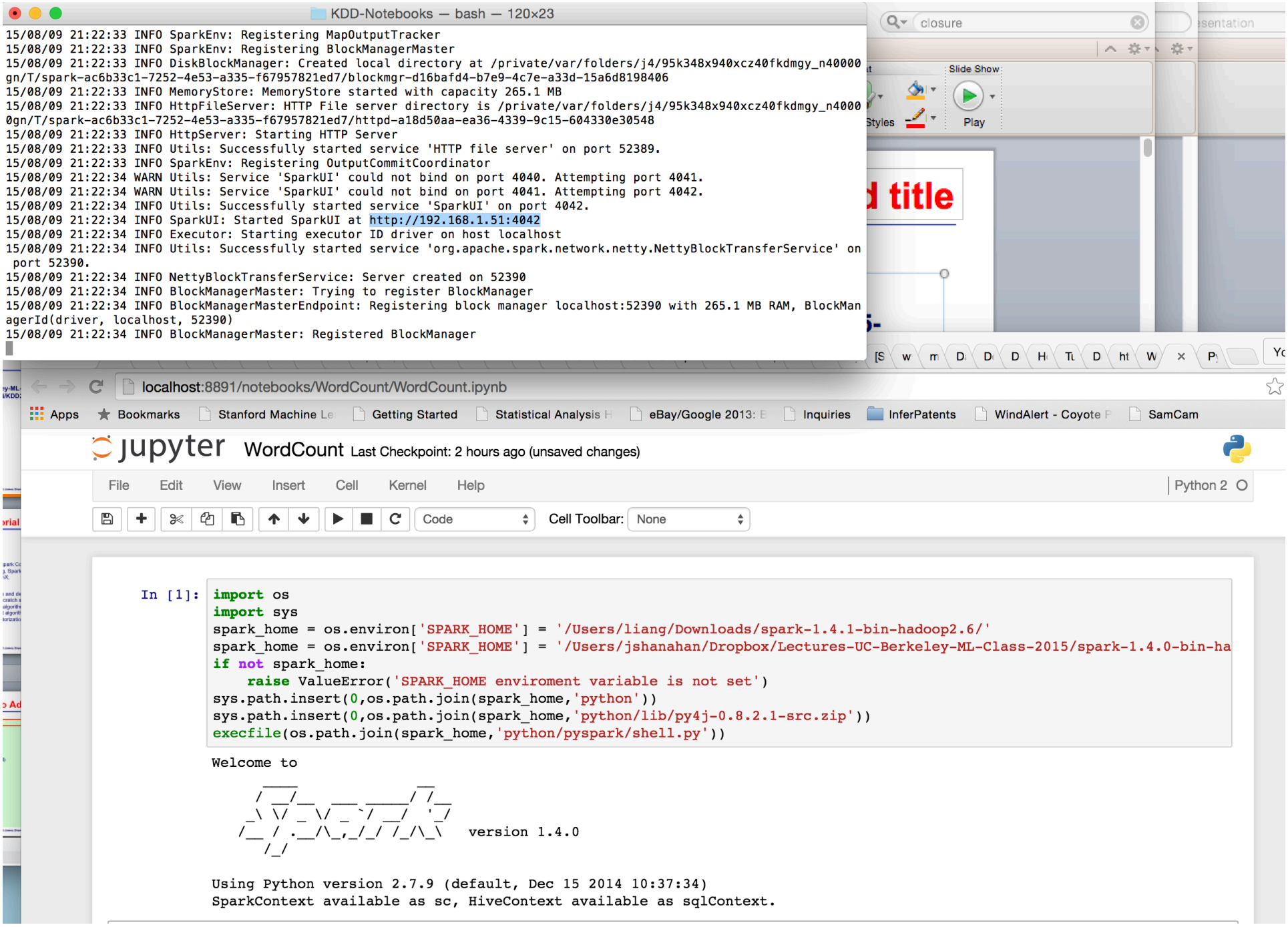
Modify path for SPARK_HOME

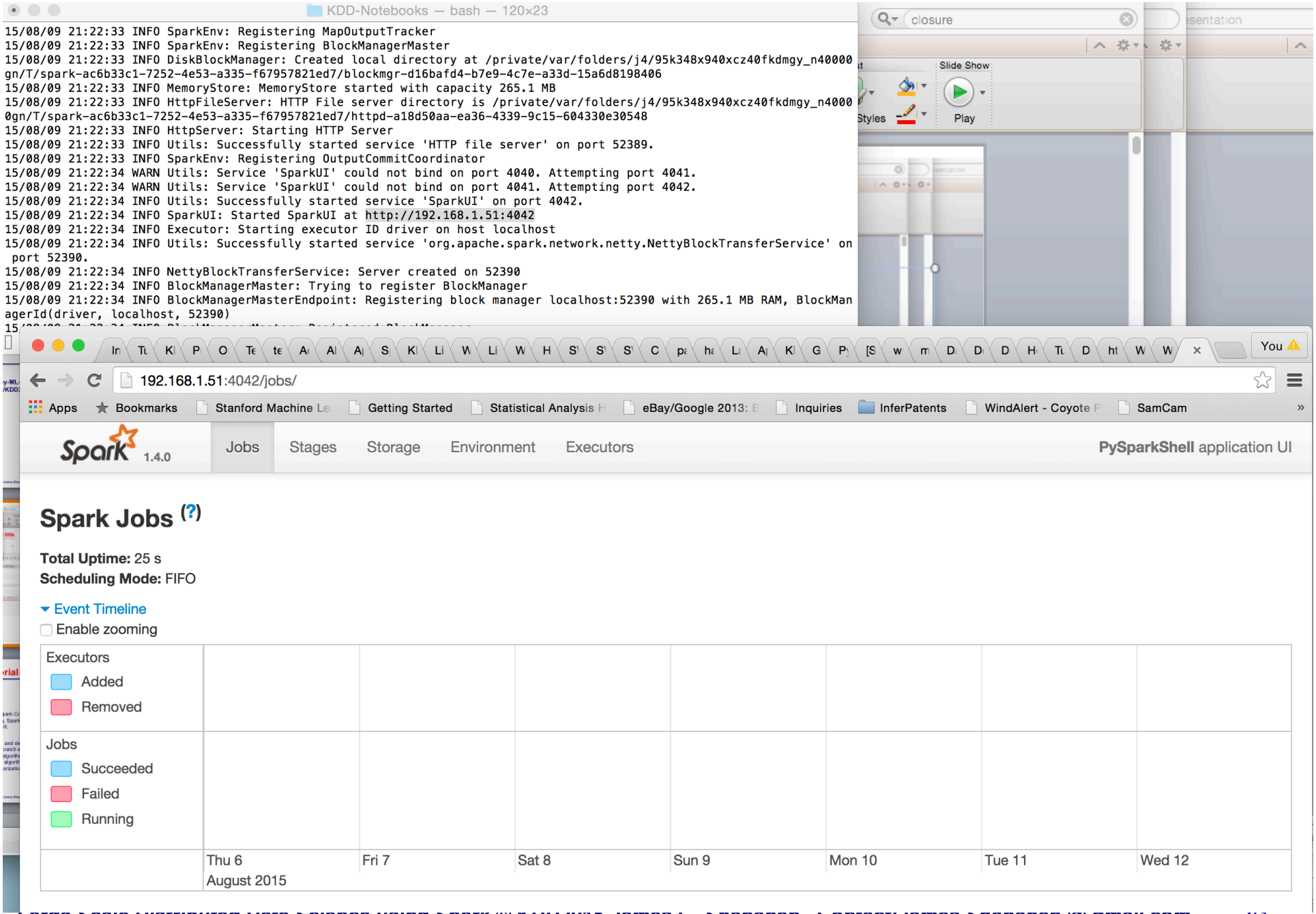


WordCount example test

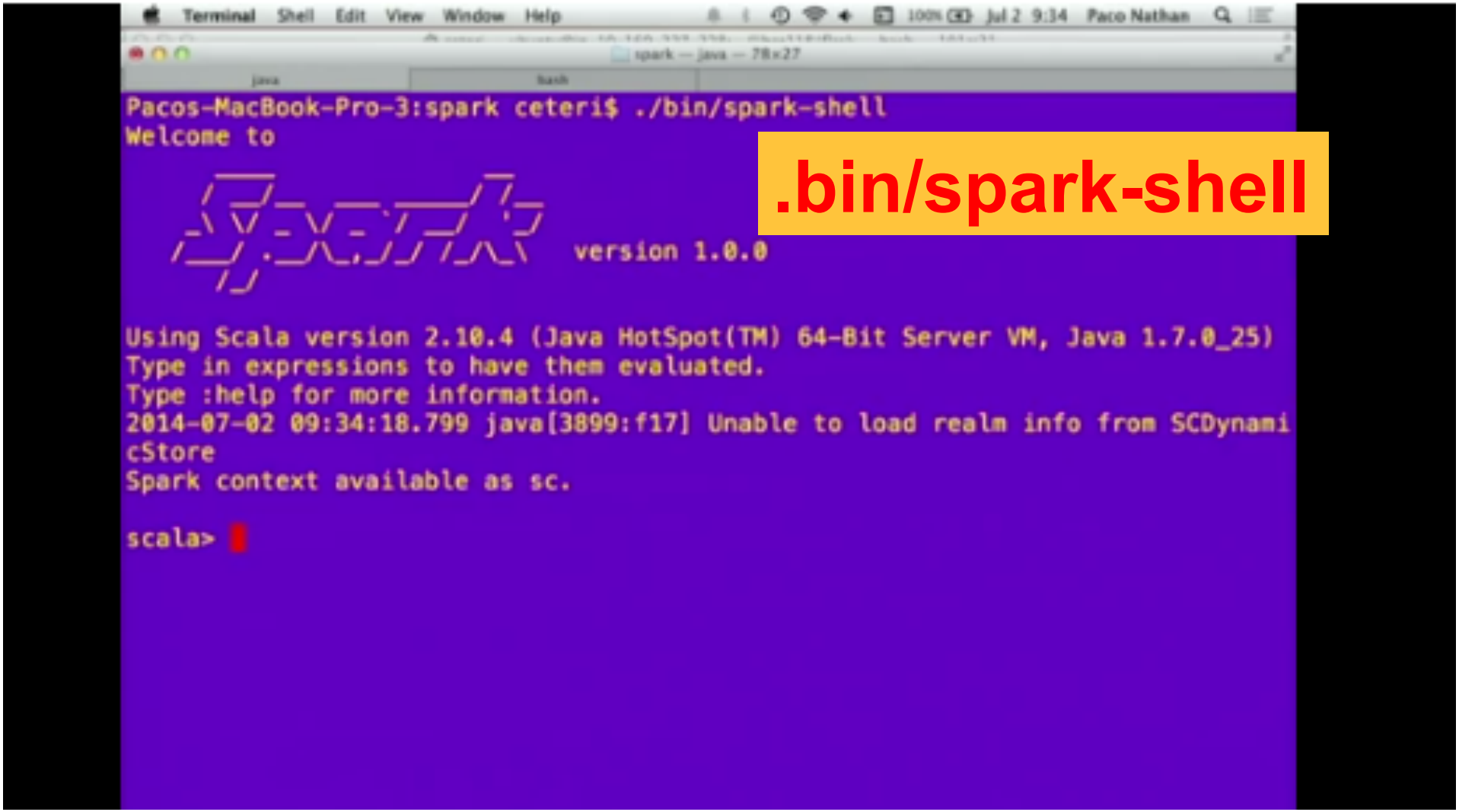
- https://www.dropbox.com/s/ul0l3q98w54dr8x/WordCount.ipynb?dl=0

- **Complete during the break**

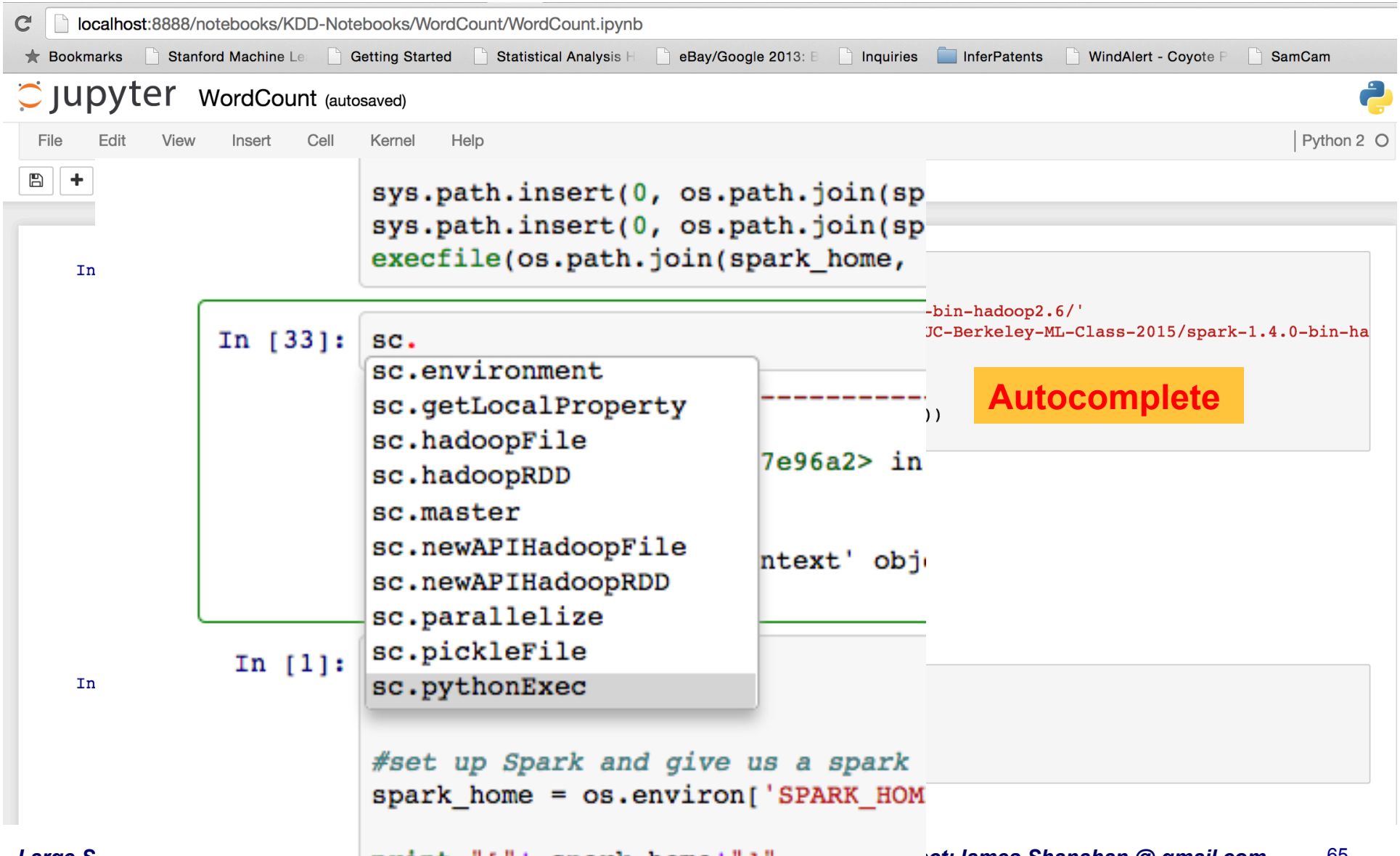

KDD-Notebooks — bash — 120×23
15/08/09 21:22:33 INFO SparkEnv: Registering MapOutputTracker
15/08/09 21:22:33 INFO SparkEnv: Registering BlockManagerMaster
15/08/09 21:22:33 INFO DiskBlockManager: Created local directory at /private/var/folders/j4/95k348x940xcz40fkdmgy_n40000gn/T/spark-ac6b33c1-7252-4e53-a335-f67957821ed7/blockmgr-d16bafd4-b7e9-4c7e-a33d-15a6d8198406
15/08/09 21:22:33 INFO MemoryStore: MemoryStore started with capacity 265.1 MB
15/08/09 21:22:33 INFO HttpFileServer: HTTP File server directory is /private/var/folders/j4/95k348x940xcz40fkdmgy_n40000gn/T/spark-ac6b33c1-7252-4e53-a335-f67957821ed7/httpd-a18d50aa-ea36-4339-9c15-604330e30548
15/08/09 21:22:33 INFO HttpServer: Starting HTTP Server
15/08/09 21:22:33 INFO Utils: Successfully started service 'HTTP file server' on port 52389.
15/08/09 21:22:33 INFO SparkEnv: Registering OutputCommitCoordinator
15/08/09 21:22:34 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
15/08/09 21:22:34 WARN Utils: Service 'SparkUI' could not bind on port 4041. Attempting port 4042.
15/08/09 21:22:34 INFO Utils: Successfully started service 'SparkUI' on port 4042.
15/08/09 21:22:34 INFO SparkUI: Started SparkUI at http://192.168.1.51:4042
15/08/09 21:22:34 INFO Executor: Starting executor ID driver on host localhost
15/08/09 21:22:34 INFO Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlockTransferService' on port 52390.
15/08/09 21:22:34 INFO NettyBlockTransferService: Server created on 52390
15/08/09 21:22:34 INFO BlockManagerMaster: Trying to register BlockManager
15/08/09 21:22:34 INFO BlockManagerMasterEndpoint: Registering block manager localhost:52390 with 265.1 MB RAM, BlockManagerId(driver, localhost, 52390)
15/08/09 21:22:34 INFO BlockManagerMaster: Registered BlockManager
closure
Slide Show
Play
Styles
d title
localhost:8891/notebooks/WordCount/WordCount.ipynb
Apps
Bookmarks
Stanford Machine Le
Getting Started
Statistical Analysis H
eBay/Google 2013: E
Inquiries
InferPatents
WindAlert - Coyote F
SamCam
jupyter
WordCount
Last Checkpoint: 2 hours ago (unsaved changes)
File
Edit
View
Insert
Cell
Kernel
Help
Python 2
Code
Cell Toolbar:
None
In [1]:
import os
import sys
spark_home = os.environ['SPARK_HOME'] = '/Users/liang/Downloads/spark-1.4.1-bin-hadoop2.6/'
spark_home = os.environ['SPARK_HOME'] = '/Users/jshanahan/Dropbox/Lectures-UC-Berkeley-ML-Class-2015/spark-1.4.0-bin-ha
if not spark_home:
raise ValueError('SPARK_HOME enviroment variable is not set')
sys.path.insert(0,os.path.join(spark_home,'python'))
sys.path.insert(0,os.path.join(spark_home,'python/lib/py4j-0.8.2.1-src.zip'))
execfile(os.path.join(spark_home,'python/pyspark/shell.py'))
Welcome to
version 1.4.0
Using Python version 2.7.9 (default, Dec 15 2014 10:37:34)
SparkContext available as sc, HiveContext available as sqlContext.


KDD-Notebooks — bash — 120×23
15/08/09 21:22:33 INFO SparkEnv: Registering MapOutputTracker
15/08/09 21:22:33 INFO SparkEnv: Registering BlockManagerMaster
15/08/09 21:22:33 INFO DiskBlockManager: Created local directory at /private/var/folders/j4/95k348x940xcz40fkdmgy_n40000gn/T/spark-ac6b33c1-7252-4e53-a335-f67957821ed7/blockmgr-d16bafd4-b7e9-4c7e-a33d-15a6d8198406
15/08/09 21:22:33 INFO MemoryStore: MemoryStore started with capacity 265.1 MB
15/08/09 21:22:33 INFO HttpFileServer: HTTP File server directory is /private/var/folders/j4/95k348x940xcz40fkdmgy_n40000gn/T/spark-ac6b33c1-7252-4e53-a335-f67957821ed7/httpd-a18d50aa-ea36-4339-9c15-604330e30548
15/08/09 21:22:33 INFO HttpServer: Starting HTTP Server
15/08/09 21:22:33 INFO Utils: Successfully started service 'HTTP file server' on port 52389.
15/08/09 21:22:33 INFO SparkEnv: Registering OutputCommitCoordinator
15/08/09 21:22:34 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
15/08/09 21:22:34 WARN Utils: Service 'SparkUI' could not bind on port 4041. Attempting port 4042.
15/08/09 21:22:34 INFO Utils: Successfully started service 'SparkUI' on port 4042.
15/08/09 21:22:34 INFO SparkUI: Started SparkUI at http://192.168.1.51:4042
15/08/09 21:22:34 INFO Executor: Starting executor ID driver on host localhost
15/08/09 21:22:34 INFO Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlockTransferService' on port 52390.
15/08/09 21:22:34 INFO NettyBlockTransferService: Server created on 52390
15/08/09 21:22:34 INFO BlockManagerMaster: Trying to register BlockManager
15/08/09 21:22:34 INFO BlockManagerMasterEndpoint: Registering block manager localhost:52390 with 265.1 MB RAM, BlockManagerId(driver, localhost, 52390)
closure
Slide Show
Styles
Play
192.168.1.51:4042/jobs/
Apps
Bookmarks
Stanford Machine Le
Getting Started
Statistical Analysis H
eBay/Google 2013: E
Inquiries
InferPatents
WindAlert - Coyote F
SamCam
Spark 1.4.0
Jobs
Stages
Storage
Environment
Executors
PySparkShell application UI
Spark Jobs (?)
Total Uptime: 25 s
Scheduling Mode: FIFO
Event Timeline
Enable zooming
Executors
Added
Removed
Jobs
Succeeded
Failed
Running
Thu 6
August 2015
Fri 7
Sat 8
Sun 9
Mon 10
Tue 11
Wed 12

Spark shell in Scala



TAB lists the available methods and attributes



Tutorial Outline

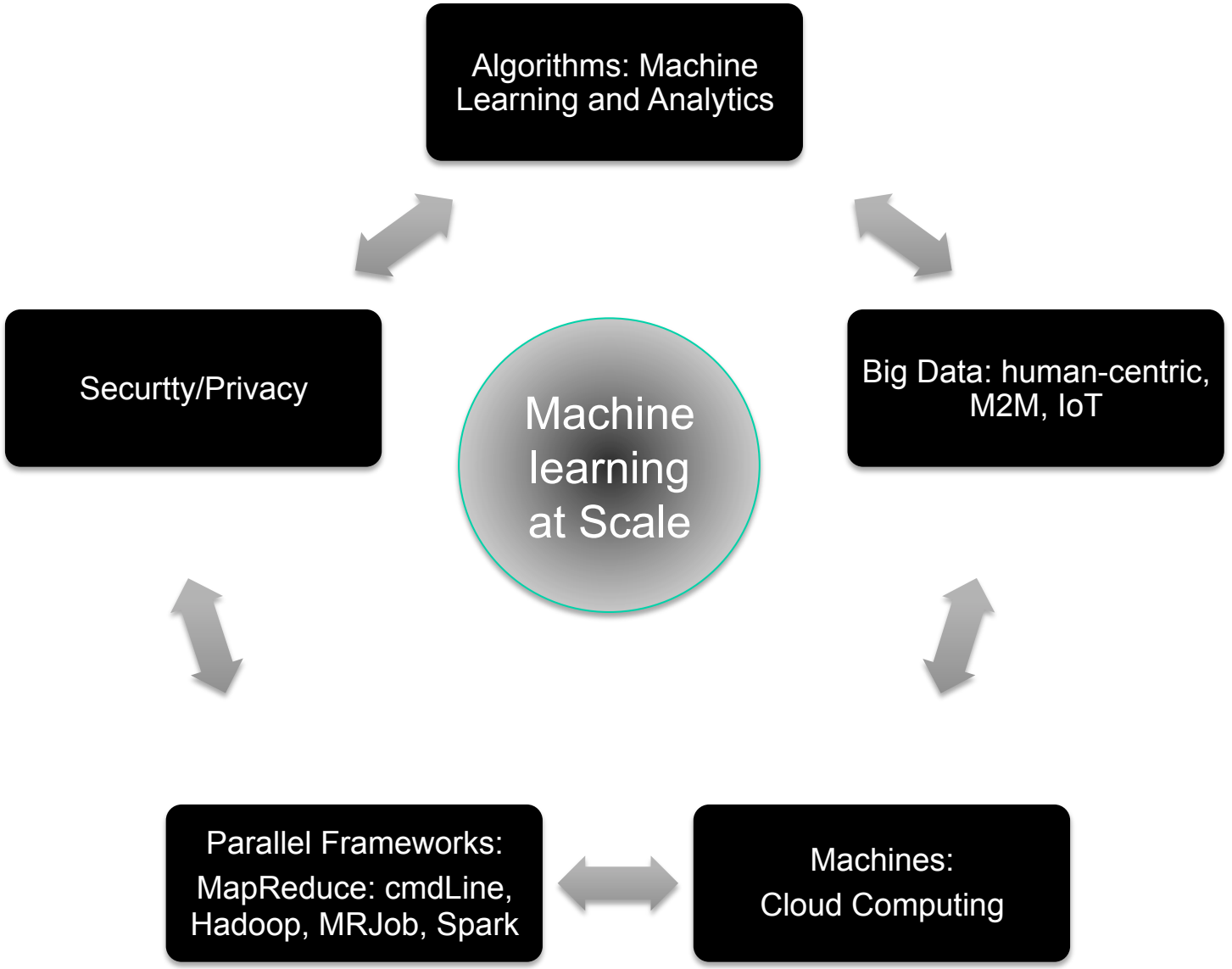
- **Part 1: Introduction [30 minutes]**
  - Welcome Survey
  - Install Spark
  - Background and motivation
- **Part 2: Spark Intro and basics [1.5 hours concept+exercises]**
  - fundamental Spark concepts, including Spark Core, data frames, the Spark Shell, Spark Streaming, Spark SQL and vertical libraries such as MLlib and GraphX;
- **Part 3: Machine learning in Spark 3-4 hours**
  - Naïve Bayes +
  - will focus on hands-on algorithmic design and development with Spark developing algorithms from scratch such as linear regression, logistic regression, graph processing algorithms such as pagerank/shortest path, *etc.*
- **Part 4: Wrap up [20]**
  - Latest version of Spark and beyond
  - Summary

Part 1

- **Part 1: Introduction**
  - Welcome Survey
  - Install Spark
  - Background and Motivation
    - Big Data Science
    - Functional Programming
    - Poorman's Map-Reduce (to dividing and conquering)

- **Background and Motivation**
  - Big Data Science
  - Functional Programming
  - Poorman's Map-Reduce (to dividing and conquering)

Machine learning at Scale

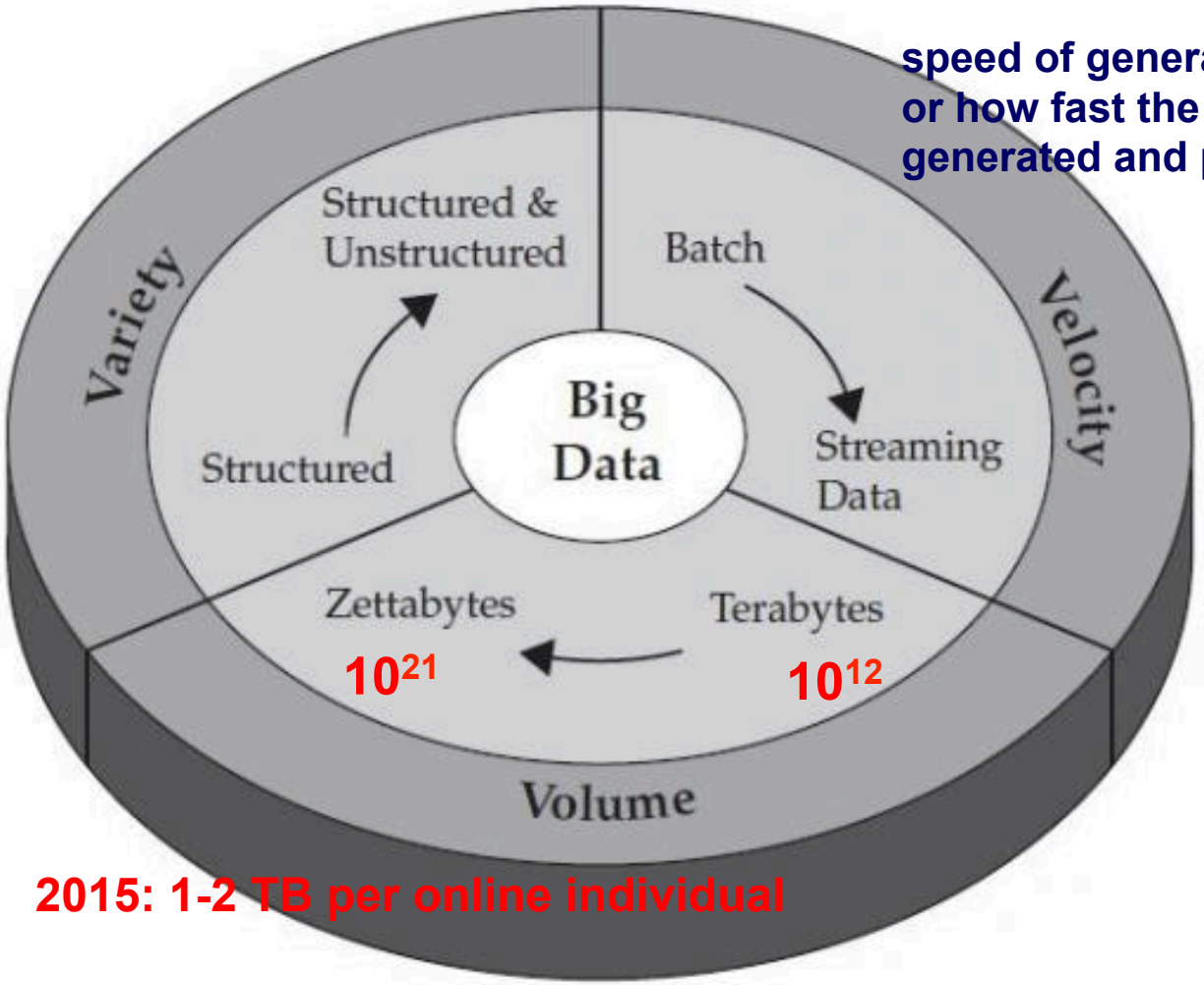


Big data Definition: use

- **Big data is a broad term for data sets so large or complex that traditional data processing applications are inadequate.**
  - PROCESSING:
    - Think of your laptop that gets overwhelmed with 3-4 gig of data (disk space is 1TB)
  - STORAGE:
    - Laptop : 1 TB
  - THROUGH-PUT
    - 1TB would take 3 hours to read it using your laptop
- **Challenges**
  - Challenges include analysis, capture, data curation, search, sharing, storage, transfer, visualization, security, and information privacy.

- **In 2012, Gartner updated its definition as follows: "Big data is high volume, high velocity, and/or high variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization."[18]**

Big Data: V^3



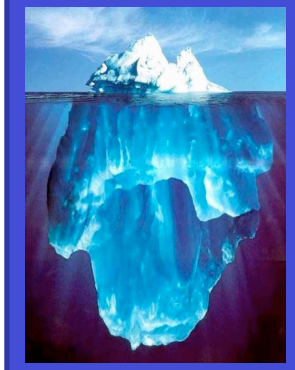
speed of generation of data or how fast the data is generated and processed

2015: 1-2 TB per online individual

Figure 1-1 *IBM characterizes Big Data by its volume, velocity, and variety—or simply, V^3.*

Sources Driving Big Data

It's All Happening On-line

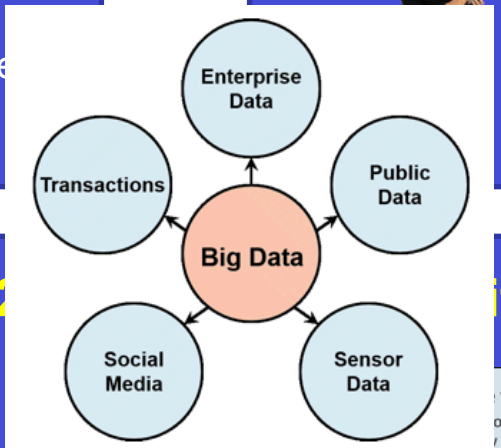


Every:
Click
Ad impression
Billing event
Fast Forward, pause,…
Friend Request
Transaction
Network message
Fault
…

User Generated (Web, Social & Mobile) Quantified Self

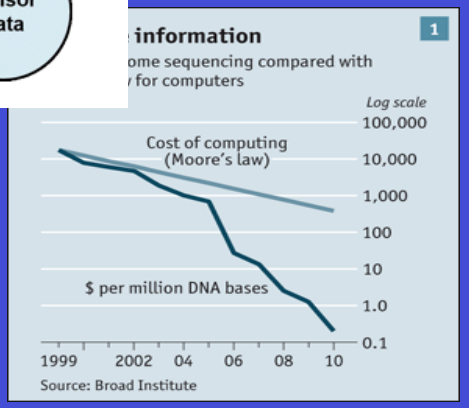



…
..

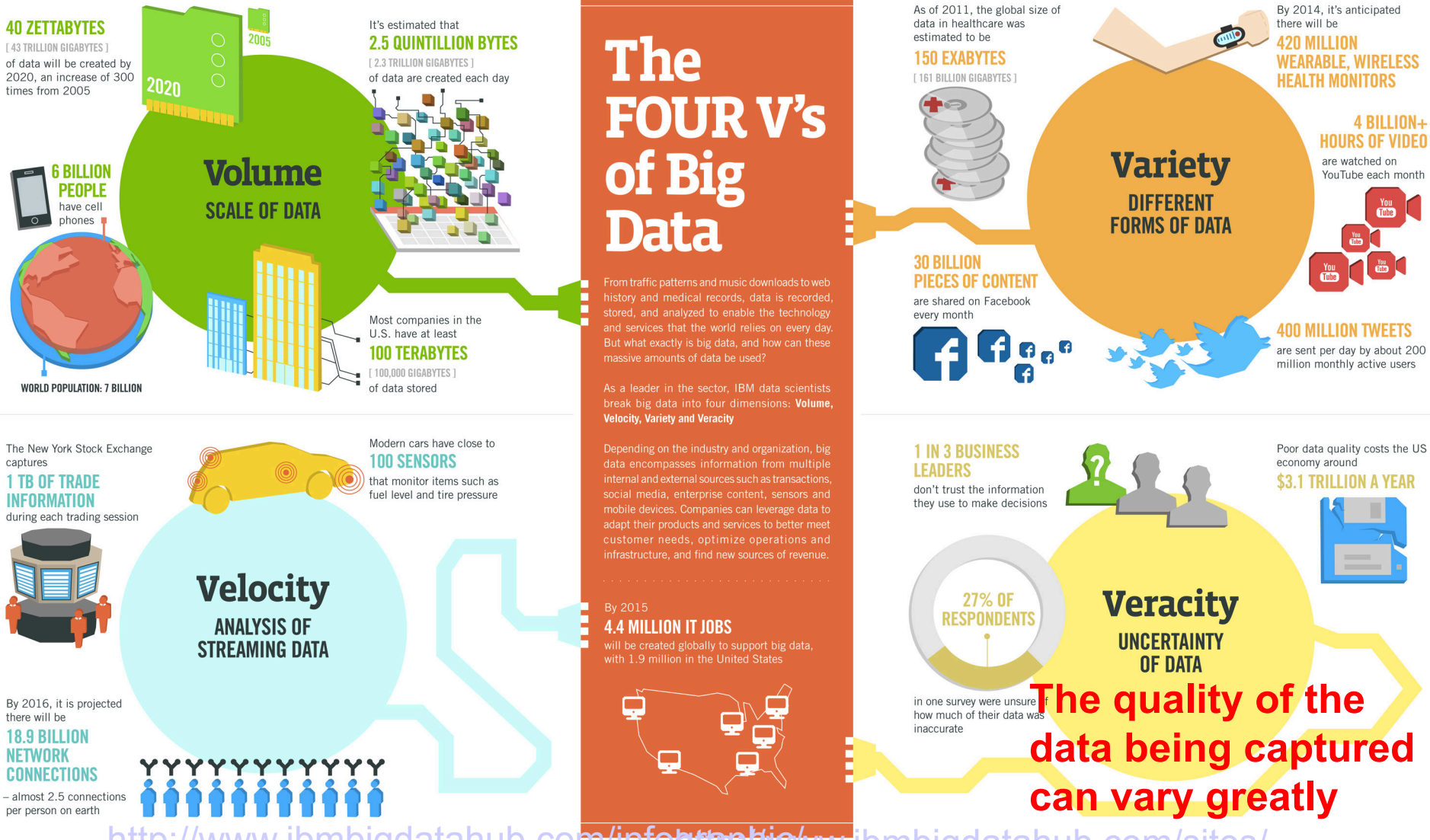
Internet of Things / M2

ific Computing



By 2005 we had 120* 10^{18}
By 2007 we had 280*20^{18}
By 2020 we will have 40* 10^{21}

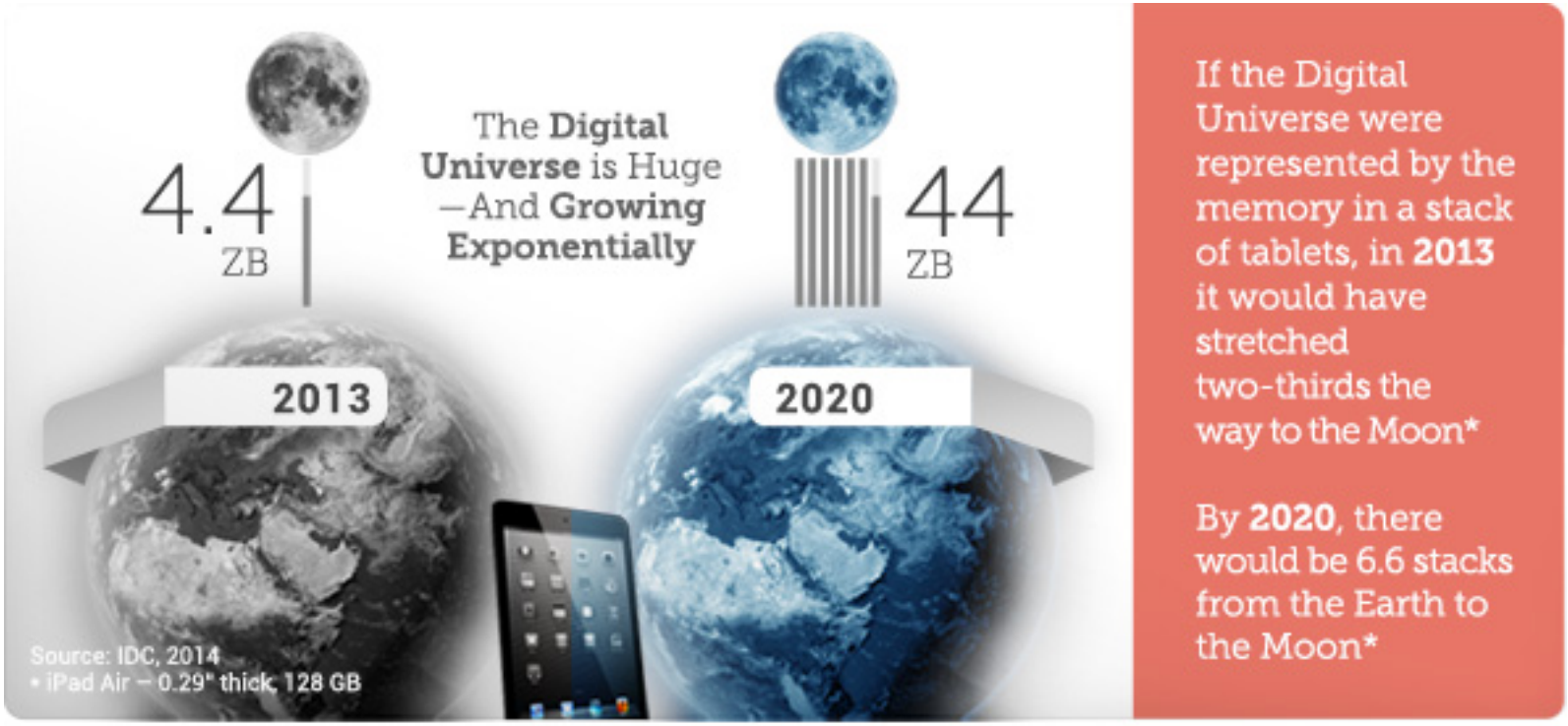
Big Data Infographic



The quality of the data being captured can vary greatly

http://www.ibmbigdatahub.com/infographic/four-vs-big-data

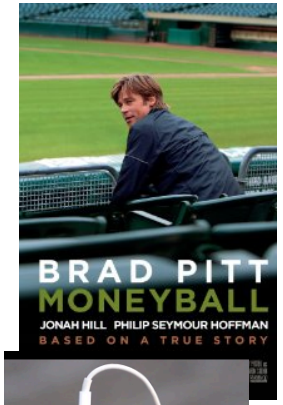
http://www.ibmbigdatahub.com/sites/default/files/infographic_file/4-Vs-of-big-data.jpg

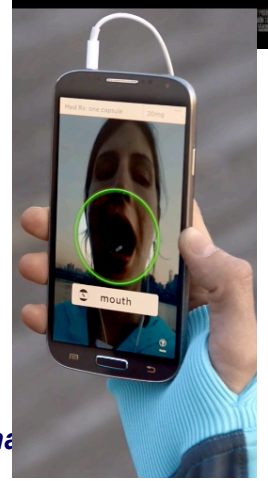


http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm

Why all the excitement?

- **Government:**
  - Obama used 80 pieces of information on each person; 4 year history (versus Romney)
  - Nate Silver used Bayesian techniques to publish analyses and predictions related to the 2008 and 2012 United States presidential election
- **Sports:**
  - Oakland Athletics baseball team and its manager Billy Beane
- **Transportation (e.g., Autonomous Vehicles)**
- **HCI: Speech Recognition and Translation**
- **Healthcare**
  - AI Cure: Do you know if your patients are taking their meds?
- **Digital Advertising**
- **Search (web, local, mobile)**





3 Vs of Big Data

1-2 TB per person today2014/2015

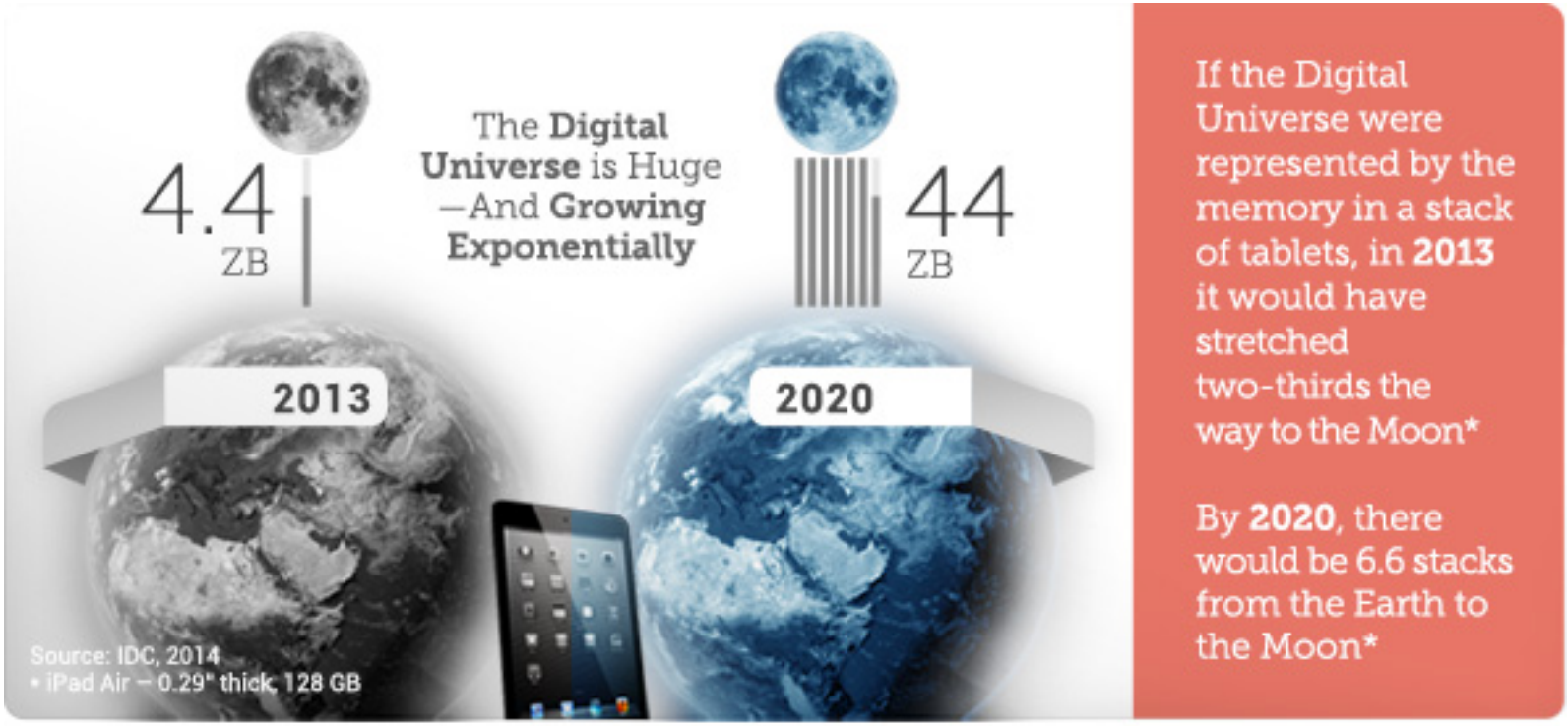
The data from these sources has a number of features that make it a challenge for a data warehouse:

Exponential Growth. An estimated 2.8ZB of data in 2012 is expected to grow to 40ZB by 2020. 85% of this data growth is expected to come from new types; with machine-generated data being projected to increase 15x by 2020. (Source IDC)

40TB per person by 2020

Varied Nature. The incoming data can have little or no structure, or structure that changes too frequently for reliable schema creation at time of ingest.

Value at High Volumes. The incoming data can have little or no value as individual, or small groups of records. But high volumes and longer historical perspectives can be inspected for patterns and used for advanced analytic applications.



http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm

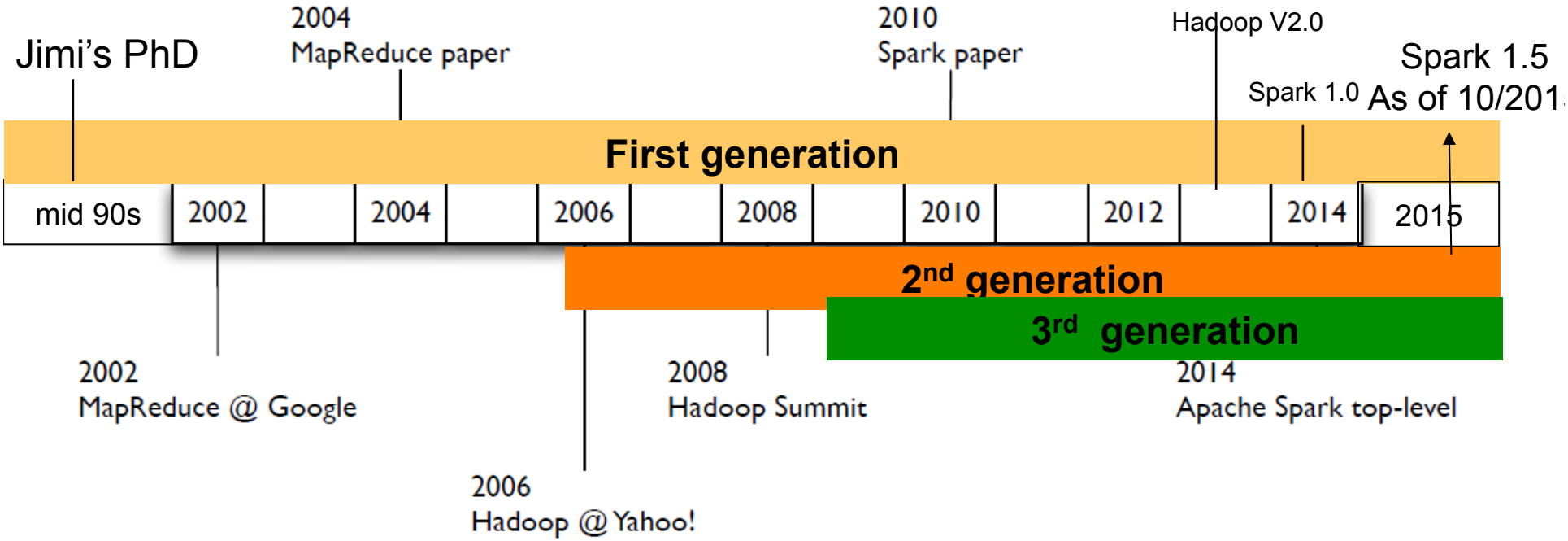
Personal; society; M2M; crowdsourcing

- **Society**
  - Graphs: Social, professional;
  - Quantified self: Eating; Sleeping; exercising
  - Voting
  - Education
  - Healthcare…. Economics, shopping, etc.
- **Internet of things**
  - Tracking Wildebeests in Serengeti, Tanzania (not just with GPS tags, but also with cameras at key strategic locations through out the Serengeti
    - Population changes in species; Scheduling safaris
  - 1 Billion smart meters by 2020;
    - 1 Petabyte of data per day? 10^9 =10^12 10^15
    - 1 Billion smart meters (One megabye of data per device per day; Poll meter 1000 times per day; 1000 bytes of data each time
  - Smart cities
- **Etc.**

Three generations of machine learning

- **First generation: dataset that fits in memory**
  - Single node learning summary statistics and some batch modeling (at small scale); SQL, R
  - Down sampling the data
- **Second generation: General purpose clusters and frameworks**
  - Distributed frameworks that allows us to divide and conquer problems
  - Learning using general purpose frameworks such as hadoop big data analysis offline, realtime decision making, homegrown specialist systems (Hadoop for analysis and modeling;), Hadoop, R
  - In-house purpose built systems; specialist sport
- **Third generation: Purpose-built libraries and frameworks**
  - Built for iterative algorithms that are common place in ML
  - huge scale realtime analysis and decision making systems
  - Specialized frameworks for large scale manipulation the type of data you are workign with.
  - For example, Machine learning libraries like MLLib in Spark, graph processing libraries like Apache Giraph or GraphX in Spark

Evolution of Map-Reduce frameworks for big data processing



Part 1

- **Part 1: Introduction**
  - Welcome Survey
  - Install Spark
  - Background and Motivation
    - Big Data Science
    - Functional Programming
    - Poorman's Map-Reduce (to dividing and conquering)

Map-Reduce primitives

- **Map-Reduce/Hadoop was inspired by the concept of functional languages:**
  - "Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages….
  - Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance." Jeffrey Dean and Sanjay Ghemawat, Google

- **MapReduce: Simplied Data Processing on Large Clusters, Jeffrey Dean and Sanjay Ghemawat, Google, 2004**

Functional Programming as a guide for Spark

- apply()
- map()
- filter()
- reduce()
- Lambda functions
- List comprehensions

Functional programming background

- **Treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.**
- **It is a declarative programming paradigm, which means programming is done with expressions.**
- **In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result f(x) each time.**
- **No side effects**
  - (side effects, i.e. changes in state that do not depend on the function inputs)
  - Can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional Programming

In computer science, **functional programming** is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate computability, the Entscheidungsproblem, function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus. In the other well-known declarative programming paradigm, logic programming, relations are at the base of respective languages.[1]

In contrast, imperative programming changes state with commands in the source language, the most simple example being assignment. Imperative programming does have functions, not in the mathematical sense, but in the sense of subroutines. They can have side effects that may change the value of program state. Functions without return values therefore make sense. Because of this, they lack referential transparency, i.e. the same language expression can result in different values at different times depending on the state of the executing program.[1]

Functional programming languages, especially purely functional ones such as Hope and Rex, have largely been emphasized in academia rather than in commercial software development. However, prominent functional programming languages such as Common Lisp, Scheme,[2][3][4][5] Clojure, Racket,[6] Erlang,[7][8][9] OCaml,[10][11] Haskell,[12][13] and F#[14][15] have been used in industrial and commercial applications by a wide variety of organizations. Functional programming is also supported in

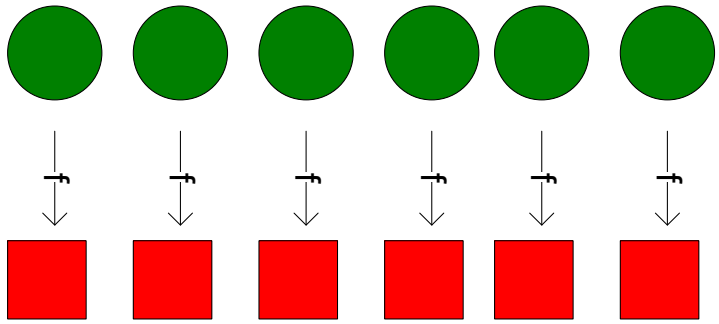
Higher-Order Functions

- A key feature of functional languages is the concept of higher order functions, or functions that can accept other functions as arguments (e.g., APL, Lisp and ML)
- A higher-order function is a function that takes another function as a parameter
- They are "higher-order" because it's a function of a function
- Examples
  - Map (aka apply to all)
  - Reduce (aka fold)
  - Filter
- Lambda works great as a parameter to higher-order functions if you can deal with its limitations

Map

`map(function, iterable, ...)`

- Map applies **function** to each element of **iterable** and creates a list of the results
- You can optionally provide more iterables as parameters to map and it will place tuples in the result list
- Map returns an iterator which can be cast to list

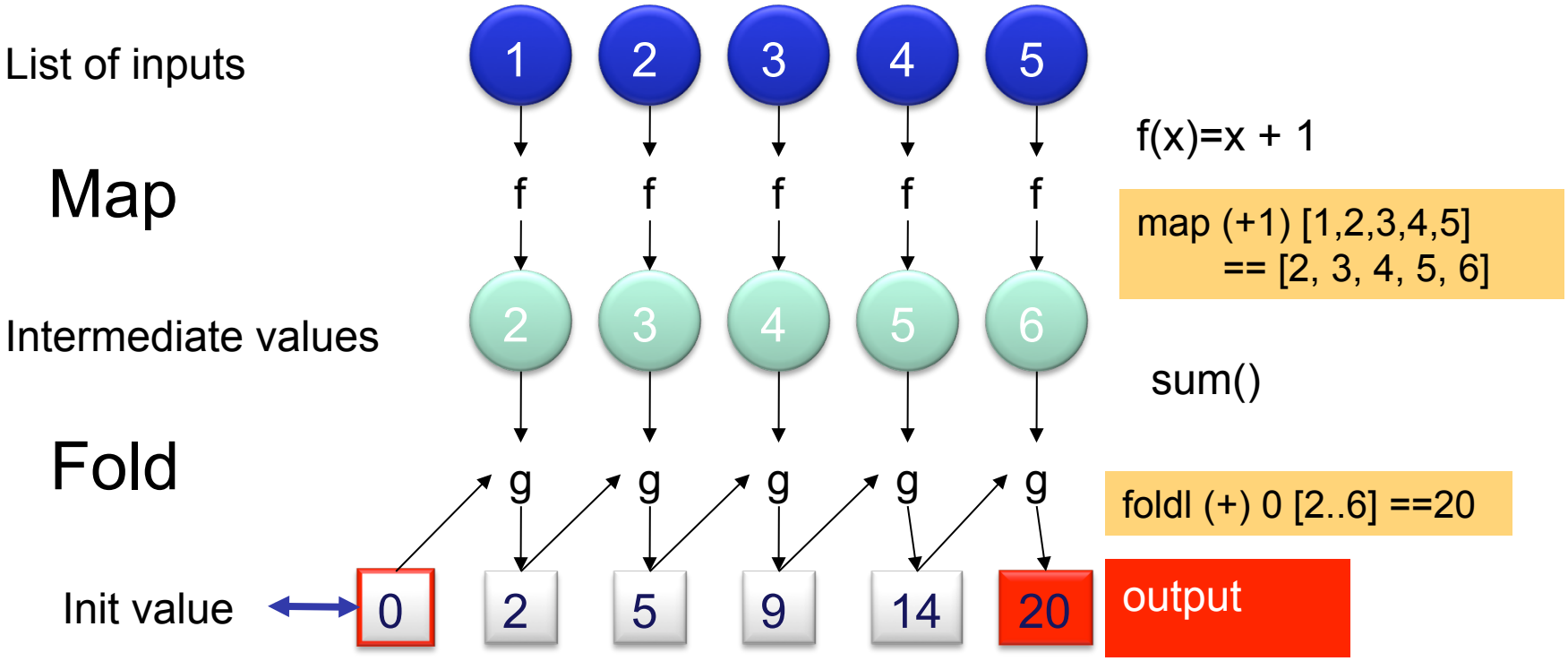


This an abstract diagram, despite this you can see the potential for parallelism especially in the map phase

MapReduce is derived from FP

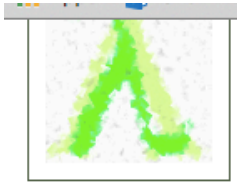
MapReduce ~ Map + Fold from functional programming!

Recursive defn but this unrolls into a loop



Lambda functions

- Shorthand version of def statement, useful for "inlining" functions and other situations where it's convenient to keep the code of the function close to where it's needed
- Can only contain an expression in the function definition, not a block of statements (e.g., no if statements, etc)
- A lambda returns a function; the programmer can decide whether or not to assign this function to a name

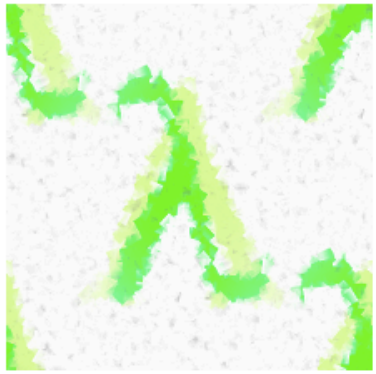


Python 2 Tutorial

Python: Lambda functions (closures)

Lambda, filter, reduce and map

Lambda Operator



Some like it, others hate it and many are afraid of the lambda operator. We are confident that you will like it, when you have finished with this chapter of our tutorial. If not, you can learn all about "List Comprehensions", Guido van Rossums preferred way to do it, because he doesn't like Lambda, map, filter and reduce either.

becasu The lambda operator or lambda function is a way to create small anonymous functions, i.e. functions without a name. These functions are throw-away functions, i.e. they are just needed where they have been created. Lambda functions are mainly used in combination with the functions filter(), map() and reduce(). The lambda feature was added to Python due to the demand from Lisp programmers.

The general syntax of a lambda function is quite simple:
lambda argument_list: expression
The argument list consists of a comma separated list of arguments and the expression is an arithmetic expression using these arguments. You can assign the function to a variable to give it a name.
The following example of a lambda function returns the sum of its two arguments:

```
>>> f = lambda x, y : x + y
>>> f(1,1)
2
```

The map() Function

The advantage of the lambda operator can be seen when it is used in combination with the map() function.
map() is a function with two arguments:

```
r = map(func, seq)
```

The first argument *func* is the name of a function and the second a sequence (e.g. a list) *seq*. *map()* applies the function *func* to all the elements of the sequence *seq*. It returns a new list with the elements changed by *func*

```
def fahrenheit(T):
    return ((float(9)/5)*T + 32)
def celsius(T):
    return (float(5)/9)*(T-32)
temp = (36.5, 37, 37.5,39)
```

```
F = map(fahrenheit, temp)
C = map(celsius, F)
```

In the example above we haven't used lambda. By using lambda, we wouldn't have had to define and name the functions fahrenheit() and celsius(). You can see this in the following interactive session:

```
>>> Celsius = [39.2, 36.5, 37.3, 37.8]
>>> Fahrenheit = map(lambda x: (float(9)/5)*x + 32, Celsius)
>>> print Fahrenheit
[102.56, 97.700000000000003, 99.140000000000001, 100.03999999999999]
```

Lambda example

- **Simple example:**

```
>>> def sum(x,y): return x+y
>>> ...
>>> sum(1,2)
3

>>> sum2 = lambda x, y: x+y
>>> sum2(1,2)
3
```

Closure (computer programming)

From Wikipedia, the free encyclopedia

For other uses of this term, including in mathematics and computer science, see Closure.
Not to be confused with Clojure.

In programming languages, **closures** (also **lexical closures** or **function closures**) are a technique for implementing lexically scoped name binding in languages with first-class functions. Operationally, a closure is a data structure storing a function[a] together with an environment:[1] a mapping associ
with the value or storage location
the function to access those *captu*
scope.

Example The following program f
`incrementBy`. The nested func
though `x` is not local to `increm`
the `y` value to the `x` value, and
find it once invoked:

```
function startAt(x)
   function incrementBy
      return x + y
   return incrementBy

variable closure1 = sta
variable closure2 = sta
```

Note that, as `startAt` returns a
return `4`, while invoking `closu`
associated environments differ, a
different values, thus evaluating th

Closure

A closure is a data structure storing a function[a] together with an environment:

- **a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or storage location the name was bound to at the time the closure was created.**
- A closure is a function that encloses its surrounding state by referencing fields external to its body. The enclosed state remains across invocations of the closure.

Closure (computer programming)

From Wikipedia, the free encyclopedia

> *For other uses of this term, including in mathematics and computer science, see Closure.*
> *Not to be confused with Clojure.*

In programming languages, **closures** (also **lexical closures** or **function closures**) are a technique for implementing lexically scoped name binding in languages with first-class functions. Operationally, a closure is a data structure storing a function[a] together with an environment:[1] a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or storage location the name was bound to at the time the closure was created.[b] A closure—unlike a plain function—allows the function to access those *captured variables* through the closure's reference to them, even when the function is invoked outside their scope.

Example The following program fragment defines a higher-order function `startAt` with a parameter `x` and a nested function `incrementBy`. The nested function `incrementBy` has access to `x`, because `incrementBy` is in the lexical scope of `x`, even though `x` is not local to `incrementBy`. The function `startAt` returns a closure containing the function `incrementBy`, which adds the `y` value to the `x` value, and a reference to the variable `x` *from this invocation* of `startAt`, so `incrementBy` will know where to find it once invoked:

```
function startAt(x)
   function incrementBy(y)
       return x + y
   return incrementBy

variable closure1 = startAt(1)
variable closure2 = startAt(5)
```

A closure is a data structure storing a function[a] together with an environment:[1] a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or storage location the name was bound to at the time the closure was created.

Note that, as `startAt` returns a function, the variables $closure_1$ and $closure_2$ are of function type. Invoking $closure_1(3)$ will return `4`, while invoking $closure_2(3)$ will return `8`. While $closure_1$ and $closure_2$ have the same function `incrementBy`, the associated environments differ, and invoking the closures will bind the name `x` to two distinct variables in the two invocations, with different values, thus evaluating the function to different results.

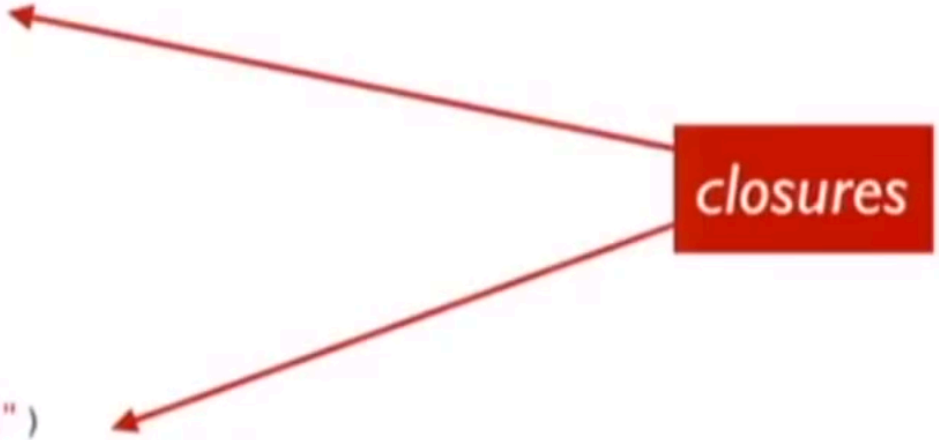
Lambda as a closure

Spark Essentials: *Transformations*

A closure is a function that encloses its surrounding state by referencing fields external to its body. The enclosed state remains across invocations of the closure.

Scala:

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



Python:

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

Map Example

Example

```
1  nums = [0, 4, 7, 2, 1, 0 , 9 , 3, 5, 6, 8, 0, 3]
2
3  nums = list(map(lambda x : x % 5, nums))
4
5  print(nums)
6  #[0, 4, 2, 2, 1, 0, 4, 3, 0, 1, 3, 0, 3]
7
8
9  def mod5(val) :
0       return x % 5
1
2  nums1 = list(map(mod5, nums))
   print(nums1)
   #[0, 4, 2, 2, 1, 0, 4, 3, 0, 1, 3, 0, 3]
```

Lambda/inline

apply()

- In very general contexts, you may not know ahead of time how many arguments need to get passed to a function (perhaps the function itself is built dynamically)
- The apply() function calls a given function with a list of arguments packed in a tuple:
  def sum(x, y): return x+y
  apply(sum, (3, 4))
  7
- Apply can handle functions defined with def or with lambda

Map Example: distance from origin

Goal: given a list of three dimensional points in the form of tuples, create a new list consisting of the distances of each point from the origin

Loop Method:

- distance(x, y, z) = sqrt(x**2 + y**2 + z**2)
- loop through the list and add results to a new list

Map Problem: distance from origin

Solution

```
1  from math import sqrt
2
3  points = [(2, 1, 3), (5, 7, -3), (2, 4, 0), (9, 6, 8)]
4
5  def distance(point) :
6      x, y, z = point
7      return sqrt(x**2 + y**2 + z**2)
8
9  distances = list(map(distance, points))
```

Filter: higher order function

`filter(`**`function, iterable`**`)`

- The filter runs through each element of **iterable** (any iterable object such as a `List` or another collection)
- It applies **function** to each element of **iterable**
- If **function** returns `True` for that element then the element is put into a `List`
- This list is returned from filter in versions of python under 3
- In python 3, filter returns an iterator which must be cast to type list with list()

Filter Example: filter all non-zeros

Example

```
1  nums = [0, 4, 7, 2, 1, 0 , 9 , 3, 5, 6, 8, 0, 3]
2
3  nums = list(filter(lambda x : x != 0, nums))
4
5  print(nums)        #[4, 7, 2, 1, 9, 3, 5, 6, 8, 3]
6
```

Filter Problem

```
NaN = float("nan")
scores = [[NaN, 12, .5, 78, math.pi],
          [2, 13, .5, .7, math.pi / 2],
          [2, NaN, .5, 78, math.pi],
          [2, 14, .5, 39, 1 - math.pi]]
```

Goal: given a list of lists containing answers to an algebra exam, filter out those that did not submit a response for one of the questions, denoted by `NaN`

Filter Problem

Solution

```
1  NaN = float("nan")
2  scores = [[NaN, 12, .5, 78, pi],[2, 13, .5, .7, pi / 2],
3            [2,NaN, .5, 78, pi],[2, 14, .5, 39, 1 - pi]]
4  #solution 1 - intuitive
5  def has_NaN(answers) :
6      for num in answers :
7          if isnan(float(num)) :
8              return False
9      return True
0  valid = list(filter(has_NaN, scores))
1  print(valid)
2
3  #Solution 2 - sick python solution
4  valid2 = list(filter(lambda x : NaN not in x, scores))
5  print(valid2)
```

Lambda/inline

Reduce

`reduce(`**`function, iterable`**`[,`**`initializer`**`])`

- Reduce will apply **function** to each element in **iterable** along with the sum so far and create a cumulative sum of the results
- **function** must take two parameters
- If initializer is provided, initializer will stand as the first argument in the sum
- Unfortunately in python 3 reduce() requires an import statement
  - `from functools import reduce`

Reduce Example

Example

```
1  nums = [1, 2, 3, 4, 5, 6, 7, 8]
2
3  nums = list(reduce(lambda x, y : (x, y), nums))
4
5  Print(nums)        #(((((((1, 2), 3), 4), 5), 6), 7), 8)
6
7
```

Reduce Problem

Goal: given a list of numbers I want to find the average of those numbers in a few lines using `reduce()`

For Loop Method:

- sum up every element of the list
- divide the sum by the length of the list

Reduce Problem

Solution

```
1  nums = [92, 27, 63, 43, 88, 8, 38, 91, 47, 74, 18, 16,
          29, 21, 60, 27, 62, 59, 86, 56]
2
3  sum = reduce(lambda x, y : x + y, nums) / len(nums)
4
```

MapReduce in Python

A framework for processing huge datasets on certain kinds of distributable problems

Map Step:

- master node takes the input,chops it up into smaller sub-problems, and distributes those to worker nodes.

- worker node may chop its work into yet small pieces and redistribute again

MapReduce

Reduce Step:

- master node then takes the answers to all the sub-problems and combines them in a way to get the output

MapReduce

Problem: Given an email how do you tell if it is spam?

- Count occurrences of certain words. If they occur too frequently the email is spam.

MapReduce in Python: single core

map_reduce.py

```
email = ['the', 'this', 'annoy', 'the', 'the', 'annoy']

def inEmail (x):
    if (x == "the"):
        return 1;
    else:
        return 0;

map(inEmail, l)                          #[1, 0, 0, 0, 1, 1, 0]

reduce((lambda x, xs: x + xs), map(inEmail, email)) #3
```

- **Purely functional**

Purely functional

Every function in Haskell is a function in the mathematical sense (i.e., "pure"). Even side-effecting IO operations are but a description of what to do, produced by pure code. There are no statements or instructions, only expressions which cannot mutate variables (local or global) nor access state like time or random numbers.

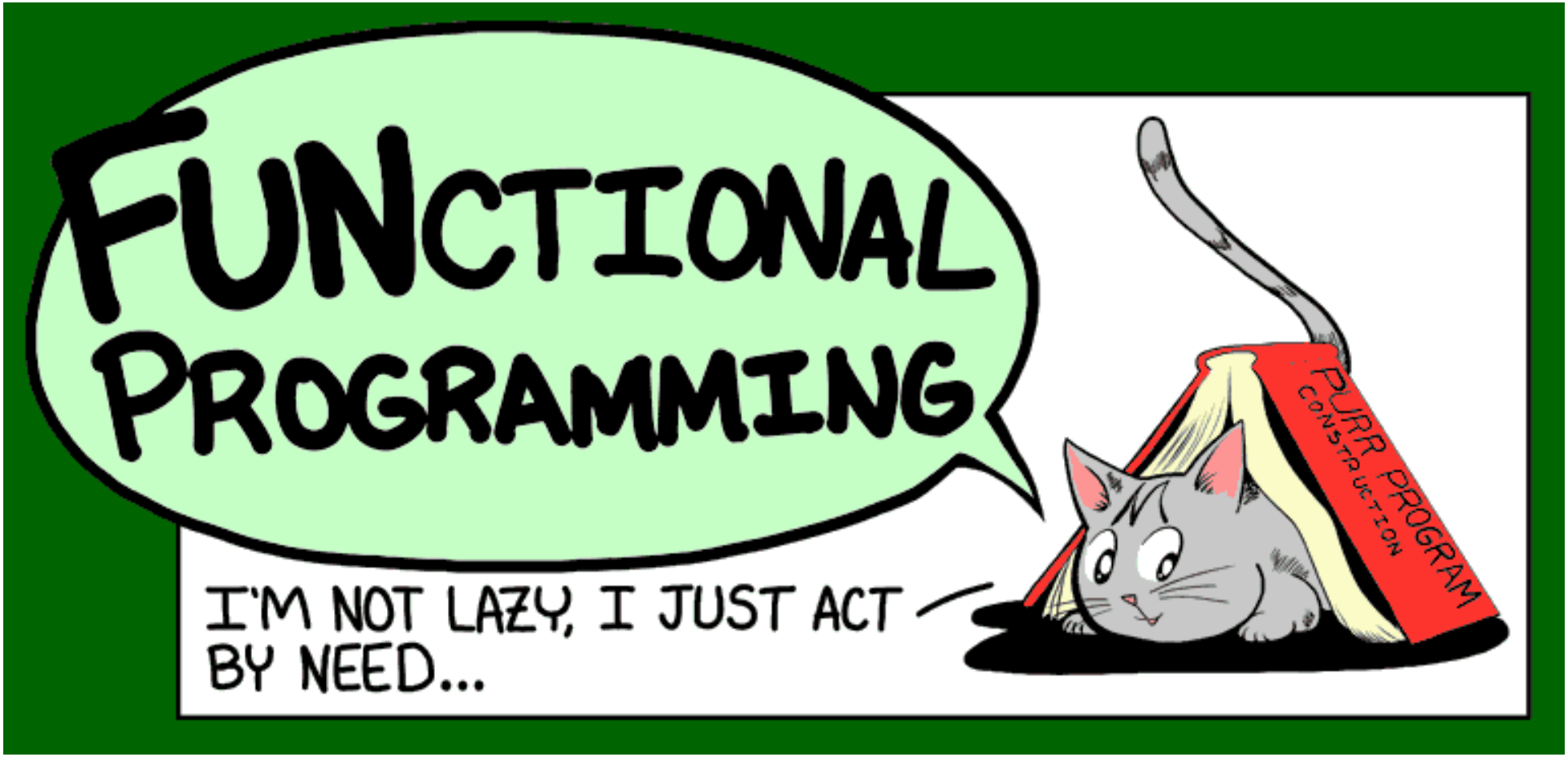
The following function takes an integer and returns an integer. By the type it cannot do any side-effects whatsoever, it cannot mutate any of its arguments.

```
square :: Int -> Int
square x = x * x
```

Functional Programming Review

- Functional operations do not modify data structures: They always create new ones
- Original data still exists in unmodified form
- Data flows are implicit in program design
- Order of operations does not matter
- Lists are primitive data types

Hadoop is not so lazy but Spark is!



- **In this section we talked about functional programming and have seen the map and reduce higher order functions and we seen the potential to parallelize at least the map function**

- **Over the next couple of sections we will see how the Spark framework has adopted FP constructs (albeit not purely observing the FP high standards)**
  - Lazy evaluation
  - Data is immutable
  - Map, reduce, and 80 other functions

Part 1

- **Part 1: Introduction**
  - Welcome Survey
  - Install Spark
  - Background and Motivation
    - Big Data Science
    - Functional Programming
    - Poorman's Map-Reduce (to dividing and conquering)

Assume 100 Billion webpages

- **How store them?**
  - 10K per page 10^{11} = 10^{15} Bytes (1 Petabyte of raw data)
- **How can we scan them?**
  - 10^{15} X 10^{-8} X 10^{-5} = 10^{2} days
- **How to do something useful with them?**
  - Document frequency for a term?
  - Extract outlinks of page and say just count the number of inlinks for a webpage

Why Parallelism?

- **Data size is increasing**
  - Single node architecture is reaching its limit
    - Scan 1000 TB on 1 node @ 100 MB/s = 120 days
  - Store that amount of data?
- **Growing demand to leverage data to do useful tasks**
- **Parallelism: Divide and conquer**
- **Standard/Commodity and affordable architecture emerging**
  - Cluster of commodity Linux nodes (CPU, memory and disk)
  - Gigabit ethernet interconnect

Design Goals for Parallelism

1. **Scalability to large data volumes:**
   - Scan 1000 TB (1PB) on 1 node @ 50 MB/s = 120 days
   - Scan on 10000-node cluster = 16 minutes!
   - 10^{15} X 10^{-8} X 10^{-5} = 10^{2} days = 10^{7} seconds
   - 10^{7} seconds X 10^{4} nodes = 10^{3} seconds = 16 minutes
   - 100 Gig per node (10^{11} bytes at 10^{8} / second

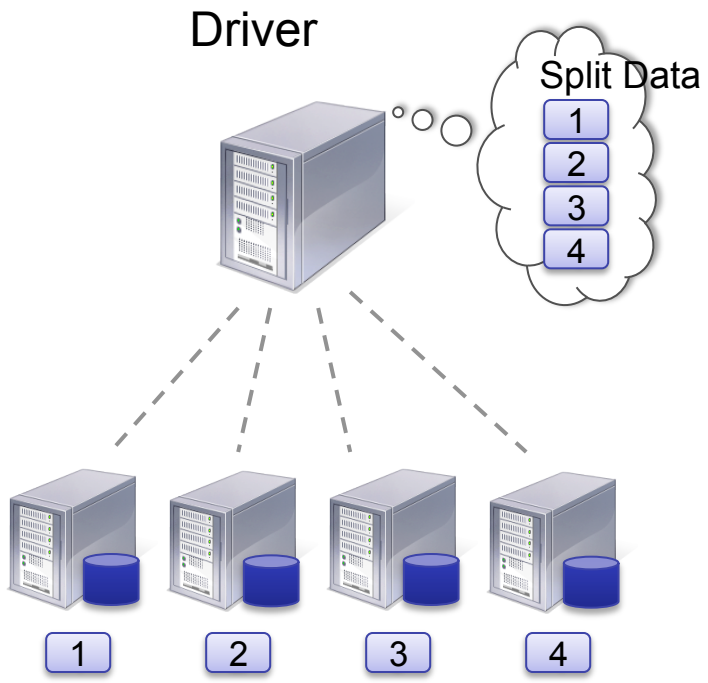
2. **Cost-efficiency:**
   - Commodity nodes (cheap, but unreliable)
   - Commodity network
   - Automatic fault-tolerance (fewer admins)
   - Easy to use (fewer programmers)

Command Line: Divide and Conquer

- **Partitions the data**
- **The framework processes the objects within a partition in sequence, and can processes multiple partitions in parallel**

Challenges?

Partition and distribute data



Slave Processors or Computers
Executors

- **Challenge**
  - State the challenge
  - Tools and rules
  - Timeout to solve

- **We are going to hack this up using a unix-based divide and conquer strategy**

- **AKA poorman's distributed computational framework**

Ground Rules

- **Limit ourselves to using the following Unix commands only:**
  - split, grep, wc, merge, cat, for, echo
  - cut, paste and bc #to get a total
  - |, &, wait #parallel computing

grep command

- **Problem: Need to search very large file**
  - Ex: Your java applet records each user that logs in to your website
    - Assume the log file generated is large (everyone wants to see your website!)
    - You want to look at only the lines in log file that contain 'mcorliss'
- **In unix, use 'grep' command**
  - 'grep <string> <filename>' returns all lines in file that contain string
    - If string contains ' ' (space) then need to enclose in quotes
    - <string> is case-sensitive: "mcorliss" != "Mcorliss"

```
prompt$ grep "mcorliss" logfile
  Jan 10 10:06:54 log in by mcorliss from 66.94.234.13
  Jan 12 11:36:22 log in by mcorliss from 65.92.231.10
                          ...
```

  - What if we want only lines with "mcorliss" on January 10?

Regular Expressions

- **Fortunately, grep supports more powerful matching**
  - Using regular expressions
  - 'grep <reg. exp.> filename'
- **For example, to solve previous problem:**

```
prompt$ grep "Jan 10.*mcorliss" logfile

Jan 10 10:06:54 log in by mcorliss from 66.94.234.13
```

- **Searching for songs whose title starts with "love"**

```
prompt$ grep "^love" musiclibrary.txt
```

- **Searching for a 7-digit phone number in a file:**

```
prompt$ grep "[0-9]\{3\}[ -]\?[0-9]\{4\}" file
```

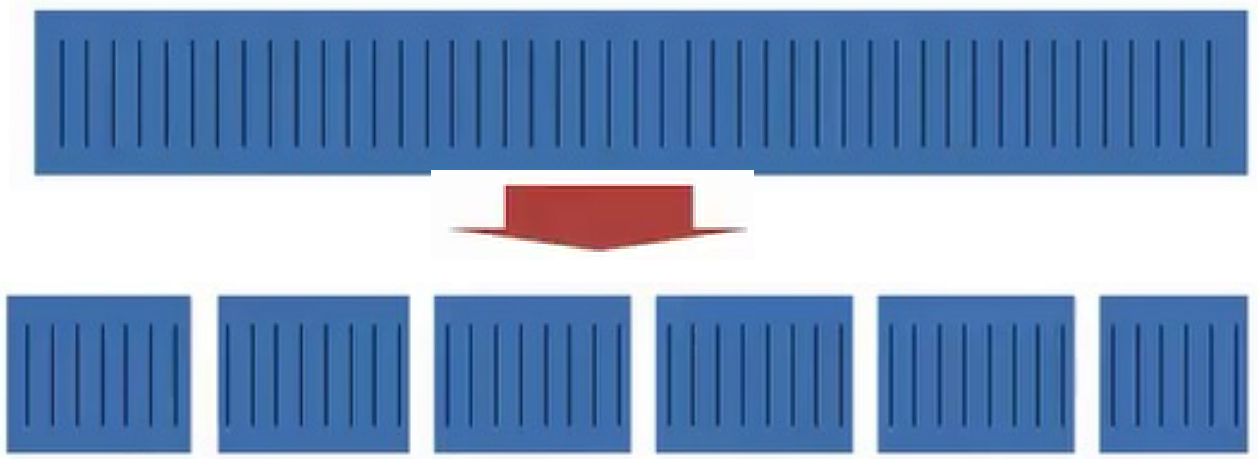
Pipe command "|": Multiple Matches

- **Regular expressions are powerful**
  - But sometimes want to do multiple independent matches
  - For example, match on lines that contain 'foo', 'goo', and 'zoo'
    - Trying to do this using regular expressions, would be very painful
- **Can use pipes denoted as | in Unix:**

```
prompt$ grep foo file | grep goo | grep zoo
```

split a large file into chunks/folds

- **split**
  - `cat logfile.txt | split -l 20 -d fold`
    divide logfile.txt into files of 20 lines apiece, using "*fold*" as the prefix and with numeric suffixes
- **split -l 30 #30 lines per file**
- **split -b 24m #24 meg chunks**



Unix “split” and “cat” command

Split the file based up on the number of lines

Split the file into multiple pieces based up on the number of lines using -l option as shown below.

```
$ wc -l testfile
2591 testfile
$ split -l 1500 testfile importantlog
$ wc -l *
  1500 importantlogaa
  1091 importantlogab
  2591 testfile
```

divide testfile into files of 1500 lines apiece, using “importantlog” as the prefix and with suffixes “aa”, “ab” in this case

Split and Cat

To split *filename* to parts each 50 MB named *partaa*, *partab*, *partac*,....

```
split -b50m filename part
```

To join the files back together again use the `cat` command

```
cat xaa xab xac > filename
```

or

```
cat xa[a-c] > filename
```

Split into chunks, wc

- **split**
  - `cat file.txt | split -l 20 -d fold`
    divide file.txt into files of 20 lines apiece, using “fold” as the prefix and with numeric suffixes
- **wc**
  - WC is a counting utility
  - `wc -[l|c|w] file.txt`
    counts number of lines, characters, or words in a file

>>>$ wc -l setup.py

271 setup.py 271 lines in the setup.py file

For loop in Unix + echo

- ```
  for f in *.txt; do
    echo "file >> $f";
   #cat`$f'| wc -l ;
  done
  ```

```
setup.py
JAMES-SHANAHANs-Desktop-Pro:mlpy-3.5.0 jshanahan$ ls
CHANGES.txt     LICENSE.txt     build           mlpy
COPYRIGHT.txt   PKG-INFO        docs            setup.py
INSTALL.txt     README.txt      gpl-3.0.txt
JAMES-SHANAHANs-Desktop-Pro:mlpy-3.5.0 jshanahan$ for f in *.txt; do    echo "$f" ; done
CHANGES.txt
COPYRIGHT.txt
INSTALL.txt
LICENSE.txt
README.txt
gpl-3.0.txt
JAMES-SHANAHANs-Desktop-Pro:mlpy-3.5.0 jshanahan$
```
```

Get total using cut, paste and bc

```
JAMES-SHANAHANs-Desktop-Pro:mlpy-3.5.0 jshanahan$ seq 10
1
2
3
4
5
6
7
8
9
10
JAMES-SHANAHANs-Desktop-Pro:mlpy-3.5.0 jshanahan$ seq 10 | paste -sd+ - | bc
55
JAMES-SHANAHANs-Desktop-Pro:mlpy-3.5.0 jshanahan$ seq 10 >tmpp
JAMES-SHANAHANs-Desktop-Pro:mlpy-3.5.0 jshanahan$ cat tmpp
1
2
3
4
5
6
7
8
9
10
JAMES-SHANAHANs-Desktop-Pro:mlpy-3.5.0 jshanahan$ cat tmpp|cut -f 1|paste -sd+ - | bc
55
JAMES-SHANAHANs-Desktop-Pro:mlpy-3.5.0 jshanahan$
```

seq generates a sequence of numbers

cat tmpp | cut -f 1 | paste -sd+ - | bc

```
JAMES-SHANAHANs-Desktop-Pro-2:Notebooks jshanahan$ bc <<< "2+2"
4
```

1+2+3….+10|bc

Parallel computing with "&" and wait

- The "&" causes parent process to spawn off parallel processes…
- And wait will cause the parent process to wait until child processes have finished

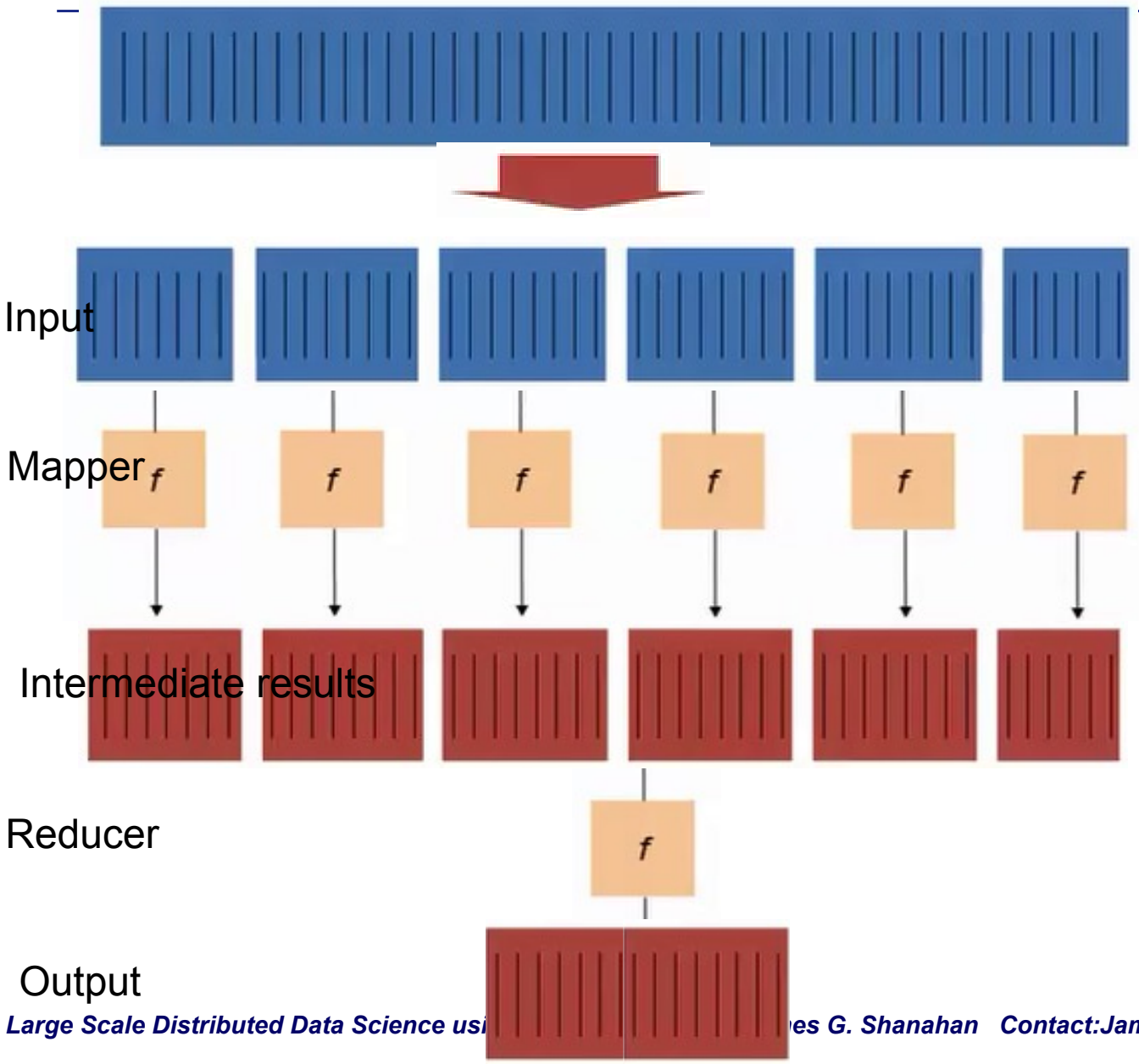
```
1 JAMES-SHANAHANs-Desktop-Pro:~ jshanahan$ seq 10000000|wc &
2 [1] 72233
3 JAMES-SHANAHANs-Desktop-Pro:~ jshanahan$ wait; echo "Finished waiting"
4  10000000 10000000 113888814
5 [1]+  Done                    seq 10000000 | wc
6 Finished waiting
7 JAMES-SHANAHANs-Desktop-Pro:~ jshanahan$
```

Child process; takes time

Prints only after waiting

- Here seq 1000000|wc & causes a the parent terminal process to spawn off a process to do this line count task.
- Meanwhile the next command "wait" causes the shell to wait until the child process (seq 1000000|wc) finishes. Once it finishes the shell can continue and run the echo command

Schematic of Parallel Processing



A big data file

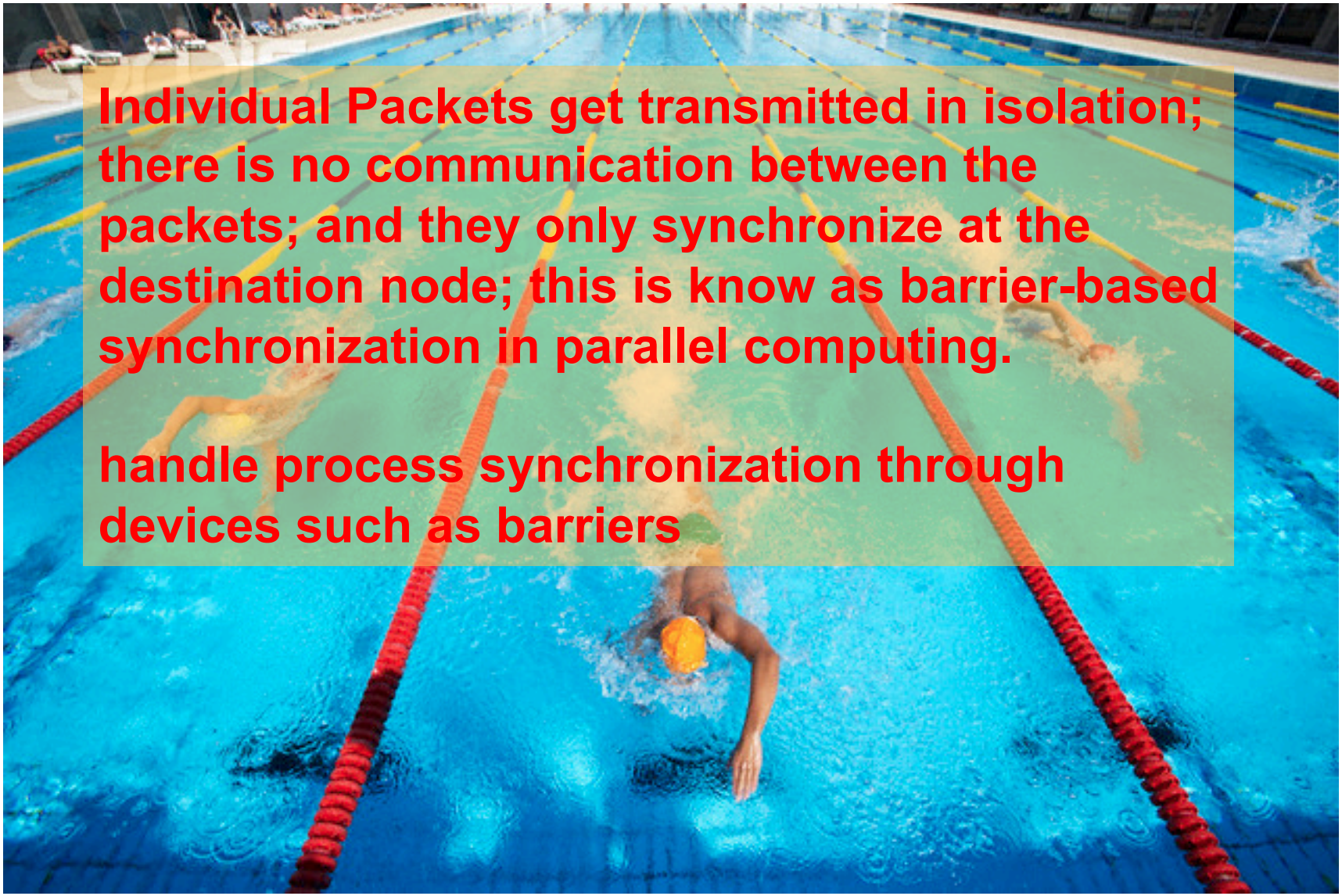
File is split into smaller chunks and distributed to k nodes

f is a function that operates on each data item

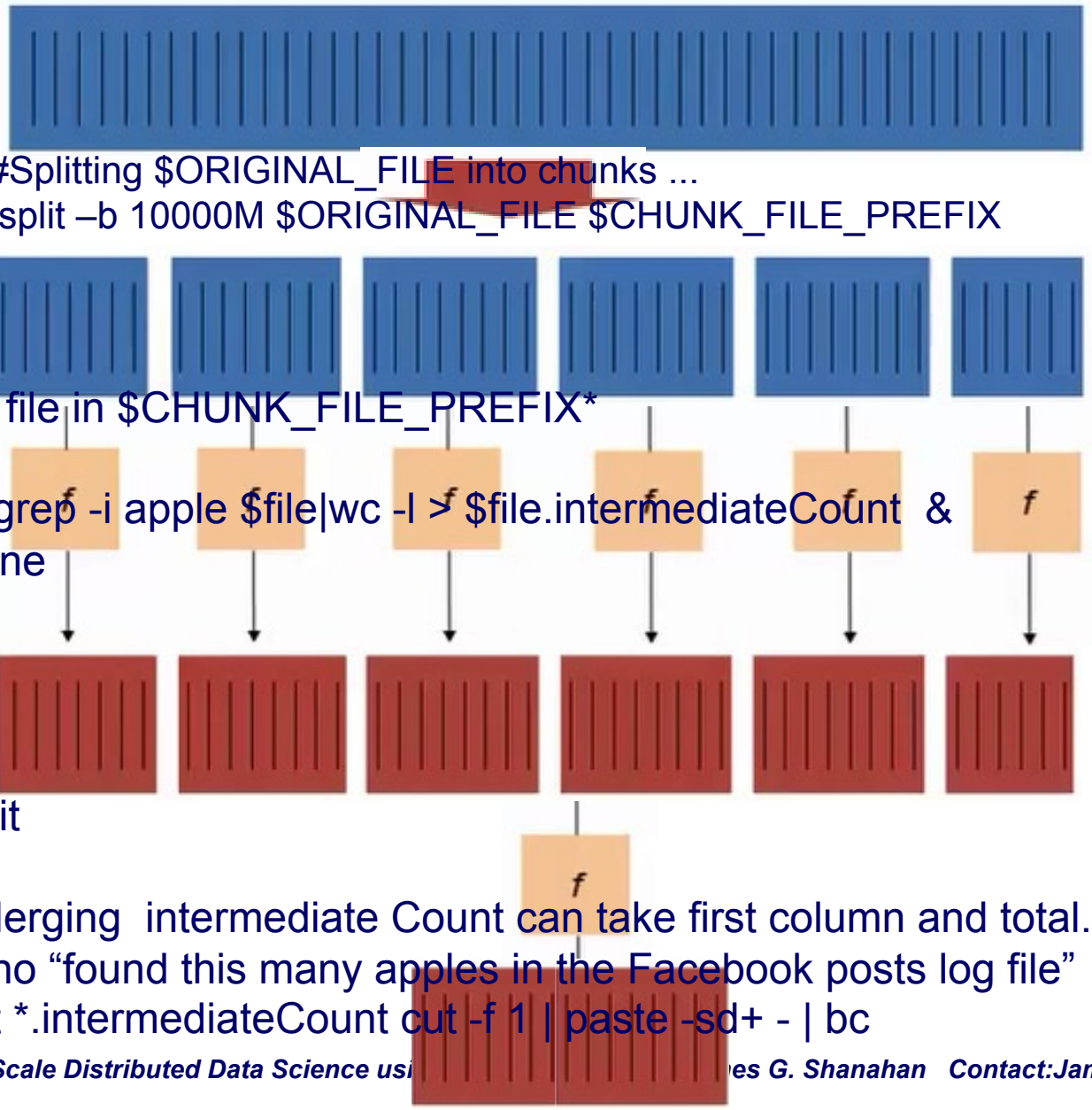
A distributed set of answers

Merge outputs

Individual Packets get transmitted in isolation



Schematic of Parallel Processing



A big data file

File is split into smaller 100Gig chunks and distributed to k nodes

f is a function that operates on each data item

A distributed set of answers

Merge outputs

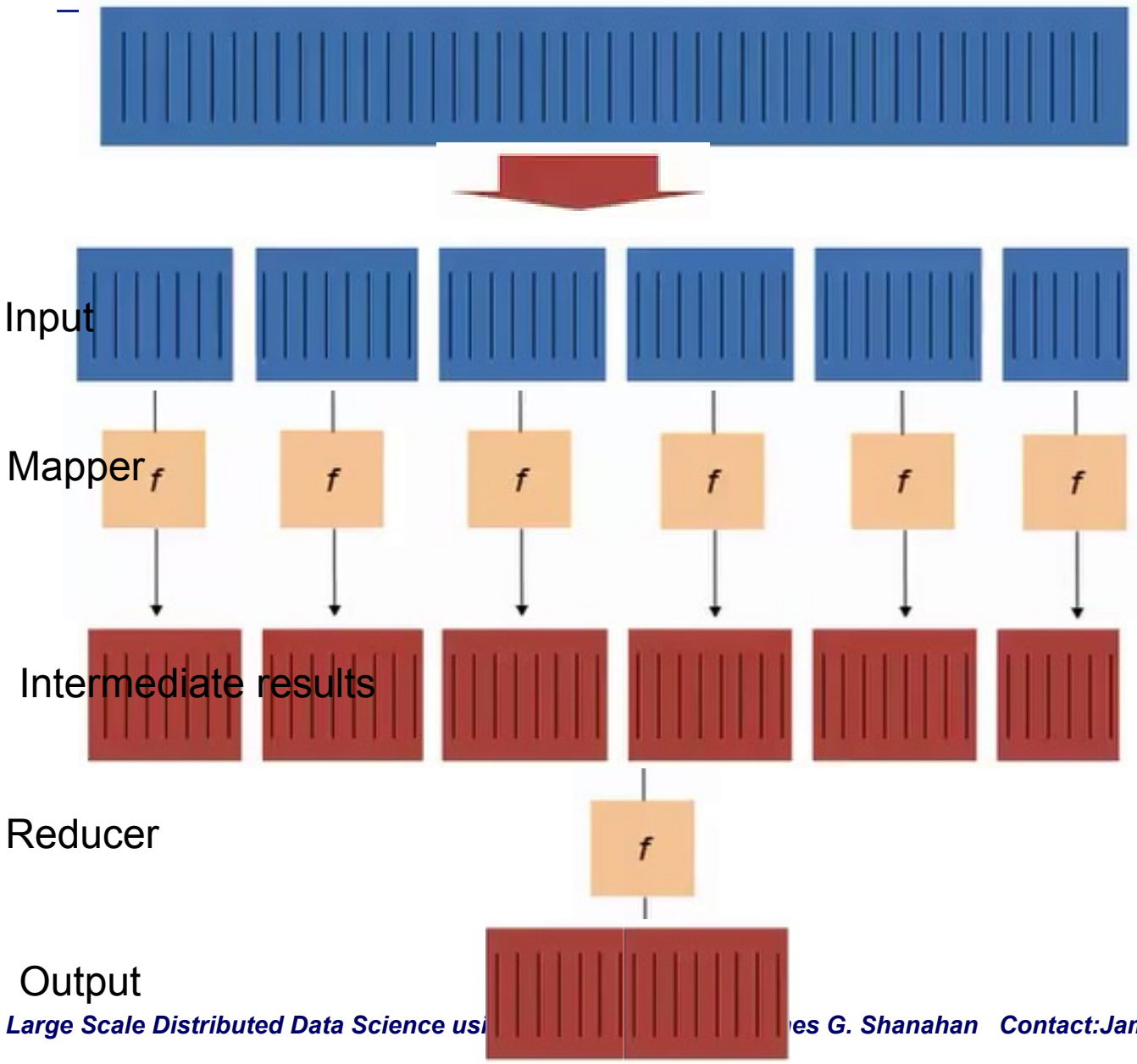
```
1. #Splitting $ORIGINAL_FILE into chunks ...
2.  split –b 10000M $ORIGINAL_FILE $CHUNK_FILE_PREFIX

for file in $CHUNK_FILE_PREFIX*
do
   grep -i apple $file|wc -l > $file.intermediateCount  &
Done

wait

#Merging  intermediate Count can take first column and total...
echo “found this many apples in the Facebook posts log file”
cat *.intermediateCount cut -f 1 | paste -sd+ - | bc
```

Schematic of Parallel Processing



A big data file

File is split into smaller chunks and distributed to k nodes

f is a function that operates on each data item

A distributed set of answers

Merge outputs

Solution

```
1. ORIGINAL_FILE=$1
2. CHUNK_FILE_PREFIX=$ORIGINAL_FILE.split.
3. SORTED_CHUNK_FILES=$CHUNK_FILE_PREFIX*.sorted
```

pGrepCount

```
4. usage ()
5. {
6.       echo Parallel grep
7.       echo usage: pGrepCount file1 100
8.       echo greps file file1 for a “apple” and counts the number of lines
9.       echo Note: file1 will be split in chunks up to 10Gig Chunks
10.      echo  and each chunk will be grepCounted in parallel
11. }
12.#   Splitting $ORIGINAL_FILE into chunks ...
13. split –b 10000M $ORIGINAL_FILE $CHUNK_FILE_PREFIX
14.#   Distribute
15.for file in $CHUNK_FILE_PREFIX*
16.do
17.    grep -i apple $file|wc -l > $file.intermediateCount  &
18.done
19.wait

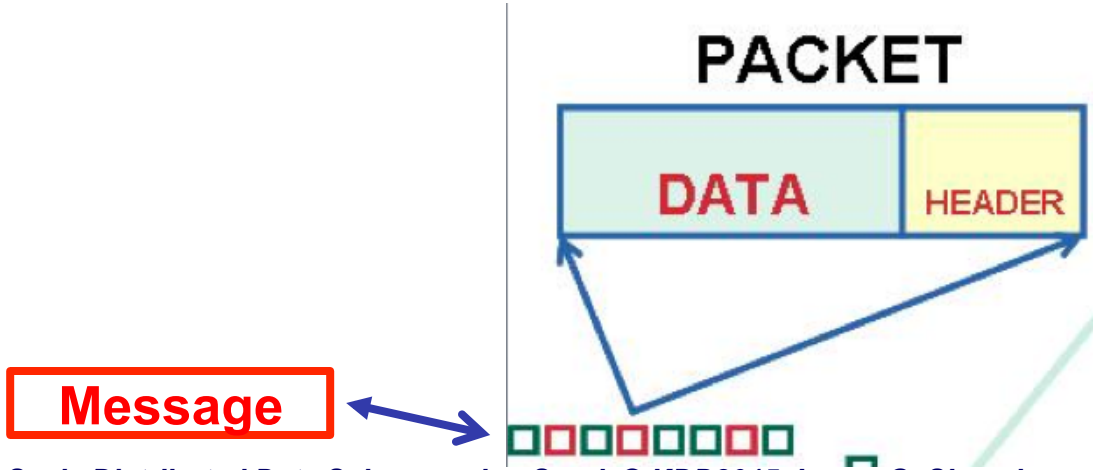
20.#  Merging  intermediate Count can take first column and total...
21.numOfInstances=$(cat *.intermediateCount cut -f 1 | paste -sd+ - | bc)
22.echo “found [$numOfInstances] of apples in the Facebook posts log file”
```

Mapper

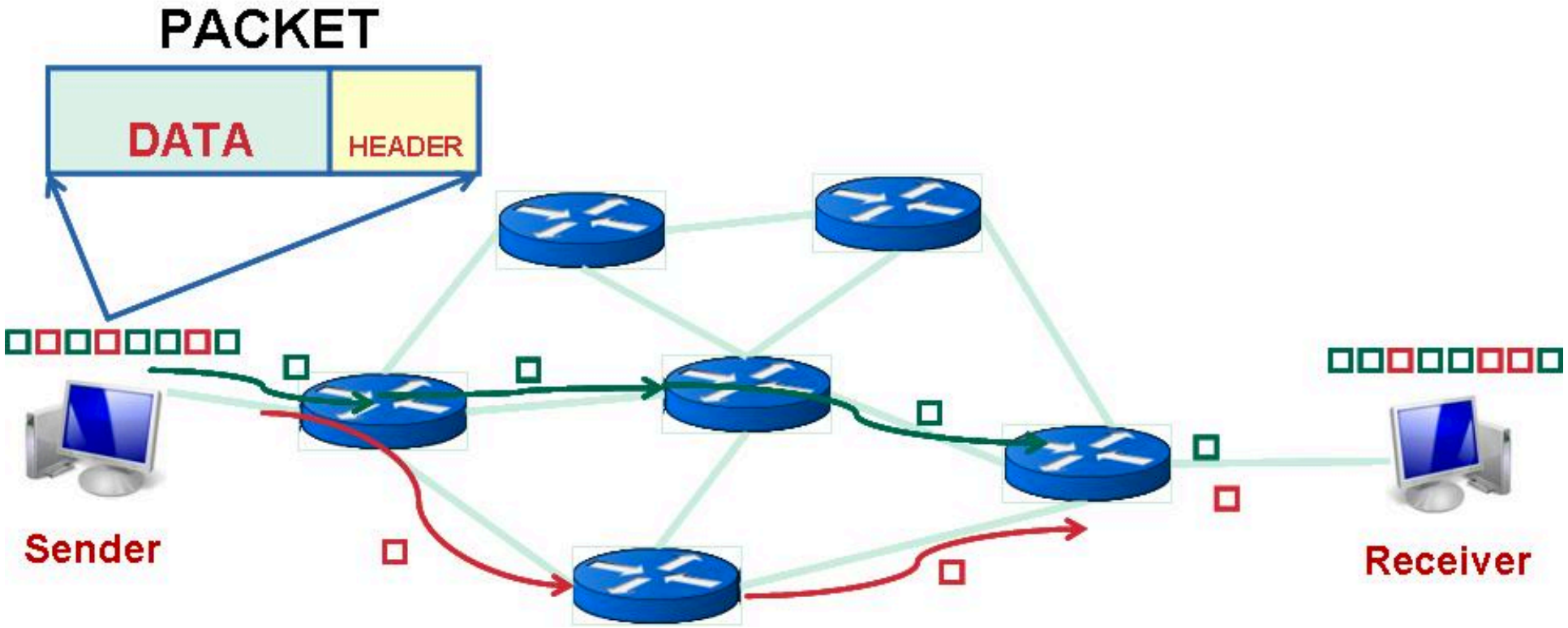
Reducer

D&C is common: e.g., Packet switching

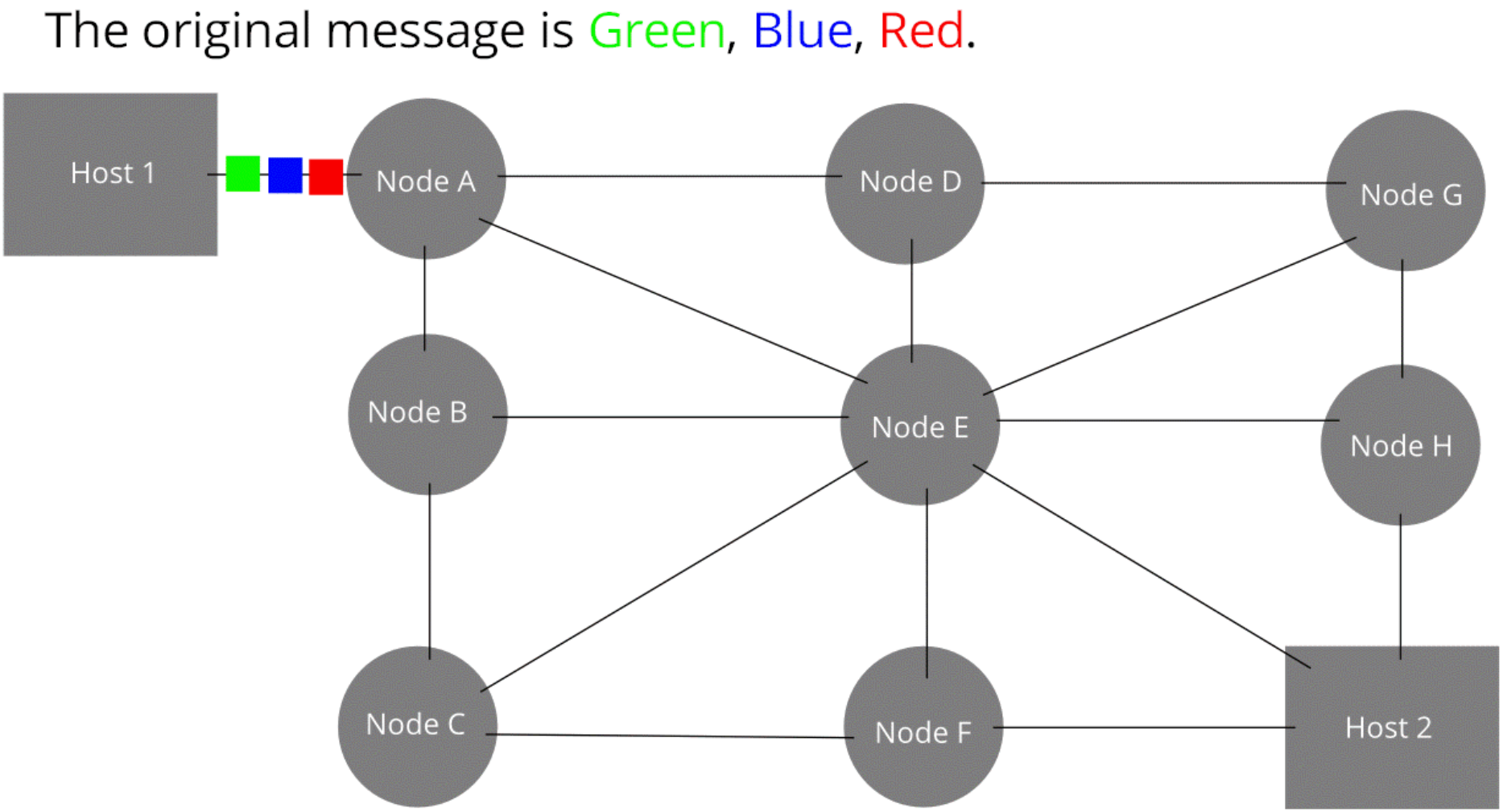
- Packet switching is a digital networking communications method that transmits messages in chunks or blocks, called *packets*
- Packets are transmitted via a medium that may be shared by multiple simultaneous communication sessions.
- Packet switching increases network efficiency, robustness and enables technological convergence of many applications operating on the same network.



Packet = Header and Payload

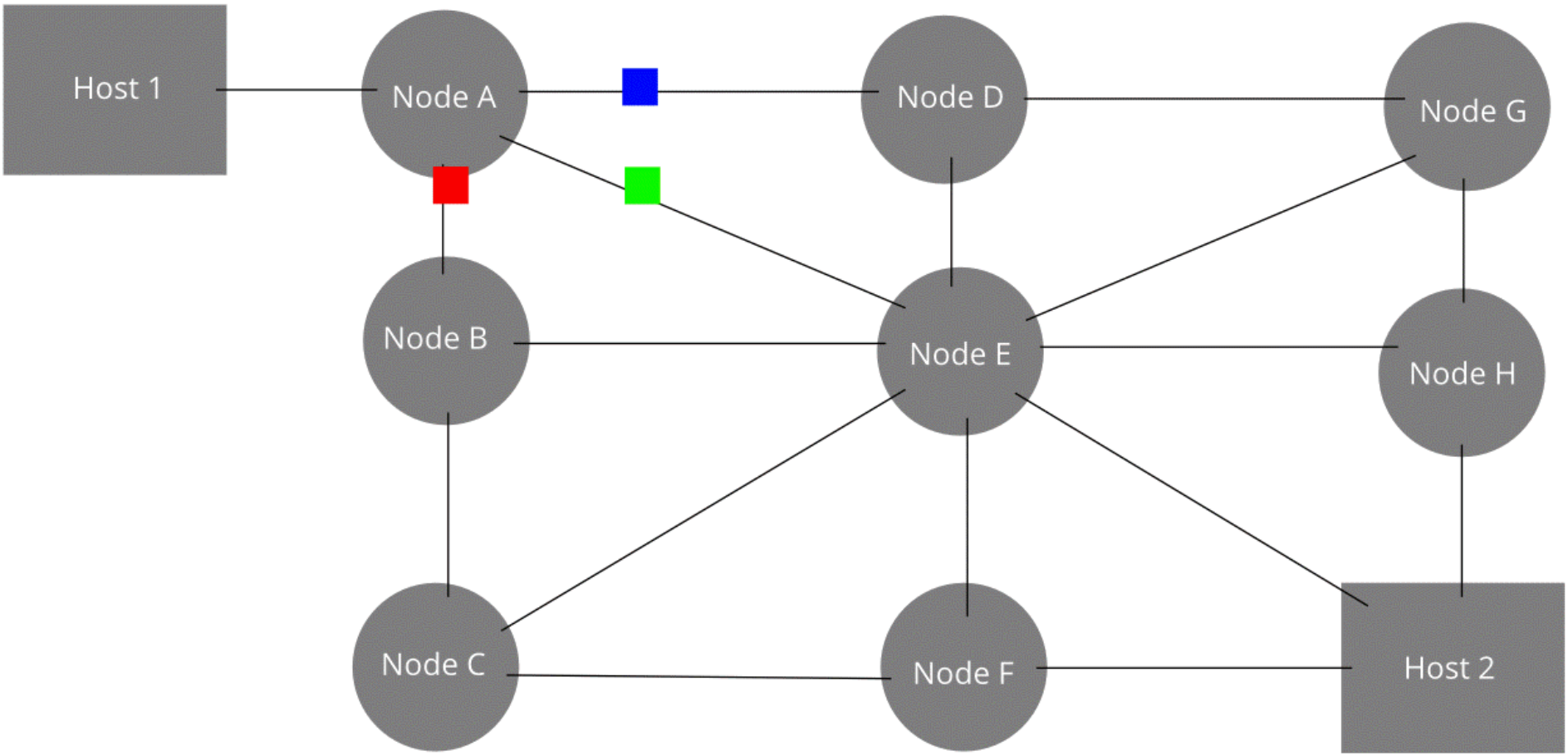


Message Passing: Divide and conquer/send



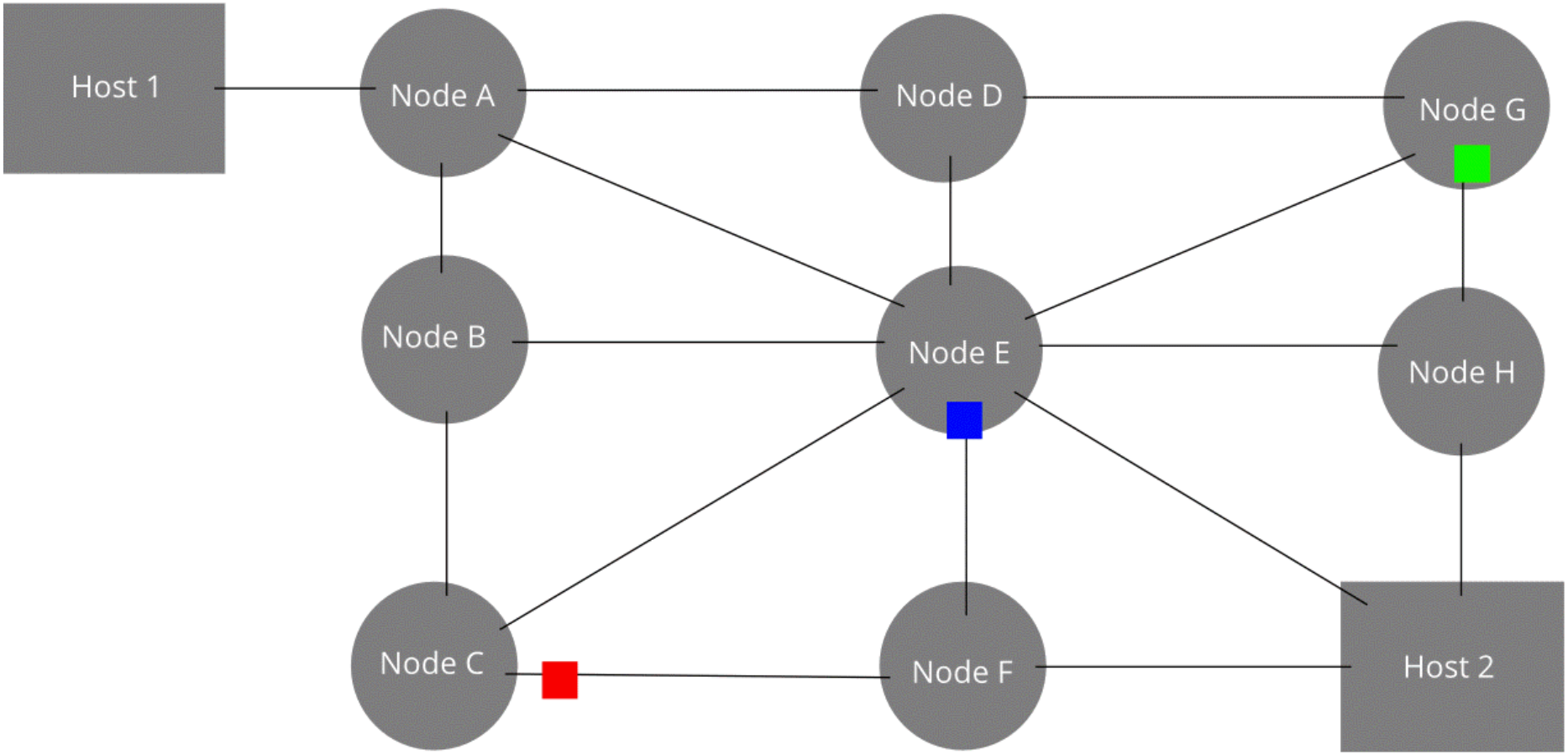
Route packets in parallel if possible

The data packets take different routes to their destinations.



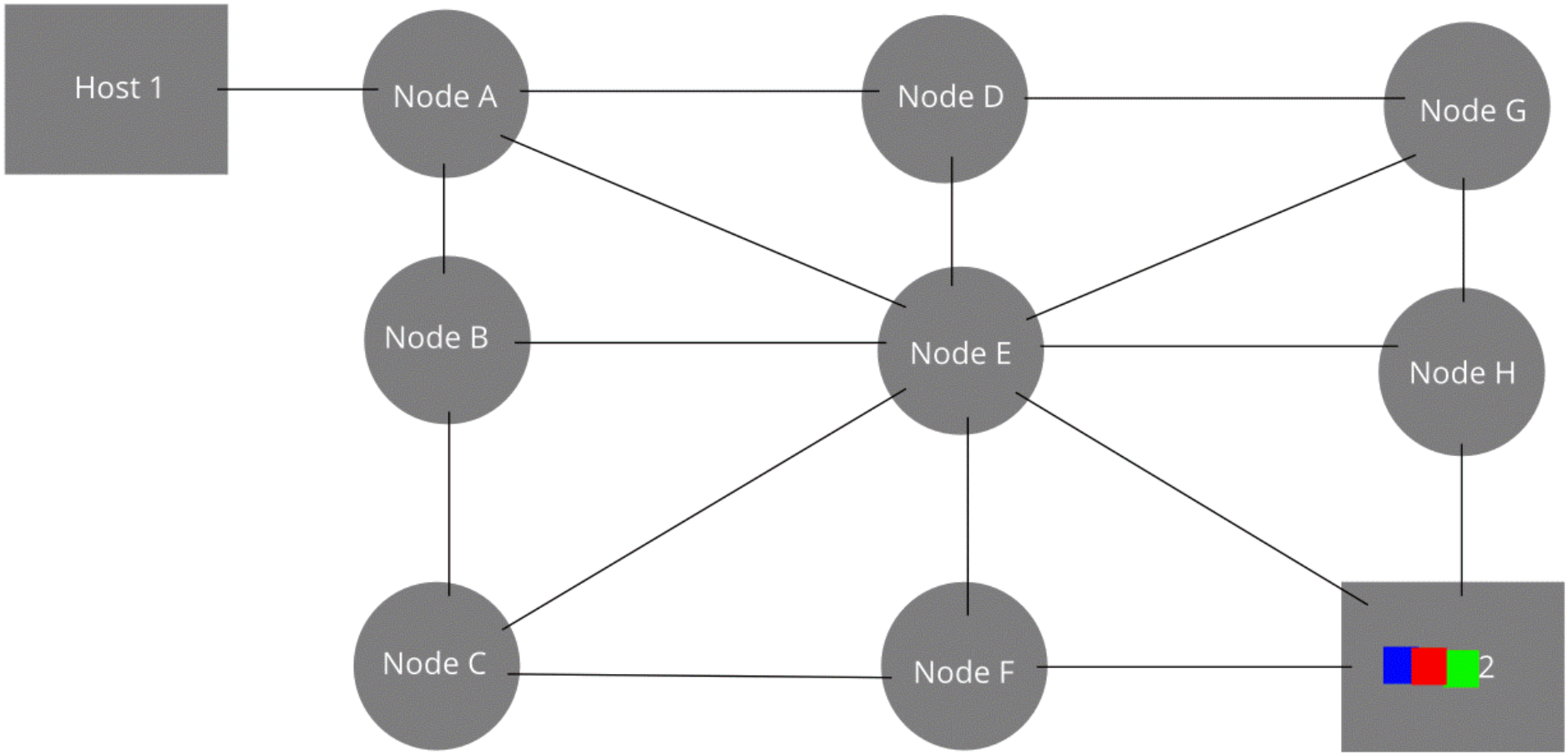
Routing packets in parallel if possible

The data packets take different routes to their destinations.



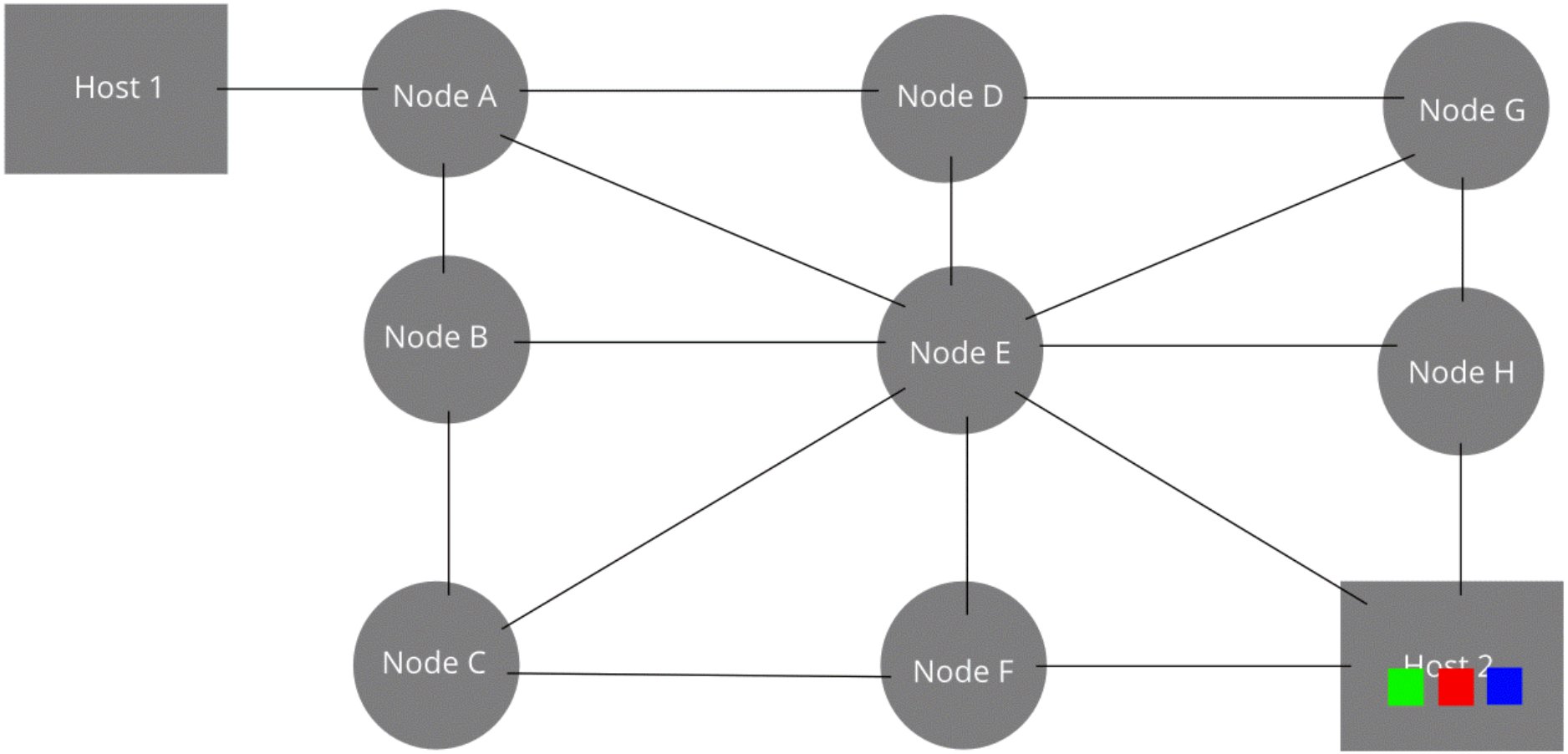
Barrier sync: wait until all packets arrive

The data packets take different routes to their destinations.

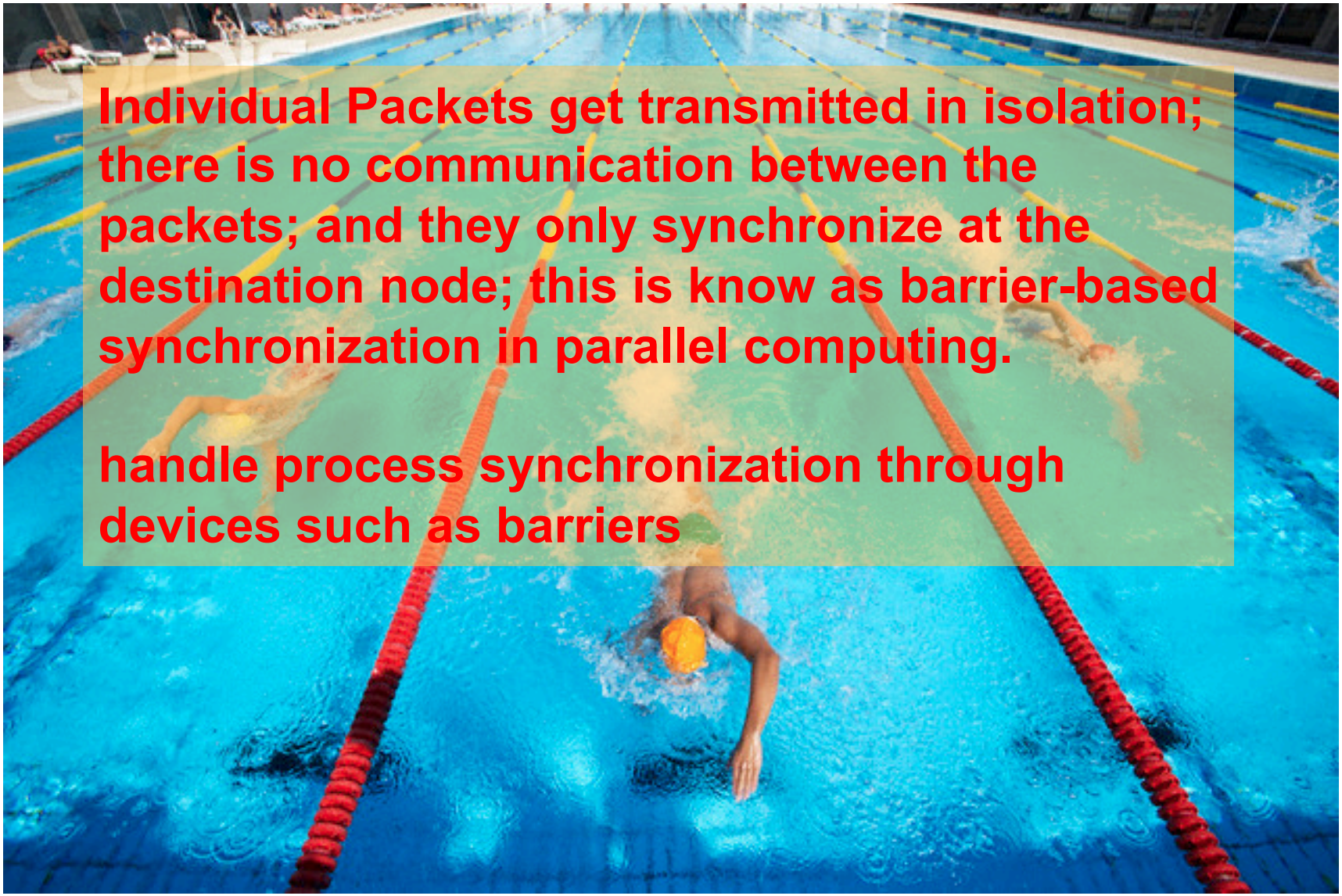


Barrier sync: reassemble payload

The received message is Green, Red, Blue



Individual Packets get transmitted in isolation



- **The key to success here was Divide and Conquer**
- **Decompose a large task into smaller ones**

- **We came up with a very nice framework for parallelizing tasks on the command line!!**
  - But it is limited
  - Granularity of task is somewhat coarse
  - No fault tolerance
  - No control over shared filespace
- **Divide and conquer does not come for free: there are obligations in terms of communication, synchronization, and fault tolerance**

Issues to be addressed

- How to break large problem into smaller problems? Decomposition for parallel processing
- How to assign tasks to workers distributed around the cluster?
- How do the workers get the data?
- How to synchronize among the workers?
- How to communicate with works?
- How to share partial results among workers?
- How to do all these in the presence of errors and hardware failures?

Divide and conquer does not come for free: there are obligations in terms of communication, synchronization, and fault tolerance

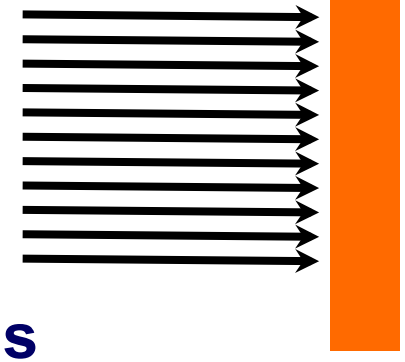
Data Parallelism – Embarrassingly Parallel Tasks

- Little or no effort is required to break up the problem into a number of parallel tasks, and there exists no dependency (or communication) between those parallel tasks.
- Examples:
  - map() function in Python:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> map(lambda e: e*e, x)
>>> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Synchronization thru a Barrier

- A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/ processes reach this barrier.
- Another popular way of syncing is the barrier method;
- Explicitly handle process synchronization through devices such as barriers
- it is pretty effective, and very coarse grained and can be great in certain types of problems.
- In parallel computing, a barrier is a type of synchronization method.



- Better than MPI and shared memory solutions

Summary: Communication/Synchronization

- **Those problems that can be decomposed into independent subtasks, requiring no communication/synchronization between the subtasks except a join or barrier at the end are very parallelizable (embarrassingly parallel)**
  - Mutex pattern
  - Join pattern
  - Barrier pattern

Categories of Parallel Computation Tasks

- **Applications are often classified according to how often their subtasks need to synchronize or communicate with each other.**
- **Fine-grained parallelism**
  - An application exhibits fine-grained parallelism if its subtasks must communicate many times per second; (share memory programming)
- **Coarse-grained parallelism**
  - It exhibits coarse-grained parallelism if they do not communicate many times per second,
- **Embarrassingly parallel**
  - An application is embarrassingly parallel if it rarely or never has to communicate. Divide a conquer. Summing a list of numbers.
  - Such applications are considered the easiest to parallelize.
  - Can be realized on a shared nothing architecture (see this shortly)

Examples of embarrassingly parallel problems

- **An application is embarrassingly parallel if it rarely or never has to communicate**
- **Applications**
  - Summing a list of numbers
  - Matrix multiplication
  - Lots of machine learning algorithms
    - Tree growth step of the random forest machine learning technique.
    - Genetic algorithms and other evolutionary computation metaheuristics.
  - Serving static files on a webserver to multiple users at once.
  - Computer simulations comparing many independent scenarios, such as climate models.
  - Ensemble calculations ofnumerical weather prediction.
  - Discrete Fourier Transform where each harmonic is independently calculated.
  - Many many more….

Not everything is a MR

- MRs are ideal for "embarrassingly parallel" problems
- Very little communication
- Easily distributed
- Linear computational flow

- Happy to report that most of the machine learning algorithms of practical importance are embarrassingly parallel

First generation MapReduce

- **0th Generation**
  - Command line
- **1st Generation**
  - MapReduce, Hadoop
  - Native versus Streaming
- **2nd Generation**
  - MrJob, Scalding
  - Write Map-Reduce pipelines
- **3rd Generation**
  - Spark
  - Memory backed, Rich API (80 calls)

Tutorial Outline

- **Part 1: Introduction**
  - Welcome Survey
  - Install Spark
  - Background and motivation
- **Part 2: Spark Intro and basics**
  - fundamental Spark concepts, including Spark Core, data frames, the Spark Shell, Spark Streaming, Spark SQL and vertical libraries such as MLlib and GraphX;
- **Part 3: Machine learning in Spark**
  - will focus on hands-on algorithmic design and development with Spark developing algorithms from scratch such as linear regression, logistic regression, graph processing algorithms such as pagerank/shortest path, *etc.*
- **Part 4: Wrap up**
  - Spark 1.5 and beyond
  - Summary

Part 2: Spark Intro and Basics

- **Part 2: Spark Intro and basics**
  - Base RDD
  - Fault tolerance (and lineage)
  - Transformations and Actions
  - Persistence
  - Animated Example
  - Pair RDDs
  - Word count example

- **Traditional distributed data processing frameworks have limited APIs such as Map/ Reduce**
- **Spark is a much more expressive API (over 80 functions)**

Spark: A developer's perspective

- **At a high level, every Spark application consists of a driver program that runs the user's main function and executes various parallel operations on a cluster.**
- **The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel.**
  - RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it.
  - Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations.
  - Finally, RDDs automatically recover from node failures.

Resilient distributed dataset (RDD)

- An RDD is simply a distributed collection of elements (Key-Value records).
- In Spark all work is expressed as either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result.

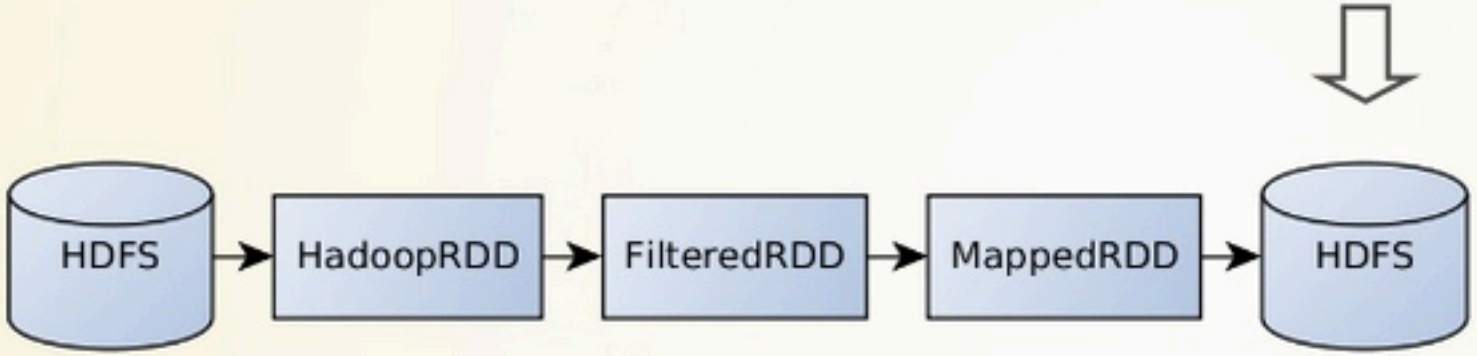
- Under the hood, Spark automatically distributes the data contained in RDDs across your cluster and parallelizes the operations you perform on them.

RDD: Collection of values: Map/Filter/Reduce

Key	Value
d1	the quick brown fox
d2	the fox ate the mouse
d3	how now brown cow

Step by Step

RDD: each row is a key, value pair
Focus on simple RDDs in this section, i.e., value only RDDs
Next section talk about RDDs of key/value pairs



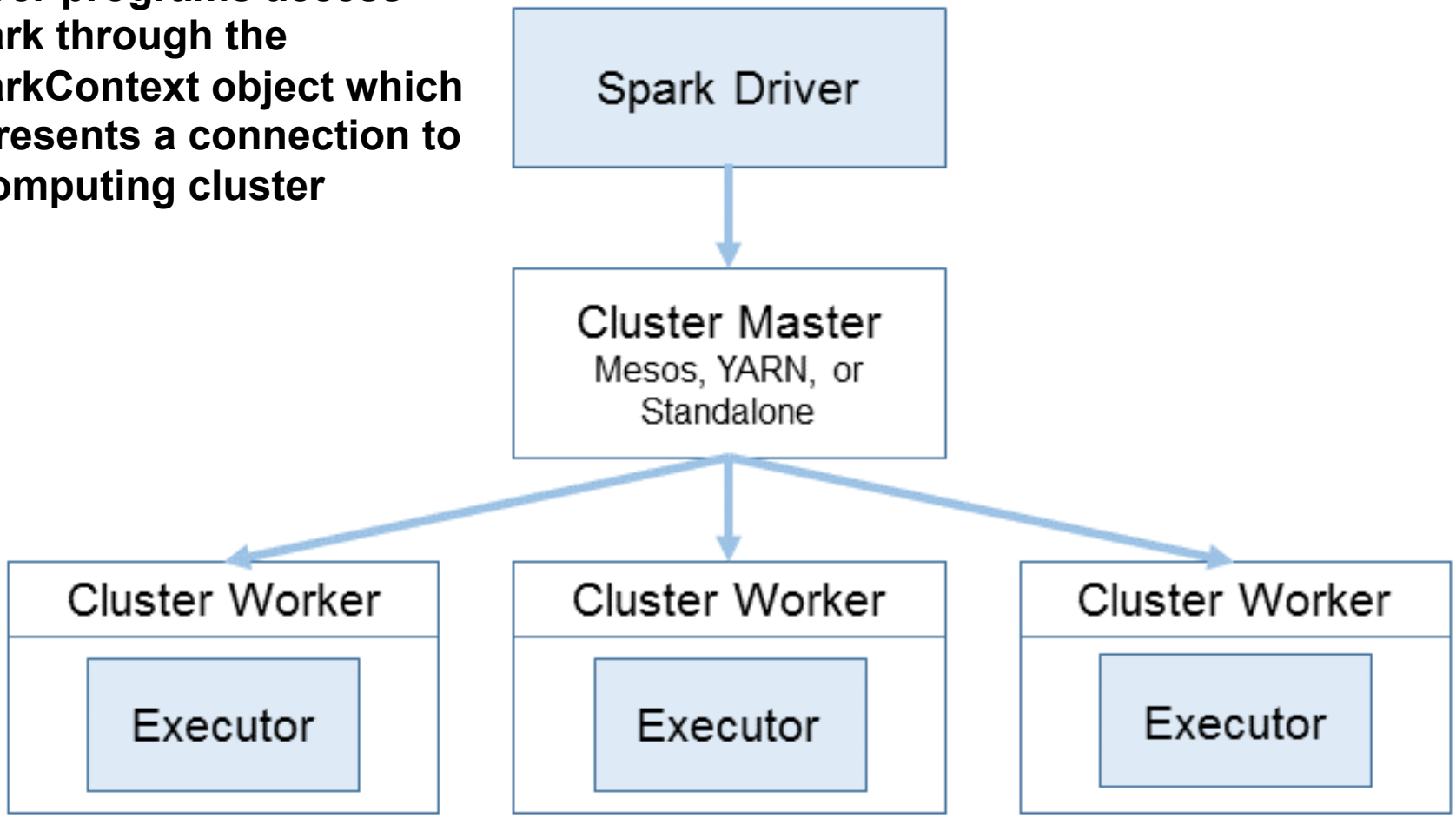
The RDD.saveAsTextFile() action triggers a job. Tasks are started on scheduled executors.

Spark Programming Interface

- **Spark has APIs available in**
  - Scala
  - Java
  - Python
  - R, SQL
- **A lot of Spark's API revolves around passing functions to its operators to run then on the cluster (ships codes to executors)**
- **Provides:**
  - Resilient distributed datasets (RDDs)
    - Partitioned collections with controllable caching, built in fault tolerance
  - Operations on RDDs
    - Transformations (define RDDs), actions (compute results)
  - Restricted shared variables (broadcast, accumulators)
- **Goal: make parallel programs look like local ones**

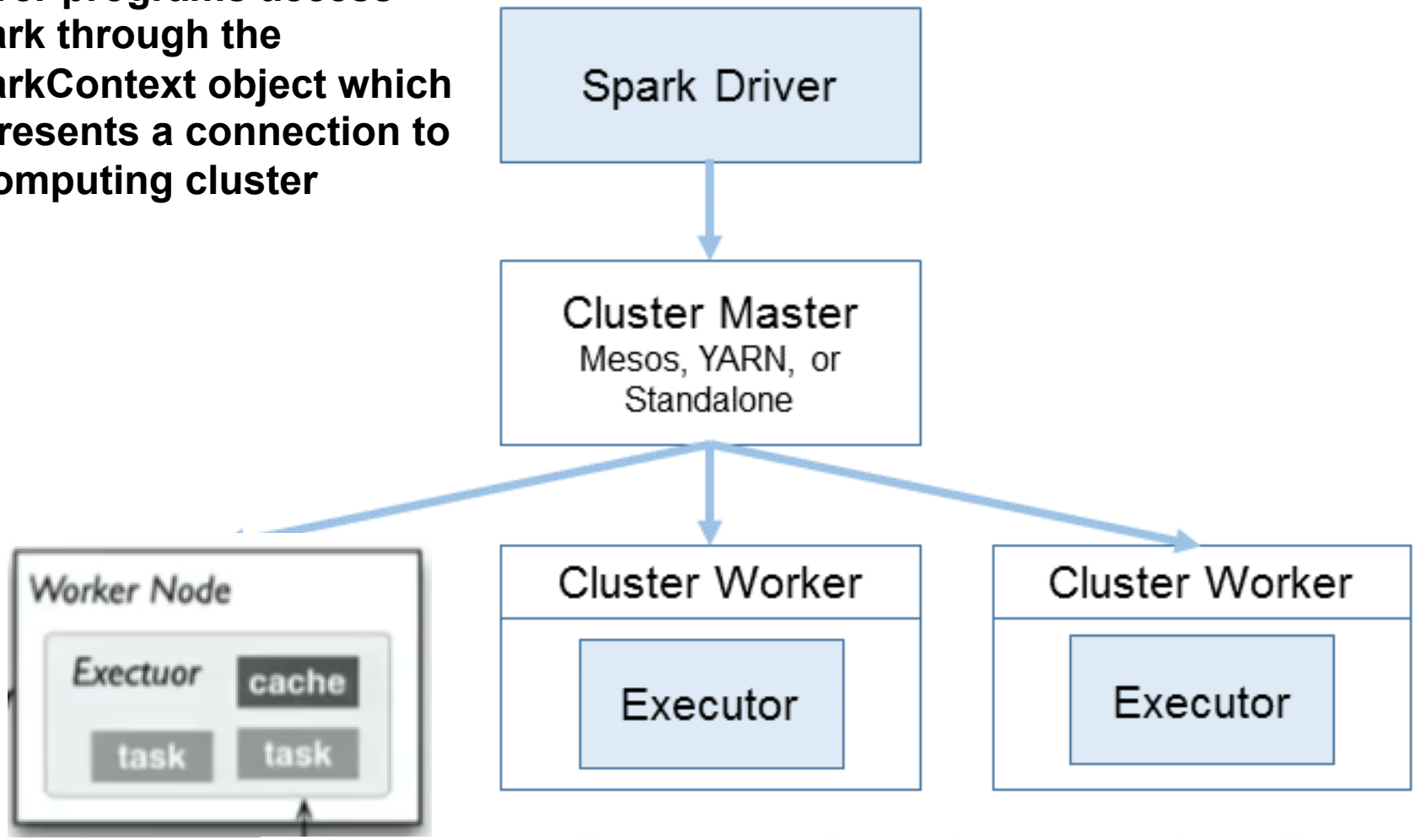
Spark Cluster

Driver programs access Spark through the SparkContext object which represents a connection to a computing cluster



Spark Cluster

Driver programs access Spark through the SparkContext object which represents a connection to a computing cluster



spark.apache.org/docs/latest/cluster-overview.html

Spark Essentials: *Master*

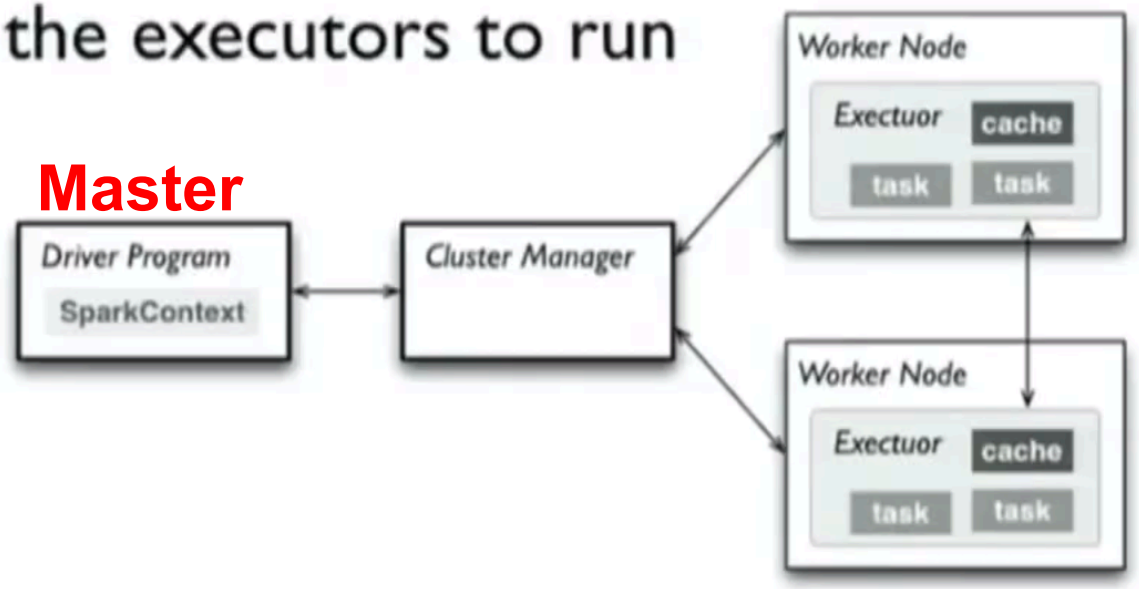
The `master` parameter for a `SparkContext` determines which cluster to use

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*

Driver programs access Spark through the SparkContext object which represents a connection to a computing cluster

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 **(serializes code and data)**
4. sends *tasks* for the executors to run



Spark Essentials: *RDD*

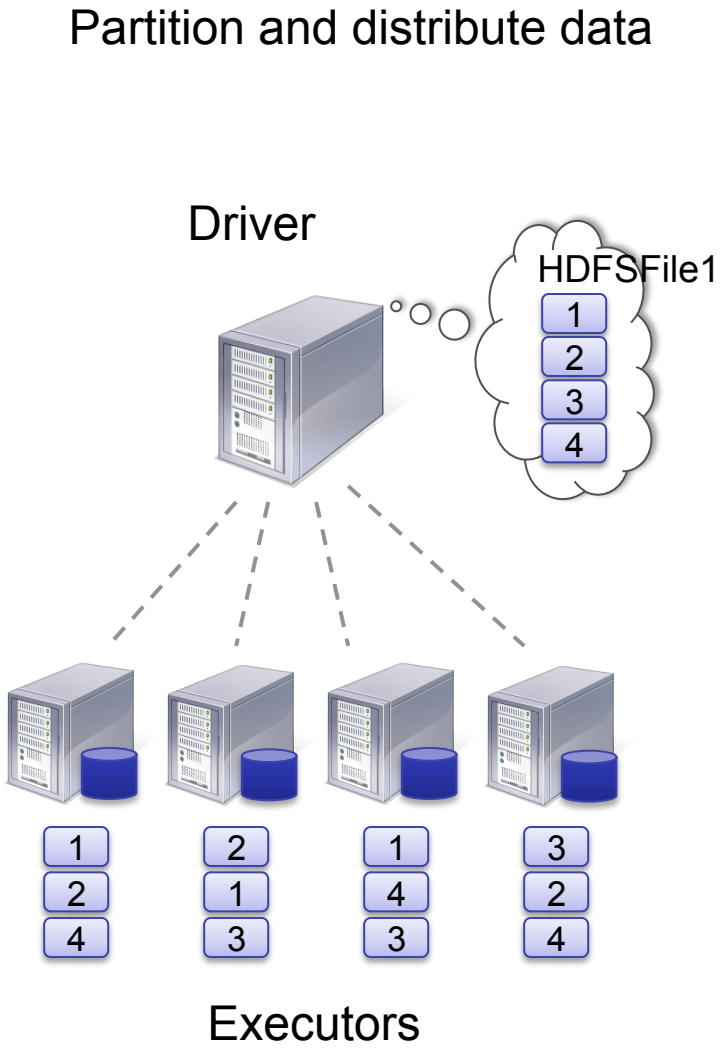
Resilient **D**istributed **D**atasets (RDD) are the primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel

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

RDD is a distributed collection of elements

- **In Spark all work is expressed as either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result.**
- **An RDD is laid out across a cluster of machines as collection of Partitions, each including a subset of the data**
- **The framework processes the objects within a partition in sequence, and processes multiple partitions in parallel.**
- **Data (e.g., clicks, or record linkage) is stored in a text file, with one observation on each line.**
  - JSON, zipped, AVRO, Parquet



Mapper: Create RDD and ship to cluster; then apply mapper (X+1)

```
In [19]: #RDD mapper
         sc.parallelize([1,2,3,4]).map(lambda x: x+1).collect()

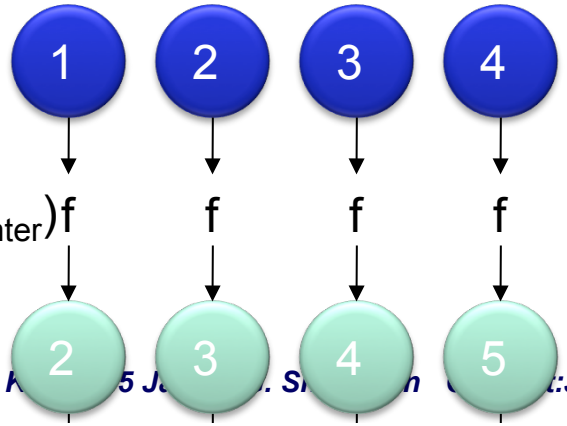
         #returns [2, 3, 4, 4]

Out[19]: [2, 3, 4, 5]
```

Distribute | Map | Gather back to driver

- **Spark is oriented around *Key-Value* records**
  - plusOneData= Apply(Mapper=f(x)=x × 1, input=X)
- **Map function:**

$Map(Key_{in}, Value_{in}) \rightarrow list(K_{inter}, V_{inter})$



f(x)=x × 1

RDD with 4 elements

Distribute

Map

Gather back to driver

```
In [19]: #RDD mapper
         sc.parallelize([1,2,3,4]).map(lambda x: x+1).collect()

         #returns [2, 3, 4, 4]

Out[19]: [2, 3, 4, 5]
```

```
In [19]: #RDD mapper
         sc.parallelize([1,2,3,4]).map(lambda x: x+1).collect()

         #returns [2, 3, 4, 4]

Out[19]: [2, 3, 4, 5]
```

```
In [33]: def powerOfTwo(x):
             return x*x

         def plusOne(x):
             return x+1

         def myReduce(x, y):
             return x+y

         print "Reduce by lambda:   %d" % sc.parallelize([1,2,3,4]).map(powerOfTwo).map(plusOne).reduce(lambda x, y: x + y)
         print "Reduce by fucntion: %d" % sc.parallelize([1,2,3,4]).map(powerOfTwo).map(plusOne).reduce(myReduce)
         #.reduce()
```

python anonymous function vs. top level function

- **Arguments to Map/Reduce operators are closures and can be python anonymous functions (Lambdas) or any top level function**

Find the line with most words

RDD actions and transformations can be used for more complex computations. Let's say we want to find the line with the most words:

Scala | Python

```
>>> textFile.map(lambda line: len(line.split())).reduce(lambda a, b: a if (a > b) else b)
15
```

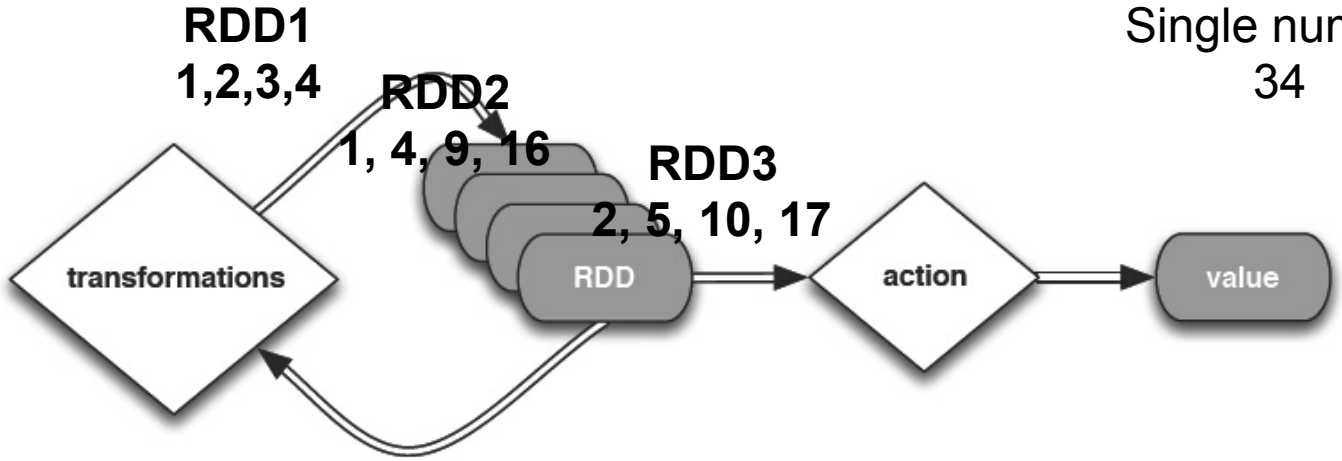
This first maps a line to an integer value, creating a new RDD. `reduce` is called on that RDD to find the largest line count. The arguments to `map` and `reduce` are Python anonymous functions (lambdas), but we can also pass any top-level Python function we want. For example, we'll define a `max` function to make this code easier to understand:

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*`)

sc.parallelize([1,2,3,4]).map(powerOfTwo).map(plusOne).reduce(lambda x, y: x + y)



```
# Create an RDD from a local text file

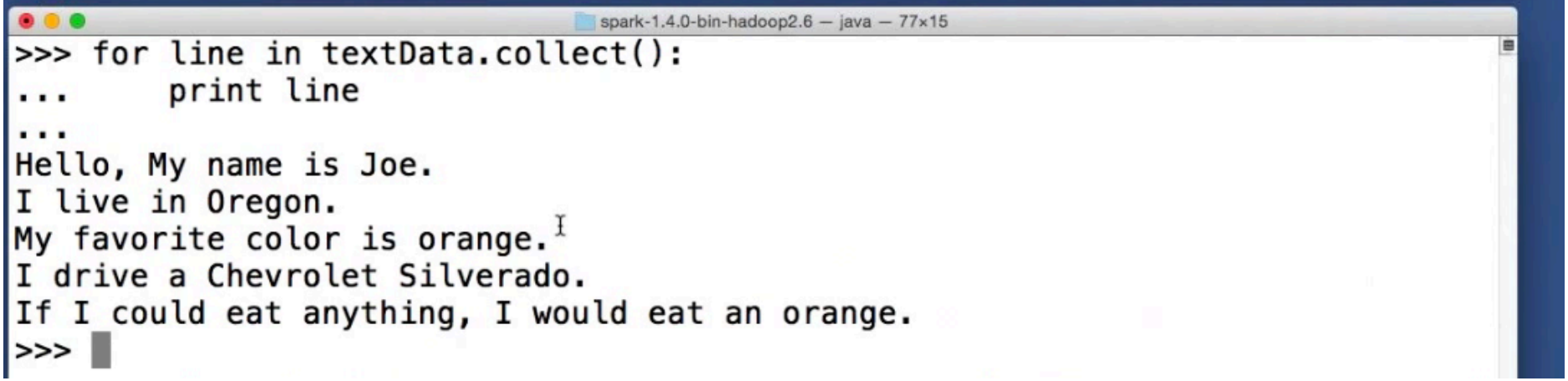
textData = sc.textFile("textData.txt")

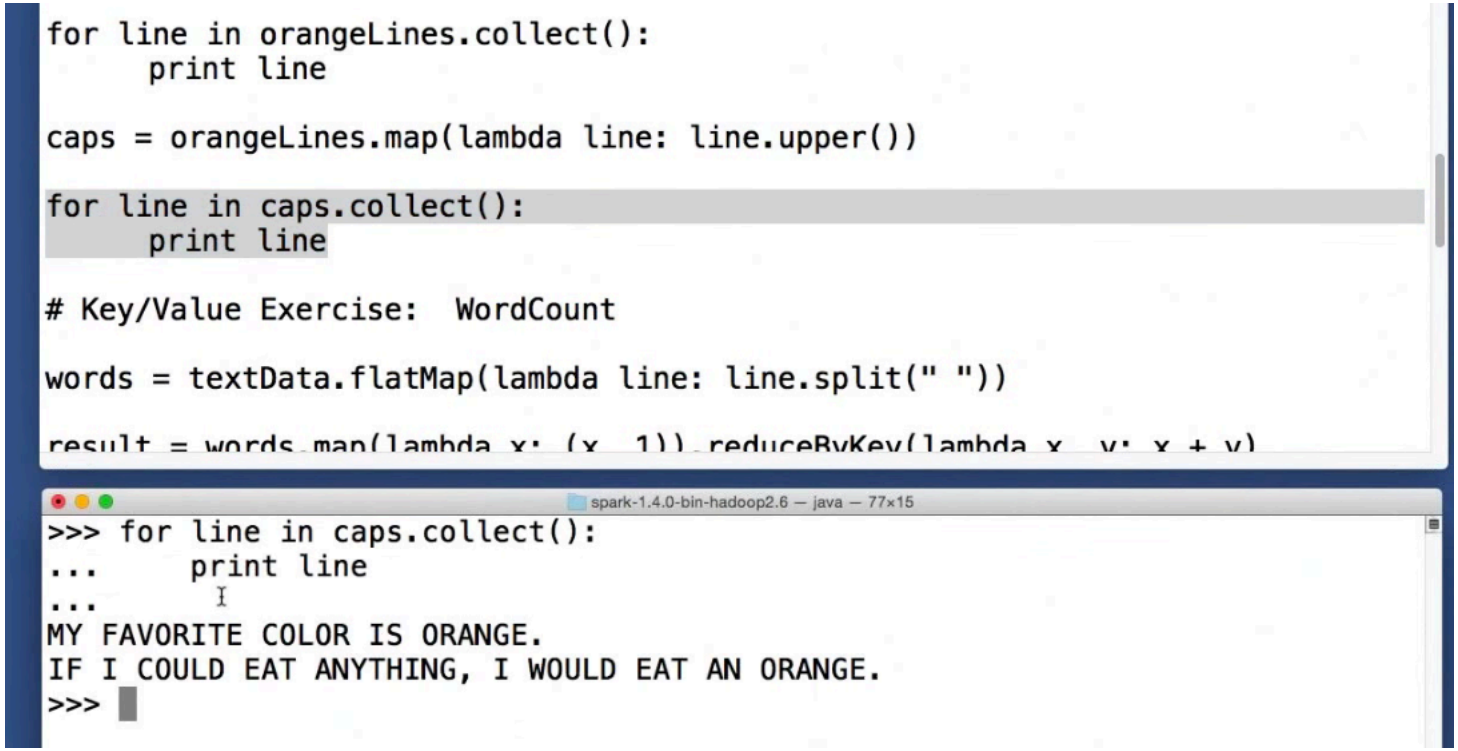
# View the contents of the RDD

for line in textData.collect():
    print line

# Lazily filter any lines that contain the word "orange"

orangeLines = textData.filter(lambda line: "orange" in line)
```



- 
for line in orangeLines.collect():
print line
caps = orangeLines.map(lambda line: line.upper())
for line in caps.collect():
print line
# Key/Value Exercise: WordCount
words = textData.flatMap(lambda line: line.split(" "))
result = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x + y)
spark-1.4.0-bin-hadoop2.6 — java — 77×15
>>> for line in caps.collect():
... print line
...
MY FAVORITE COLOR IS ORANGE.
IF I COULD EAT ANYTHING, I WOULD EAT AN ORANGE.
>>>

```
from pyspark.mllib.classification import NaiveBayes, NaiveBayesModel
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.regression import LabeledPoint

def parseLine(line):
    parts = line.split(',')
    label = float(parts[0])
    features = Vectors.dense([float(x) for x in parts[1].split(' ')])
    return LabeledPoint(label, features)

data = sc.textFile('sample_naive_bayes_data.txt').map(parseLine)

# Split data aproximately into training (60%) and test (40%)
training, test = data.randomSplit([0.6, 0.4], seed = 0)

# Train a naive Bayes model.
model = NaiveBayes.train(training, 1.0)

# Make prediction and test accuracy.
predictionAndLabel = test.map(lambda p : (model.predict(p.features), p.label))
accuracy = 1.0 * predictionAndLabel.filter(lambda (x, v): x == v).count() / test.count()

# Save and load model
model.save(sc, "myModelPath")
sameModel = NaiveBayesModel.load(sc, "myModelPath")
```

```
test.collect()
```

```
[LabeledPoint(0.0, [1.0,0.0,0.0]),
 LabeledPoint(1.0, [0.0,2.0,0.0]),
 LabeledPoint(2.0, [0.0,0.0,2.0])]
```

```
for record in test.collect():
    print record.label, record.features
```

```
0.0 [1.0,0.0,0.0]
```

Example Mappers and Reducers

```
def powerOfTwo(x):
    return x*x

def plusOne(x):
    return x+1

def myReduce(x, y):
    return x+y

print "Reduce by lambda:   %d" %
sc.parallelize([1,2,3,4]).map(powerOfTwo).map(plusOne).reduce(lambda x, y: x + y)

print "Reduce by fucntion: %d" %
sc.parallelize([1,2,3,4]).map(powerOfTwo).map(plusOne).reduce(myReduce)
#.reduce()
```

```
Reduce by lambda:   34
Reduce by fucntion: 34
```

Create RDDs

```
./bin/pyspark
```

Spark's primary abstraction is a distributed collection of items called a Resilient Distributed Dataset (RDD). RDDs can be created from Hadoop InputFormats (such as HDFS files) or by transforming other RDDs. Let's make a new RDD from the text of the README file in the Spark source directory:

```
>>> textFile = sc.textFile("README.md")
```

RDDs have *actions*, which return values, and *transformations*, which return pointers to new RDDs. Let's start with a few actions:

```
>>> textFile.count() # Number of items in this RDD
126

>>> textFile.first() # First item in this RDD
u'# Apache Spark'
```

Now let's use a transformation. We will use the `filter` transformation to return a new RDD with a subset of the items in the file.

```
>>> linesWithSpark = textFile.filter(lambda line: "Spark" in line)
```

We can chain together transformations and actions:

```
>>> textFile.filter(lambda line: "Spark" in line).count() # How many lines contain "Spark"?
15
```

Self-Contained Applications in Python (4 threads)

As an example, we'll create a simple Spark application, `SimpleApp.py`:

```
"""SimpleApp.py"""
from pyspark import SparkContext

logFile = "YOUR_SPARK_HOME/README.md"  # Should be some file on your system
sc = SparkContext("local", "Simple App")
logData = sc.textFile(logFile).cache()

numAs = logData.filter(lambda s: 'a' in s).count()
numBs = logData.filter(lambda s: 'b' in s).count()

print "Lines with a: %i, lines with b: %i" % (numAs, numBs)
```

This program just counts the number of lines containing 'a' and the number containing 'b' in a text file. Note that you'll need to replace YOUR_SPARK_HOME with the location where Spark is installed. As with the Scala and Java examples, we use a SparkContext to create RDDs. We can pass Python functions to Spark, which are automatically serialized along with any variables that they reference. For applications that use custom classes or third-party libraries, we can also add code dependencies to `spark-submit` through its `--py-files` argument by packaging them into a .zip file (see `spark-submit --help` for details). `SimpleApp` is simple enough that we do not need to specify any code dependencies.

We can run this application using the `bin/spark-submit` script:

```
# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit \
  --master local[4] \
  SimpleApp.py
...
Lines with a: 46, Lines with b: 23
```

RDDs in More Detail

- **An RDD is an immutable, partitioned, logical collection of records**
  - Need not be materialized, but rather contains information to rebuild a dataset from stable storage or computer instructions (e.g., generate 1M random numbers)
  - **Materialize RDDs** through actions, e.g., count or reduce
- **Partitioning can be based on a key in each record (using hash or range partitioning)**
- **Built using bulk transformations on other RDDs**
- **Can be cached for future reuse (or rebuilt if not cached and required again)**

Two types of RDD Operations

- **TRANSFORMATIONS: Think Map (take an RDD as input and produce an RDD); LAZY**
- **ACTIONS: E.g., Reduce.**
  - take an RDD as input and
  - return a result to the driver or write it to storage
  - AND kick of a computation

- **A transformed RDD gets (re)computed when an action is run on it**

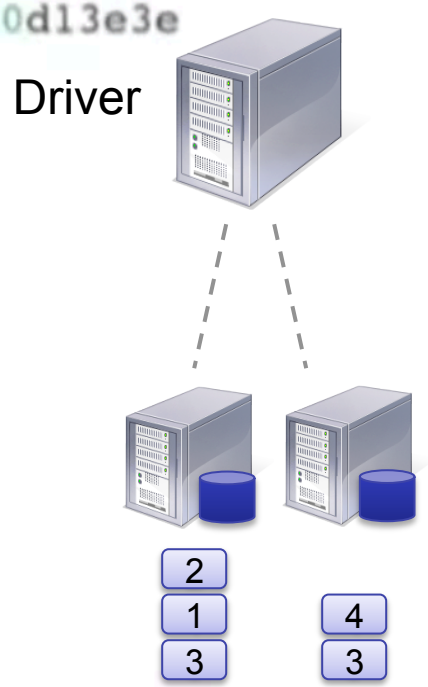
- **An RDD can be persisted into memory or disk**

Example RDD creation from an existing collection in your program

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
```

Example RDD creation from external data

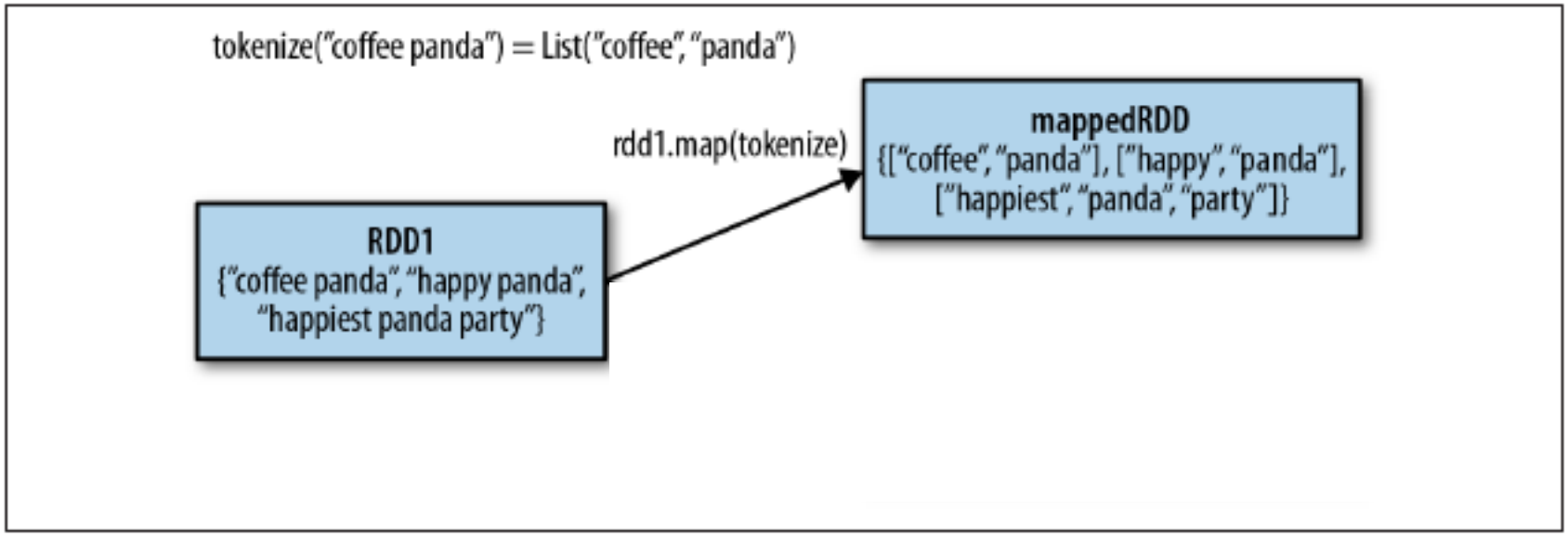
Scala:

```
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
```

Mapper: Apply tokenize to each input record

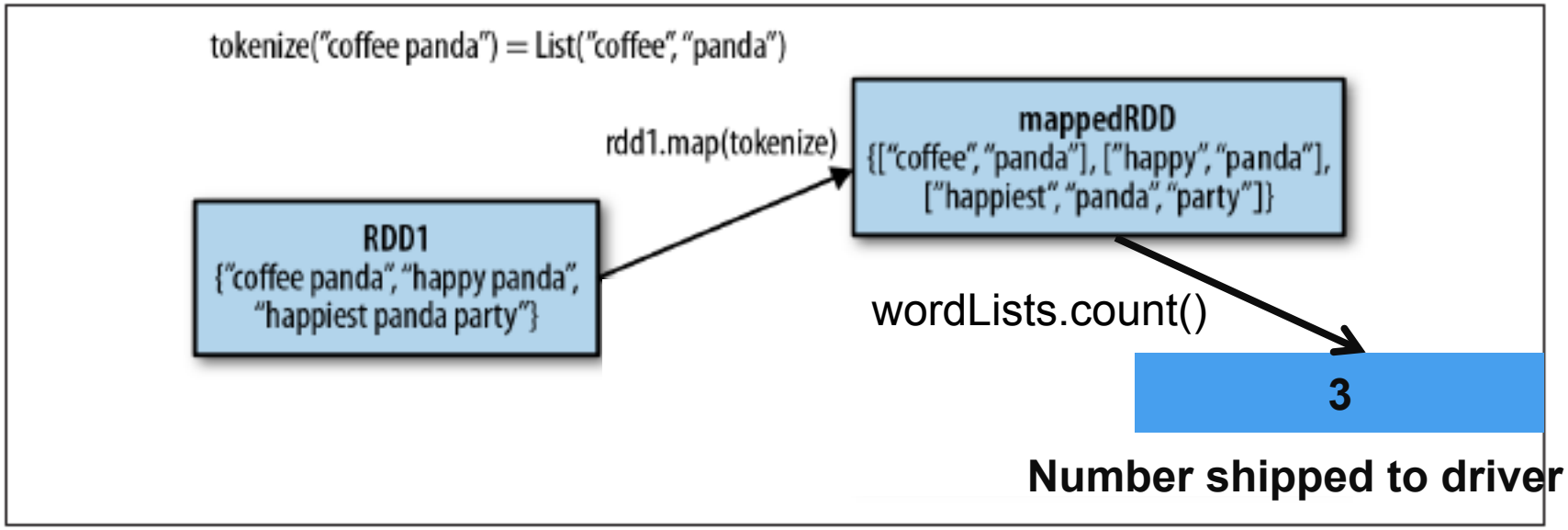



```
def tokenize(line):
    return(line.split(" ")))

rdd1= sc.parallelize(["coffee panda", "happy panda", "happiest panda party"])
words = rdd.map(tokenize)
```

Framework takes care of passing the mapper's function to the executors (task nodes)

Built in reducer for counting (action)




```
def tokenize(line):
    return(line.split(" ")))
```

```
rdd1= sc.parallelize([”coffee panda", ”happy panda”, “happiest panda party”])
wordLists = rdd1.map(tokenize)
wordLists.count()
```

Element-wise transformations

Chain mappers (transformations)

The two most common transformations you will likely be using are `map()` and `filter()` (see Figure 3-2). The `map()` transformation takes in a function and applies it to each element in the RDD with the result of the function being the new value of each element in the resulting RDD. The `filter()` transformation takes in a function and returns an RDD that only has elements that pass the `filter()` function.

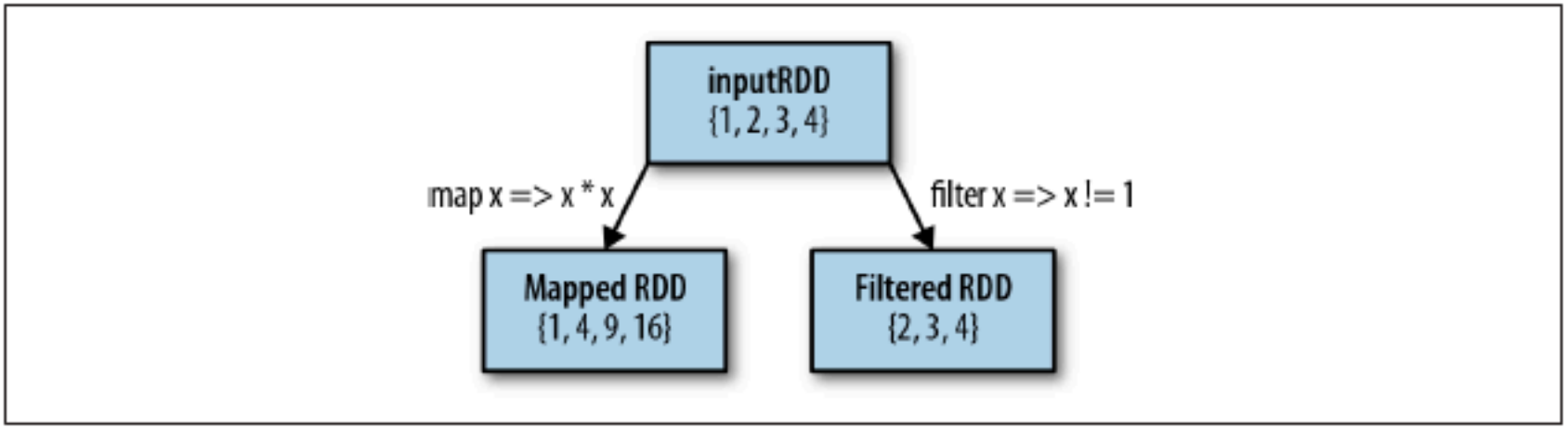


Figure 3-2. Mapped and filtered RDD from an input RDD

We can use `map()` to do any number of things, from fetching the website associated with each URL in our collection to just squaring the numbers. It is useful to note that `map()`'s return type does not have to be the same as its input type, so if we had an RDD `String` and our `map()` function were to parse the strings and return a `Double`, our input RDD type would be `RDD[String]` and the resulting RDD type would be `RDD[Double]`.

Lazy Evaluation

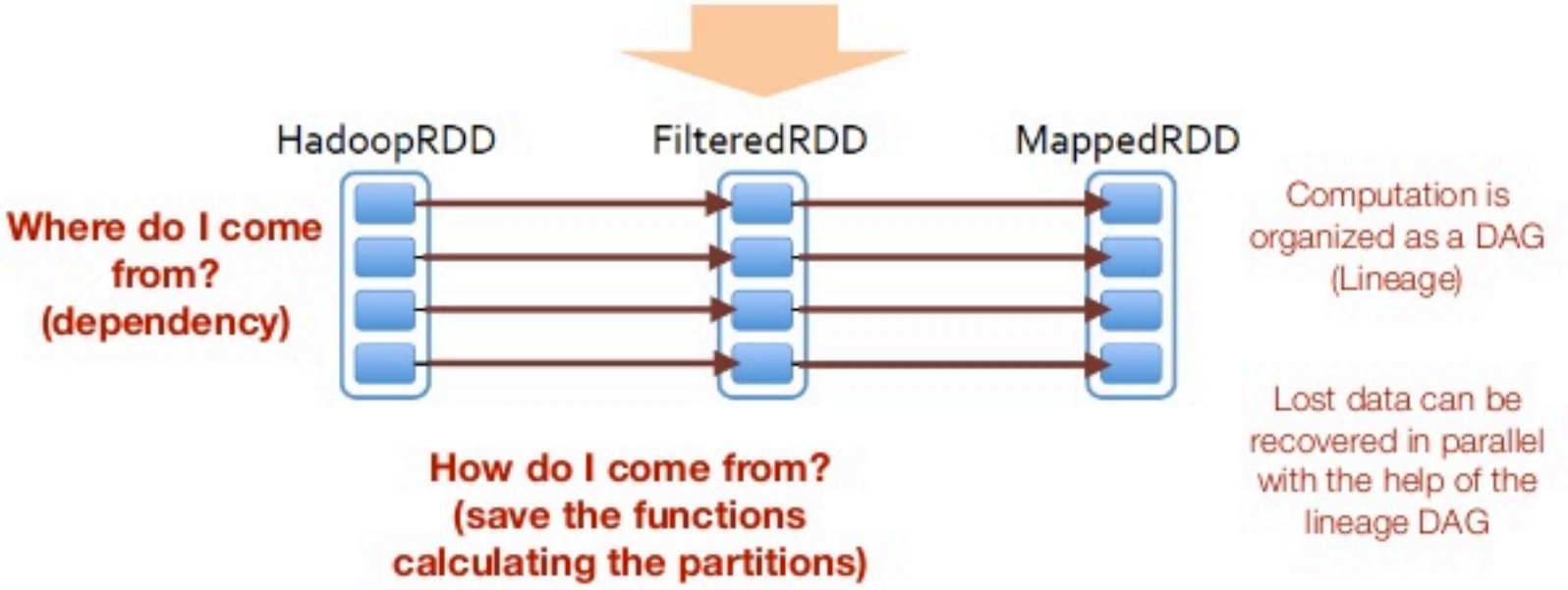
- **Transformations and actions are different because of the way Spark computes RDDs.**
- **Although you can define new RDDs any time, Spark computes them only in a lazy fashion—that is, the first time they are used in an action.**
  - lines = sc.textFile(“exampleFile”) #if not lazy would read whole file in
- **Versus**
  - lines = sc.textFile(“exampleFile”).first()

Computation is organized as a DAG (Lineage/Recipe): resiliency

From data to computation

- Lineage

```
errorMessages= sc.parallelize.filter(lambda x: 'ERROR' in x) \
                .map(lambda line: line.split(" ")[1]) #second word
```



RDD: Lineage Graph

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

`union()` is a bit different than `filter()`, in that it operates on two RDDs instead of one. Transformations can actually operate on any number of input RDDs.

A better way to accomplish the same result as in Example 3-14 would be to simply filter the `inputRDD` once, looking for either *error* or *warning*.

RDD: Spark keeps track of the set of dependancises between different RDDs using Lineage Graph

Finally, as you derive new RDDs from each other using transformations, Spark keeps track of the set of dependencies between different RDDs, called the *lineage graph*. It uses this information to compute each RDD on demand and to recover lost data if part of a persistent RDD is lost. Figure 3-1 shows a lineage graph for Example 3-14.

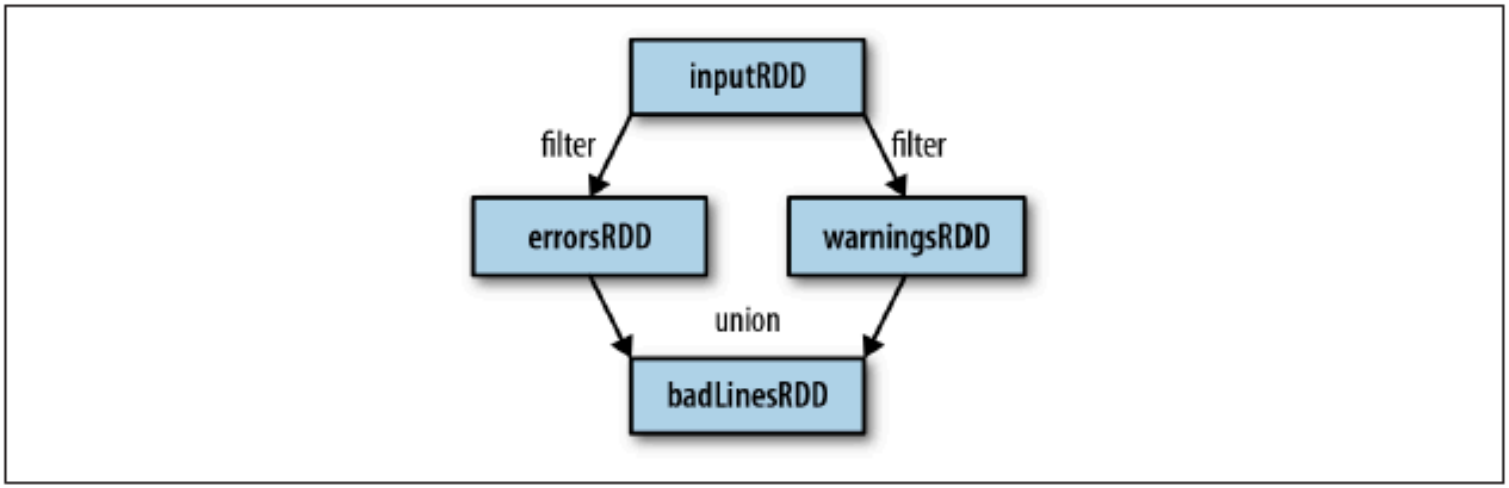


Figure 3-1. RDD lineage graph created during log analysis

counts.toDebugString() #lineage

Slide 1/2

```
In [5]: lines = sc.parallelize(["Data line 1", "Mining line 2", "data line 3", "Data line 4", "Data Mining line 5"])
        counts = lines.flatMap(lambda line: line.split(" ")) \
                      .map(lambda word: (word, 1)) \
                      .reduceByKey(lambda a, b: a + b)

        counts.collect()

Out[5]: [('1', 1),
         ('line', 5),
         ('Mining', 2),
         ('3', 1),
         ('2', 1),
         ('data', 1),
         ('5', 1),
         ('Data', 3),
         ('4', 1)]

In [ ]: counts.to
        counts.toDF
        counts.toDebugString
        counts.toLocalIterator
        counts.top

In [2]: and with NO stochasticity!)
        n
           +   1/m Σi(1   −   yi(w'xi − b))+
        4 # gradient
```

counts.toDebugString() #lineage/recipe

```
In [5]: lines = sc.parallelize(["Data line 1", "Mining line 2", "data line 3", "Data line 4", "Data Mining line 5"])
        counts = lines.flatMap(lambda line: line.split(" ")) \
                      .map(lambda word: (word, 1)) \
                      .reduceByKey(lambda a, b: a + b)

        counts.collect()
```

Slide 2/2

```
Out[5]: [('1', 1),
         ('line', 5),
         ('Mining', 2),
         ('3', 1),
         ('2', 1),
         ('data', 1),
         ('5', 1),
         ('Data', 3),
         ('4', 1)]
```

```
In [7]: counts.toDebugString()
```

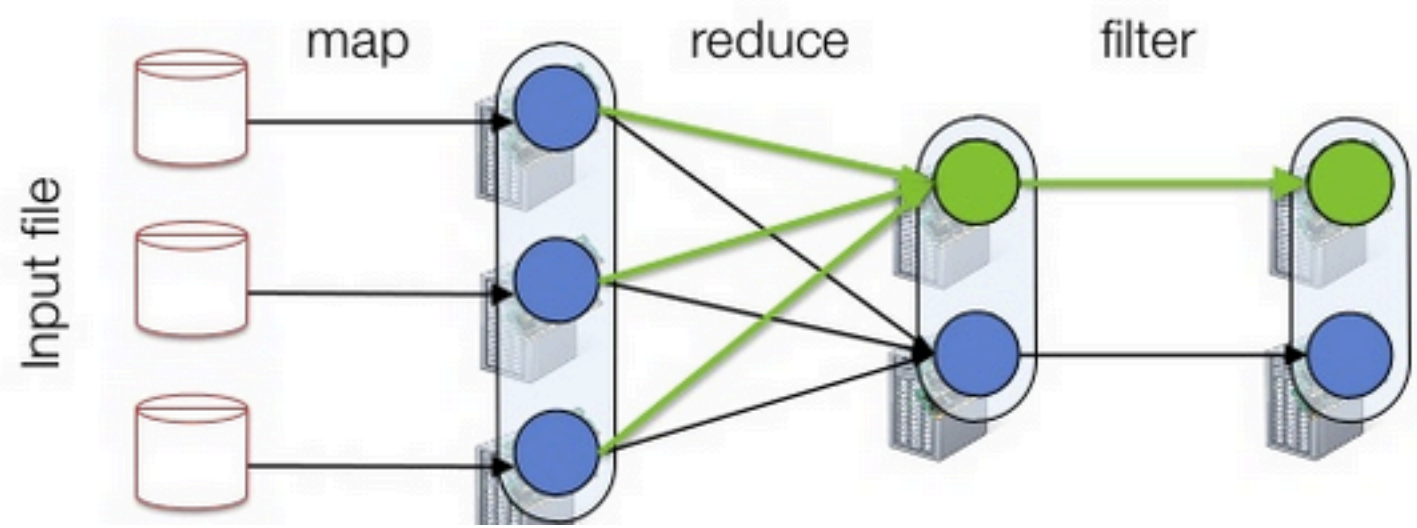
```
Out[7]: '(8) PythonRDD[7] at collect at <ipython-input-5-708c6b1f0496>:4 []\n |  MapPartitionsRDD[6] at mapPartitions at Pytho
        nRDD.scala:346 []\n |  ShuffledRDD[5] at partitionBy at NativeMethodAccessorImpl.java:-2 []\n +-(8) PairwiseRDD[4] at
        reduceByKey at <ipython-input-5-708c6b1f0496>:2 []\n    |  PythonRDD[3] at reduceByKey at <ipython-input-5-708c6b1f049
        6>:2 []\n    |  ParallelCollectionRDD[2] at parallelize at PythonRDD.scala:396 []'
```

PythonRDD[7] at collect at <ipython-input-5-708c6b1f0496>:4 []\n
| MapPartitionsRDD[6] at mapPartitions at PythonRDD.scala:346 []\n
| ShuffledRDD[5] at partitionBy at NativeMethodAccessorImpl.java:-2 []\n
+-(8) PairwiseRDD[4] at reduceByKey at <ipython-input-5-708c6b1f0496>:2 []\n
| PythonRDD[3] at reduceByKey at <ipython-input-5-708c6b1f0496>:2 []\n
| ParallelCollectionRDD[2] at parallelize at PythonRDD.scala:396 []'

Fault Tolerance

RDDs track *lineage* info to rebuild lost data

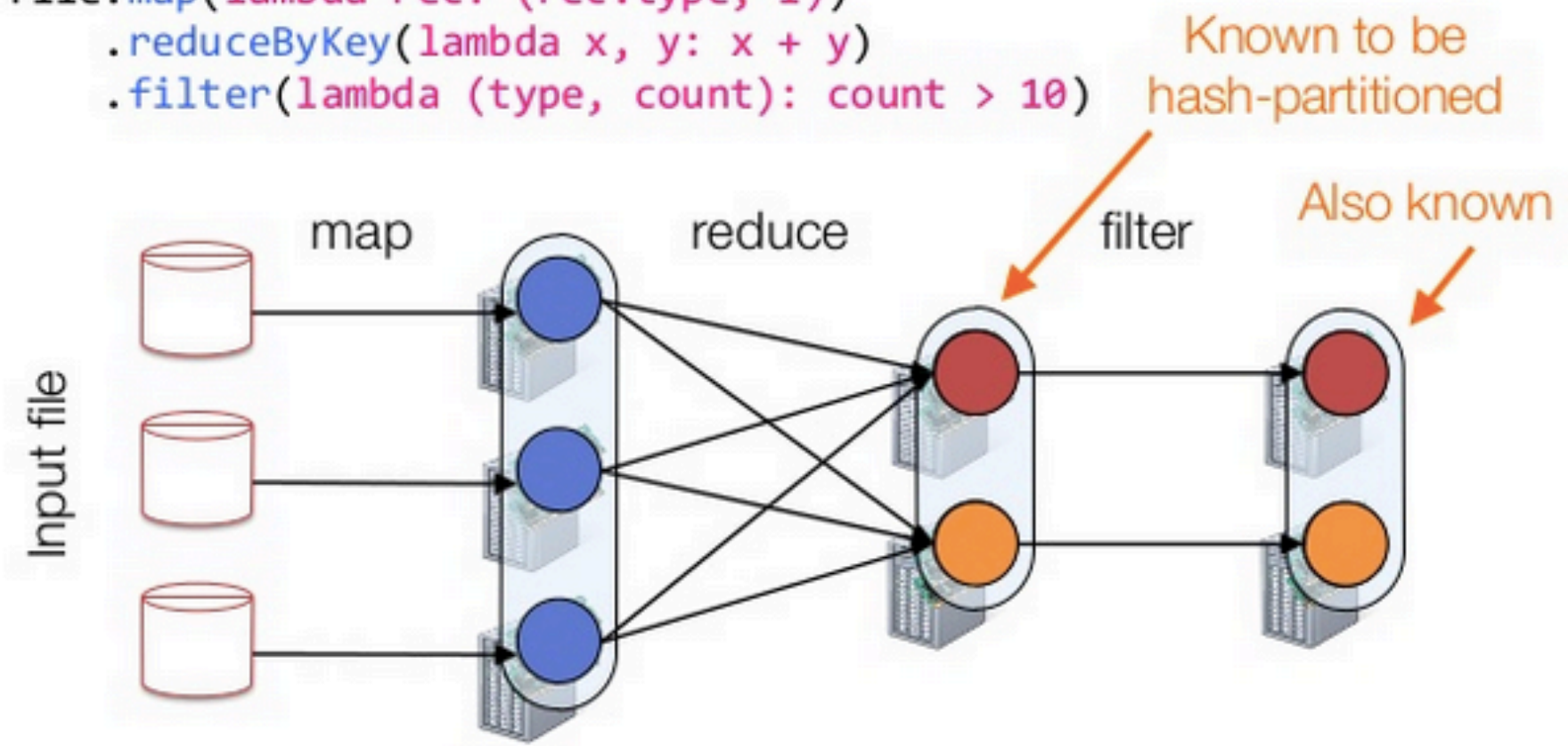
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



RDDs know their partioning function

- ..

```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



In summary

- That was a look at our first Spark program
- Just used very basic map and reduce (count) operations over the distributed key-value store (RDD)
- We explore more operations next and go a little deeper

Part 2: Spark Intro and Basics

- **Part 2: Spark Intro and basics**
  - Base RDD
  - Fault tolerance (and lineage)
  - Transformations and Actions
  - Persistence
  - Animated Example
  - Pair RDDs
  - Word count example

This section

- **Just used very basic map and reduce (count) operations over the distributed key-value store (RDD)**
- **We explore more operations next and go a little deeper:**
  - Work on base RDDs and how the Spark framework works
  - Write some code

Spark Runtime

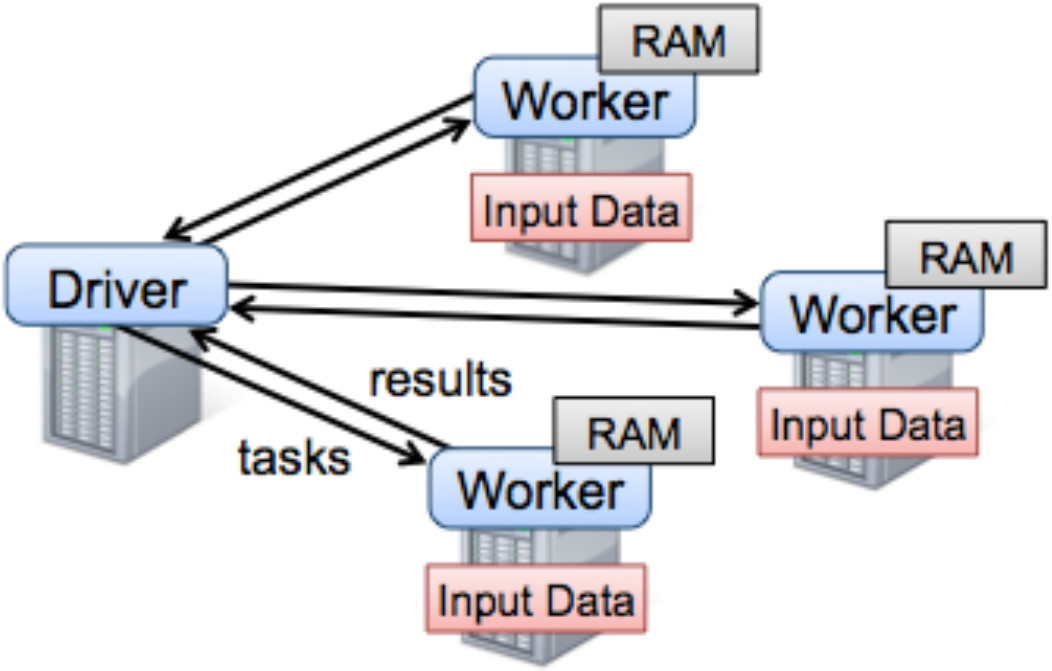


Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

Divide and conquer with Closures

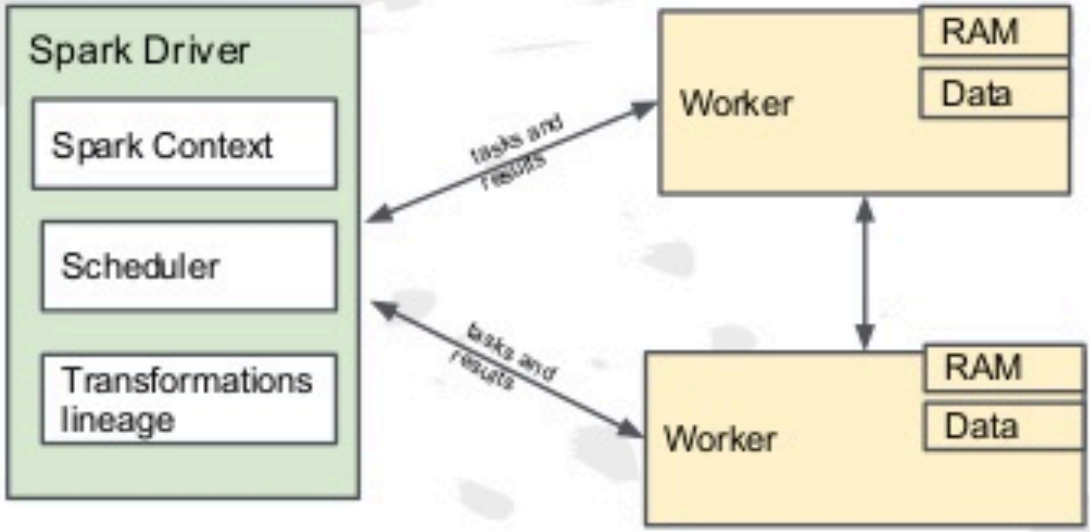
- **Decompose problems in non-ovrelapping subproblems**
- **Spark's API relies heavily on passing functions in the driver program to run on the cluster. There are three recommended ways to do this:**
  - Lambda expressions, for simple functions that can be written as an expression. (Lambdas do not support multi-statement functions or statements that do not return a value.)
  - Local defs inside the function calling into Spark, for longer code.
  - Top-level functions in a module.
- **Lazy evaluation! Optimize execution graphs**

Driver vs Workers

At the core of Spark...

Spark Driver
- Define & launch operations on RDDs
- Keeps track of RDD lineage allowing to recover lost partitions
- Use the RDD DAG in the Scheduler to define the stages and tasks to be executed

Workers
- Read/Transform/Write RDD partitions
- Need to communicate with other workers to share their cache and shuffle data



Closure (computer programming)

Closure

From Wikipedia, the free encyclopedia

For other uses of this term, including in mathematics and computer science, see Closure.
Not to be confused with Clojure.

In programming languages, **closures** (also **lexical closures** or **function closures**) are a technique for implementing lexically scoped name binding in languages with first-class functions. Operationally, a closure is a data structure storing a function[a] together with an environment:[1] a mapping associ… with the value or storage location … the function to access those *captu…* scope.

A closure is a data structure storing a function[a] together with an environment:

- **a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or storage location the name was bound to at the time the closure was created.**

Example The following program f… `incrementBy`. The nested func… though `x` is not local to `increm…` the `y` value to the `x` value, and… find it once invoked:

```
function startAt(x)
   function incrementBy
       return x + y
   return incrementBy

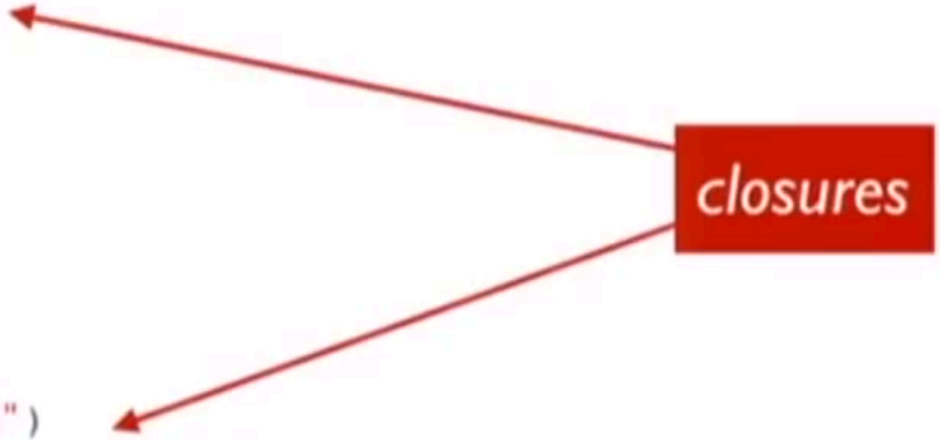
variable closure1 = startAt(1)
variable closure2 = startAt(5)
```

Note that, as `startAt` returns a function, the variables $closure_1$ and $closure_2$ are of function type. Invoking $closure_1(3)$ will return `4`, while invoking $closure_2(3)$ will return `8`. While $closure_1$ and $closure_2$ have the same function `incrementBy`, the associated environments differ, and invoking the closures will bind the name `x` to two distinct variables in the two invocations, with different values, thus evaluating the function to different results.

Spark Essentials: *Transformations*

Scala:

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



Python:

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

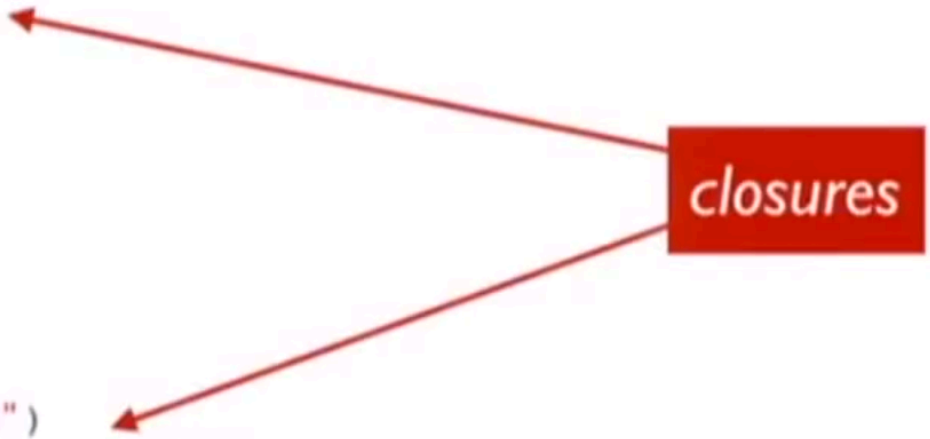
Spark Essentials: *Transformations*

Scala:

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

Python:

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



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

A Hello World Example: Summary stats of a large random dataset

- Example from scratch
- Generate a random array of numbers and put them into an RDD
- Transform them by doubling each one
- Filter all numbers > 1
- RDD distributes the data (but not immediately, it waits, it is lazy!!)
- Cache the RDD in memory [still lazy]
- Count the number of numbers < 1 (should be 0)
- Count the number of numbers > 1
- Count the number of numbers > 1 (should be the same as the previous result)

SparkContext:

- **A connection to a computing cluster**
- **Through SparkContext, driver program can access Spark.**
- **Automatically created in interactive shell**
- **You can also create your own SparkContext**

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setMaster("local").setAppName("App")
sc = SparkContext(conf = conf)
```

IP[y]: Notebook Chapter-3-RDDS Last Checkpoint: Mar 16 15:24 (unsaved changes)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

Start Spark Cluster locally and attach notebook to cluster

In [1]:
```
import os
import sys

#set up Spark and give us a spark context sc
spark_home = os.environ['SPARK_HOME']='/Users/jshanahan/Software/spark-1.2.1-bin-hadoop2.4/'  #desktop

print "["+ spark_home+"]"
if not spark_home:
    raise ValueError('SPARK_HOME environment variable is not set')

sys.path.insert(0, os.path.join(spark_home, 'python'))
sys.path.insert(0, os.path.join(spark_home, 'python/lib/py4j-0.8.2.1-src.zip'))
execfile(os.path.join(spark_home, 'python/pyspark/shell.py'))
```

```
[/Users/jshanahan/Software/spark-1.2.1-bin-hadoop2.4/]
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.2.1
      /_/

Using Python version 2.7.8 (default, Aug 21 2014 15:21:46)
SparkContext available as sc.
```

See next slide for detailed breakdown

```
1 import numpy as np
2 import pylab
3
4 dataRDD = sc.parallelize(np.random.random_sample(1000))
5 data2X= dataRDD.map(lambda x: x*2)
6 dataGreaterThan1 = data2X.filter(lambda x: x > 1.0)
7 cachedRDD = dataGreaterThan1.cache()
```

In [29]:
```
1 cachedRDD.filter(lambda x: x<1).count()
```
Out[29]: 0

In [31]:
```
1 cachedRDD.filter(lambda x: x>1).count()
```
Out[31]: 514

In [32]:
```
1 cachedRDD.filter(lambda x: x>1).count()
```
Out[32]: 514

```
In [28]:
1 import numpy as np
2 import pylab
3
4 dataRDD = sc.parallelize(np.random.random_sample(1000))
5 data2X= dataRDD.map(lambda x: x*2)
6 dataGreaterThan1 = data2X.filter(lambda x: x > 1.0)
7 cachedRDD = dataGreaterThan1.cache()

In [29]:
1 cachedRDD.filter(lambda x: x<1).count()

Out[29]: 0

In [31]:
1 cachedRDD.filter(lambda x: x>1).count()

Out[31]: 514

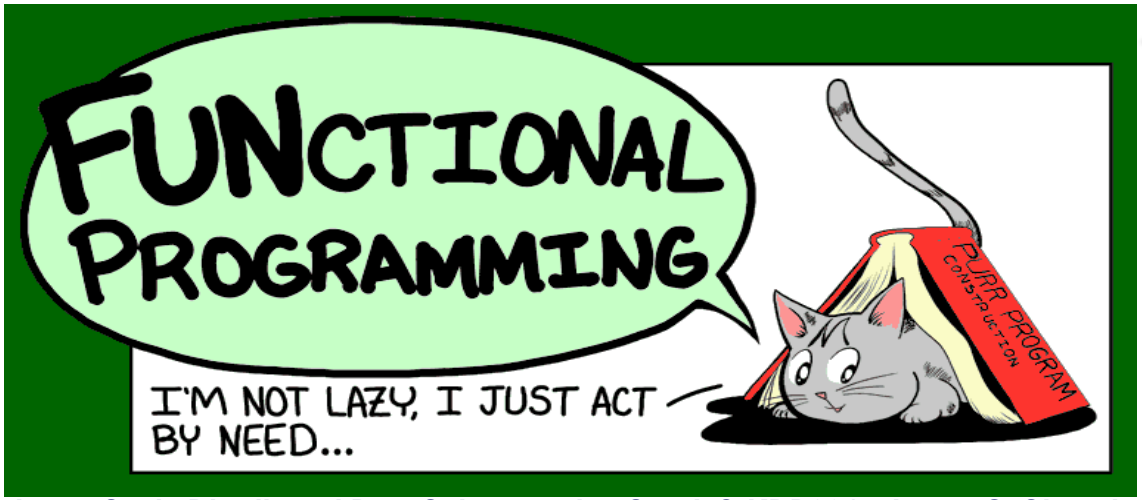
In [32]:
1 cachedRDD.filter(lambda x: x>1).count()

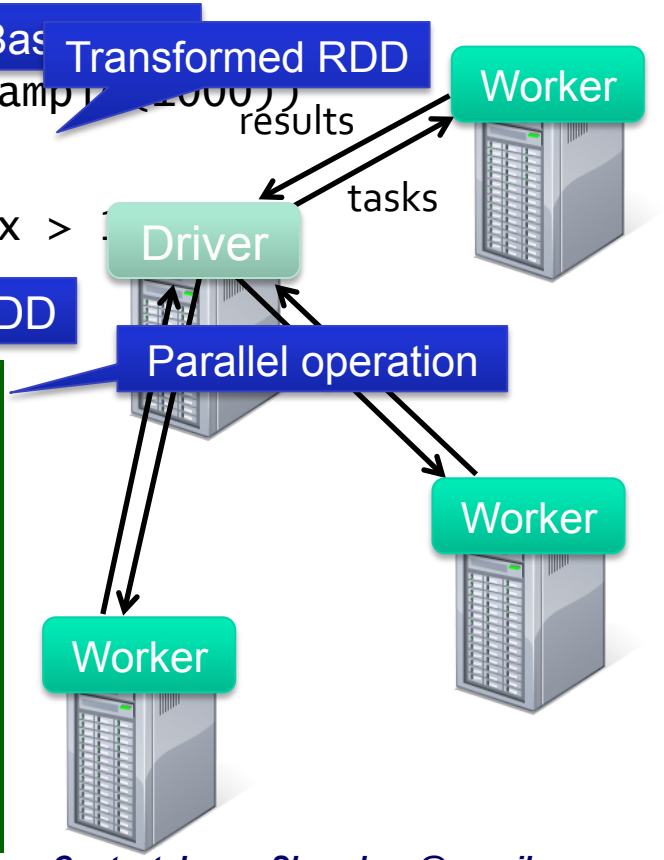
Out[32]: 514
```

Example: Random Numbers

- SC: Spark Context (connection to cluster driver)
- Load a bunch random numbers into an RDD
- Spark is Lazy…

```
dataRDD = sc.parallelize(np.random.random_samp
data2X= dataRDD.map(lambda x: x*2)
dataGreaterThan1 = data2X.filter(lambda x: x >
cachedRDD = dataGreaterThan1.cach
```





Two types RDD Operations

Transformations (define a new RDD)	Parallel operations (Actions) (return a result to driver)
map	reduce
filter	collect
sample	count
union	save
groupByKey	lookupKey
reduceByKey	…
join	
cache	
…	

Nothing is materialized

Coffee break

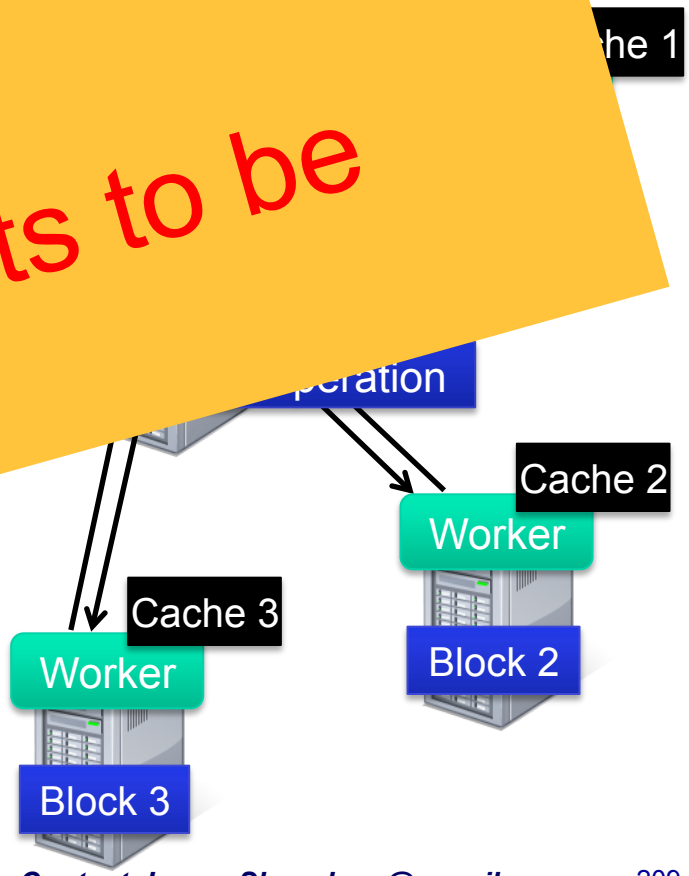
Example: Random Numbers

- Load a bunch random numbers into RDD

Is my animation broken?!
NO! Spark is lazy !
Count triggers results to be materialized

```
dataRDD = sc
da
da
cac

> ca
508
> cac
298
. . .
```



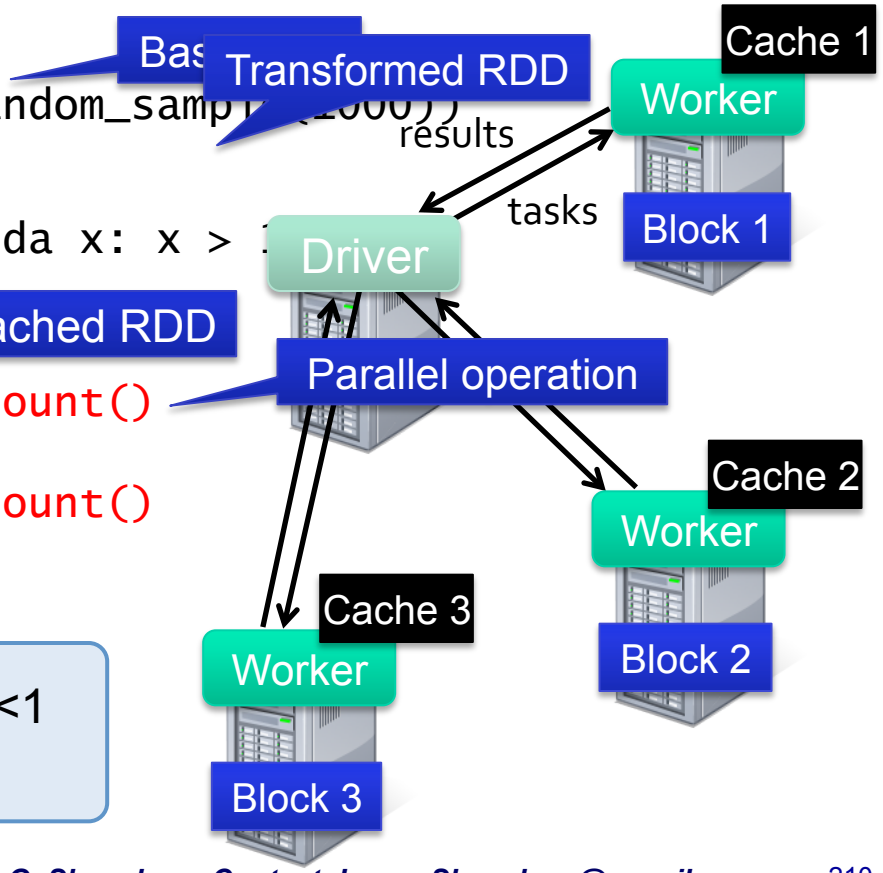
Re full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

Example: Random Numbers

- Load a bunch random numbers into an RDD
- Transform, filter, and the count (action)

```
dataRDD = sc.parallelize(np.random.random_samp
data2X= dataRDD.map(lambda x: x*2)
dataGreaterThan1 = data2X.filter(lambda x: x >
cachedRDD = dataGreaterThan1.cach

> cachedRDD.filter(lambda x: x>0.5).count()
508
> cachedRDD.filter(lambda x: x>0.7).count()
298
. . .
```



Result: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

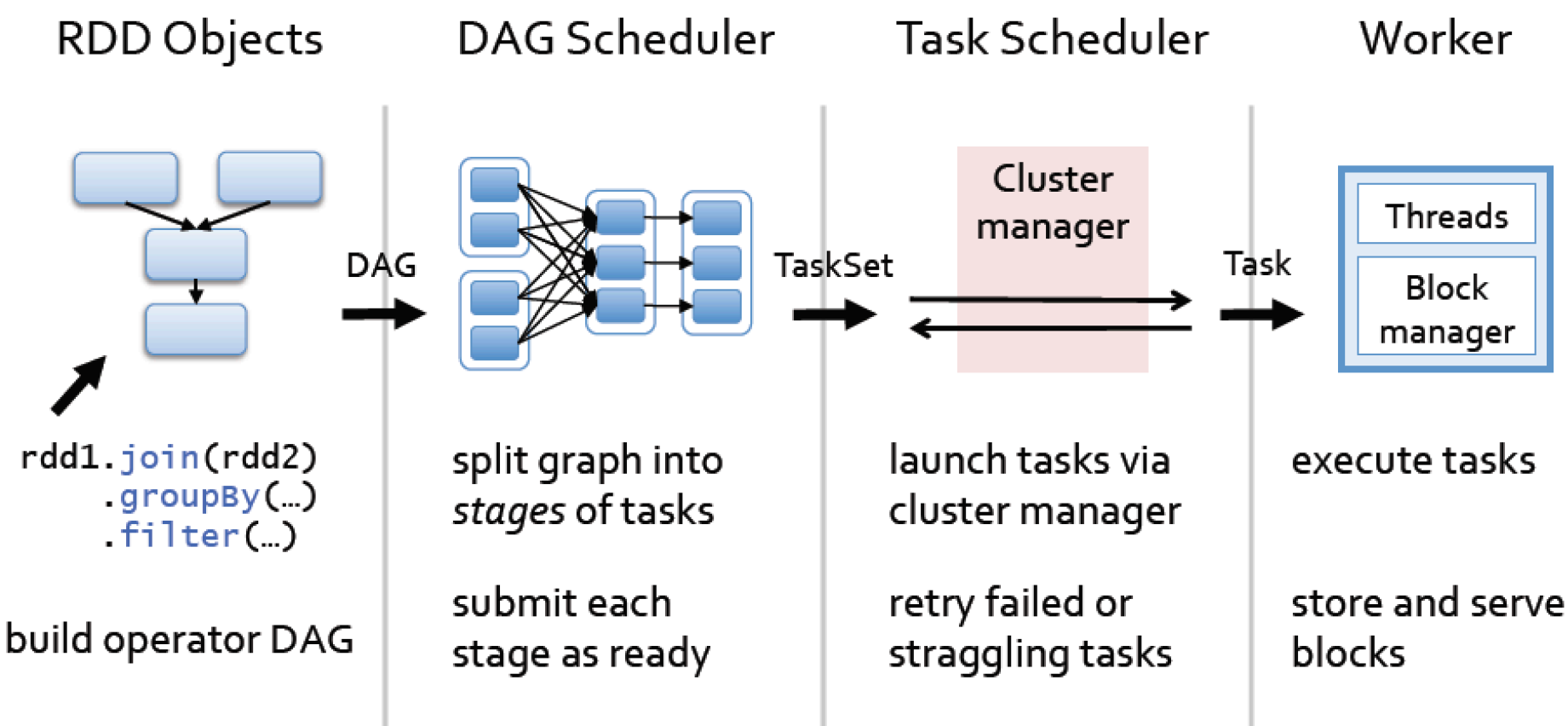
Recompute vs Cache

- **Spark's RDDs are by default recomputed each time you run an action on them.**
- **If you would like to reuse an RDD in multiple actions, you can ask Spark to persist it using RDD.persist().**

Shipping code to the cluster

Optimized steps into stages

RDD → Stages → Tasks



Resilient Distributed Datasets, or RDDs

The `SparkContext` has a long list of methods, but the ones that we're going to use most often allow us to create *Resilient Distributed Datasets*, or *RDDs*. An RDD is Spark's fundamental abstraction for representing a collection of objects that can be distributed across multiple machines in a cluster. There are two ways to create an RDD in Spark:

- Using the `SparkContext` to create an RDD from an external data source, like a file in HDFS, a database table via JDBC, or from a local collection of objects that we create in the Spark shell.
- Performing a transformation on one or more existing RDDs, like filtering records, aggregating records by a common key, or joining multiple RDDs together.

RDDs are a convenient way to describe the computations that we want to perform on our data as a sequence of small, independent steps.

RDD: parallelize, textFile

Resilient Distributed Datasets

An RDD is laid out across the cluster of machines as a collection of *partitions*, each including a subset of the data. Partitions define the unit of parallelism in Spark. The framework processes the objects within a partition in sequence, and processes multiple partitions in parallel. One of the simplest ways to create an RDD is to use the `parallelize` method on `SparkContext` with a local collection of objects:

```
val rdd = sc.parallelize(Array(1, 2, 2, 4), 4)
...
rdd: org.apache.spark.rdd.RDD[Int] = ...
```

Number of partitions

The first argument is the collection of objects to parallelize. The second is the number of partitions. When the time comes to compute the objects within a partition, Spark fetches a subset of the collection from the driver process.

To create an RDD from a text file or directory of text files residing in a distributed file system like HDFS, we can pass the name of the file or directory to the `textFile` method:

```
val rdd2 = sc.textFile("hdfs:///some/path.txt")
...
rdd2: org.apache.spark.rdd.RDD[String] = ...
```

Create and RDD from a file or from a directory

- **Transformations**

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)

Transformations vs Actions

Transformation: RDD → RDD

Once created, RDDs offer two types of operations: *transformations* and *actions*. *Transformations* construct a new RDD from a previous one. For example, one common transformation is filtering data that matches a predicate. In our text file example, we can use this to create a new RDD holding just the strings that contain the word *Python*, as shown in Example 3-2.

Example 3-2. Calling the filter() transformation

```
>>> pythonLines = lines.filter(lambda line: "Python" in line)
```

Actions: RDD → result is given the driver or saved to external storage

Actions, on the other hand, compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g., HDFS). One example of an action we called earlier is `first()`, which returns the first element in an RDD and is demonstrated in Example 3-3.

Example 3-3. Calling the first() action

```
>>> pythonLines.first()
u'## Interactive Python Shell'
```

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

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()
```

Persistance

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.

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.

RDD Operations (over 80 ops)

Transformations (define a new RDD)	**Parallel operations (Actions)** (return a result to driver)
map filter sample union *PAIR RDDs (below)* *groupByKey* *reduceByKey* *join* *cache* …	reduce collect count save *lookupKey* …

Nothing is materialized

Other RDD Operations

Transformations (define a new RDD)	map filter sample *groupByKey* *reduceByKey* cogroup	flatMap union join cross mapValues ...
Actions (output a result)	collect reduce take fold	count saveAsTextFile saveAsHadoopFile ...

Map versus flatmap

Example 3-26. Python squaring the values in an RDD

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
for num in squared:
    print "%i " % (num)
```

Sometimes we want to produce multiple output elements for each input element. The operation to do this is called `flatMap()`. As with `map()`, the function we provide to `flatMap()` is called individually for each element in our input RDD. Instead of returning a single element, we return an iterator with our return values. Rather than producing an RDD of iterators, we get back an RDD that consists of the elements from all of the iterators. A simple usage of `flatMap()` is splitting up an input string into words, as shown in Examples 3-29 through 3-31.

Input produces multiple elements; want to get all into the stream

Example 3-29. flatMap() in Python, splitting lines into words

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.first()  # returns "hello"
```

flatMap() versus map()

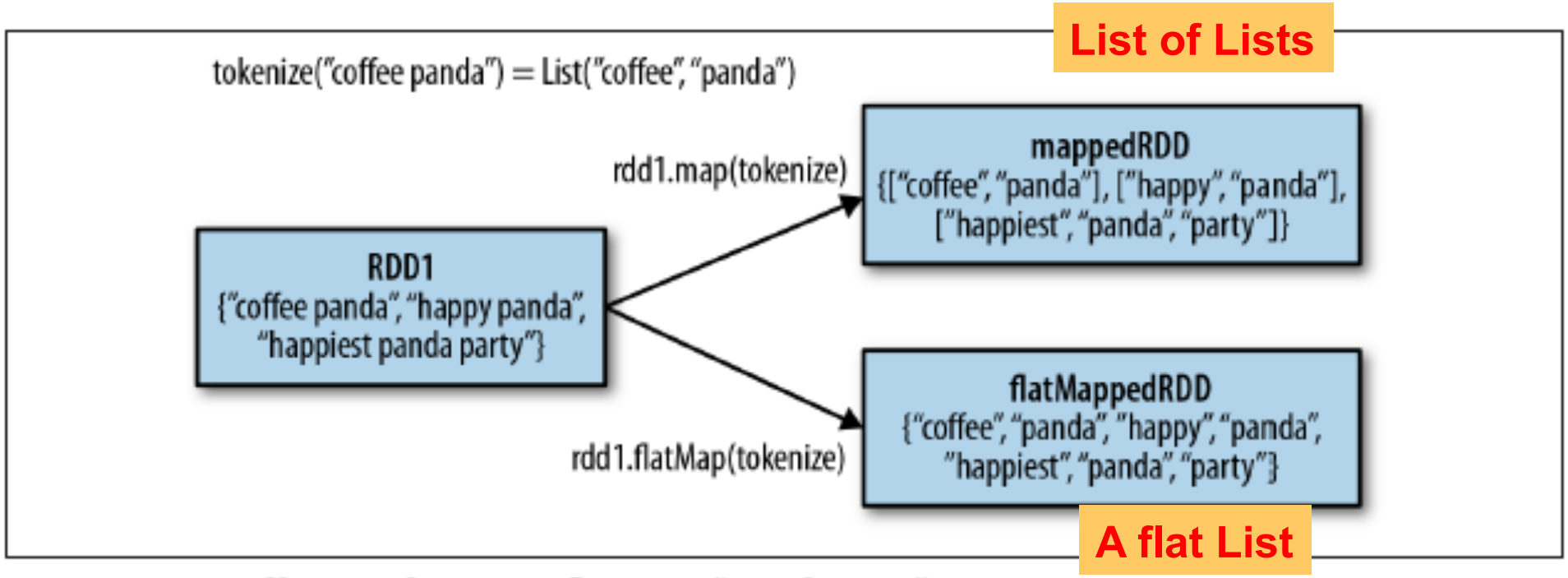
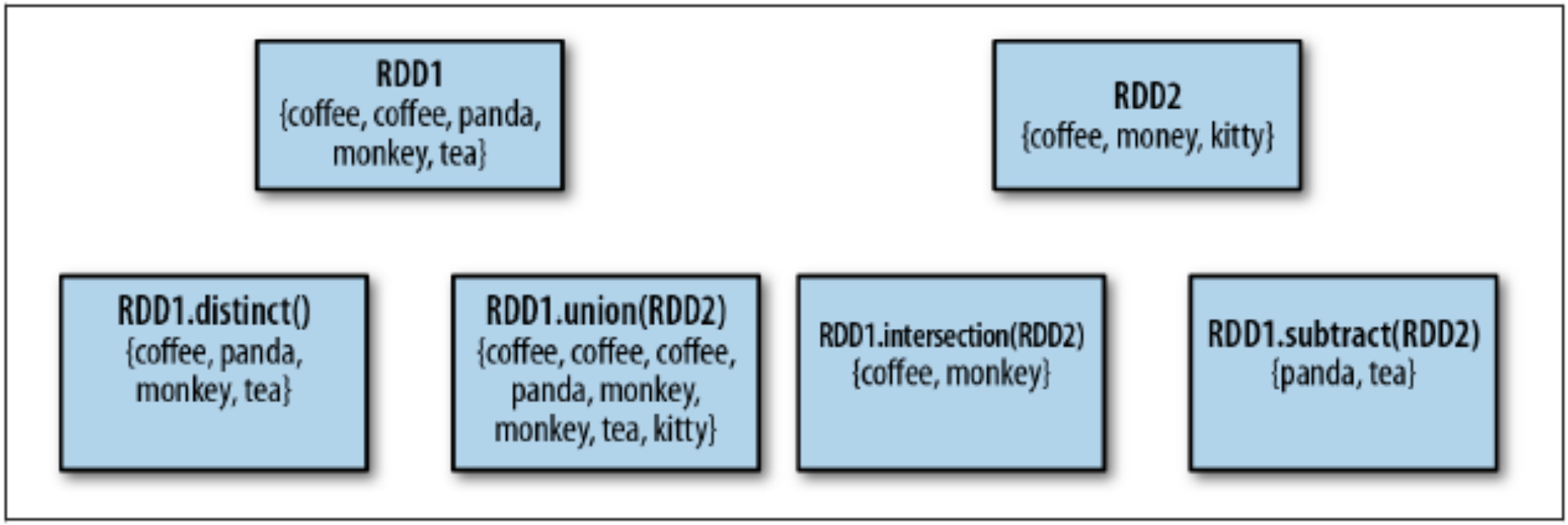
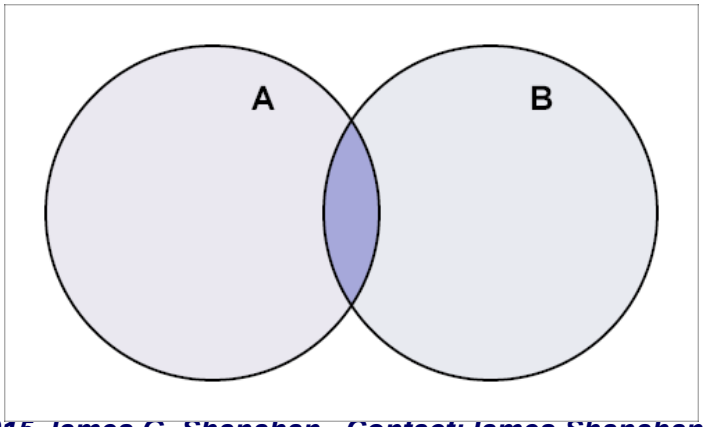


Figure 3-3. Difference between flatMap() and map() on an RDD

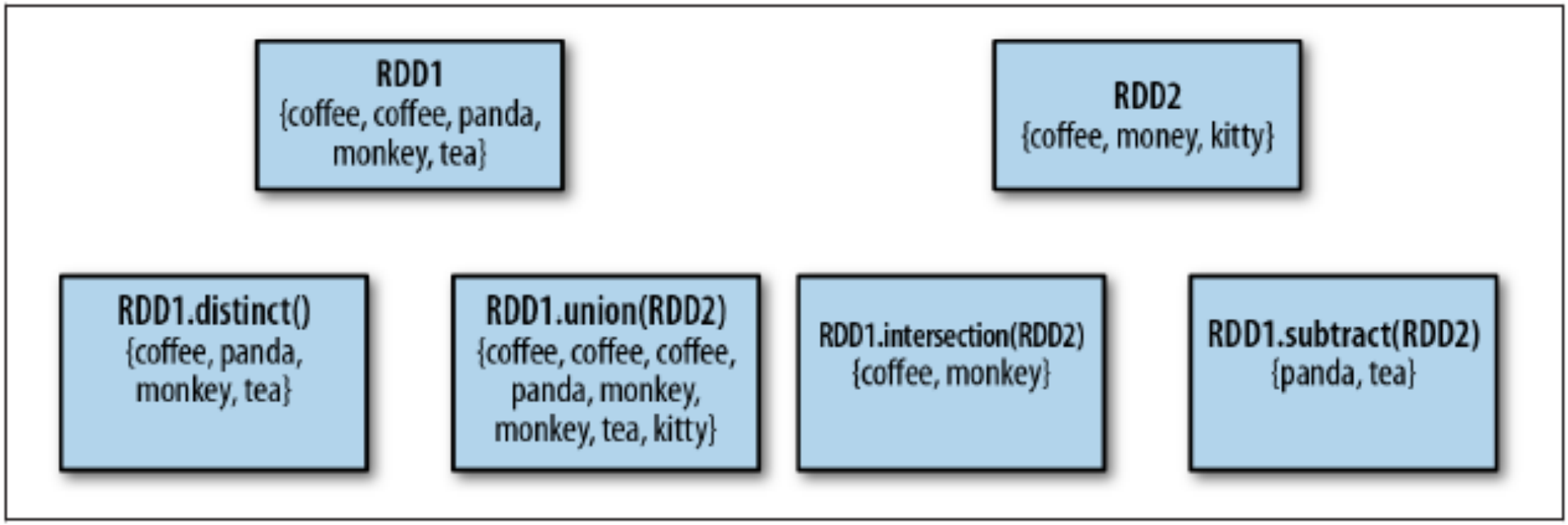
Set operations



- **Distinct**
- **Union**
- **Intersection**
- **Subtract**

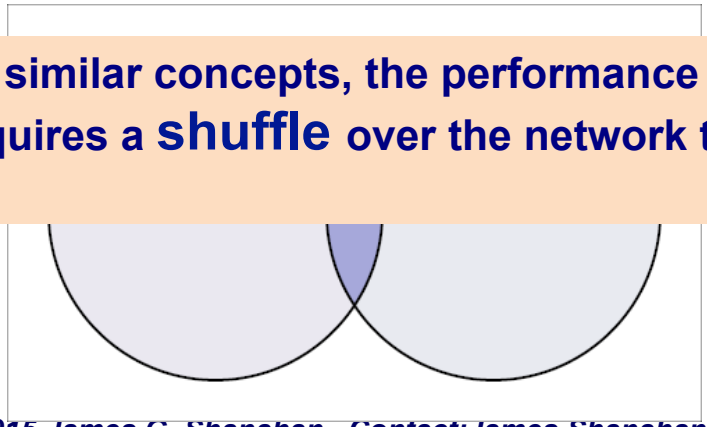


Set operations



While intersection() and union() are two similar concepts, the performance of intersection() is much worse since it requires a shuffle over the network to identify common elements.

- **Intersection**
- **Subtract**



Cartesian Product, intersection, etc.

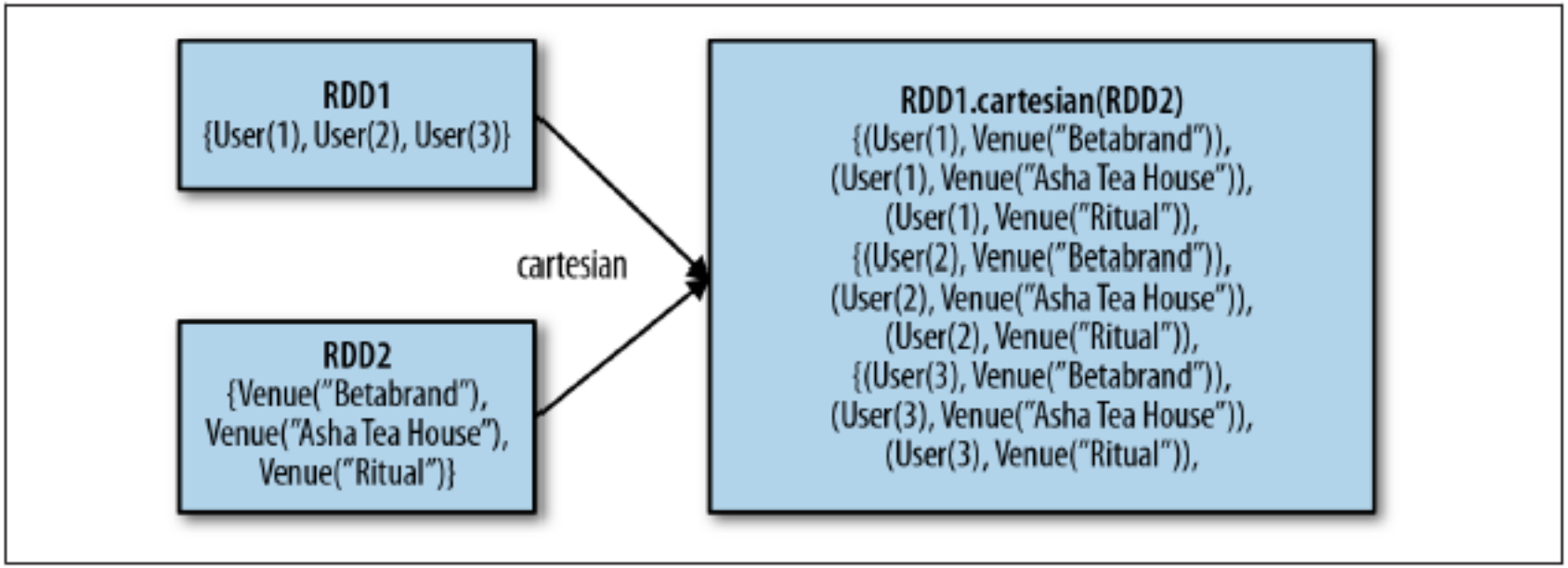


Figure 3-5. Cartesian product between two RDDs

Be warned, however, that the Cartesian product is very expensive for large RDDs.

Table 3-2. Basic RDD transformations on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
Distinct	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement **Sample**	`rdd.sample(false, 0.5)`	Nondeterministic

Table 3-3. Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

Function name	Purpose	Example	Result
`union()`	Produce an RDD containing elements from both RDDs.	`rdd.union(other)`	`{1, 2, 3, 3, 4, 5}`
`intersection()`	RDD containing only elements found in both RDDs.	`rdd.intersection(other)`	`{3}`
`subtract()`	Remove the contents of one RDD (e.g., remove training data).	`rdd.subtract(other)`	`{1, 2}`
`cartesian()`	Cartesian product with the other RDD.	`rdd.cartesian(other)`	`{(1, 3), (1, 4), … (3,5)}`

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

union()

`union()` is a bit different than `filter()`, in that it operates on two RDDs instead of one. Transformations can actually operate on any number of input RDDs.

A better way to accomplish the same result as in Example 3-14 would be to simply filter the `inputRDD` once, looking for either *error* or *warning*.

Finally, as you derive new RDDs from each other using transformations, Spark keeps track of the set of dependencies between different RDDs, called the *lineage graph*. It uses this information to compute each RDD on demand and to recover lost data if part of a persistent RDD is lost. Figure 3-1 shows a lineage graph for Example 3-14.

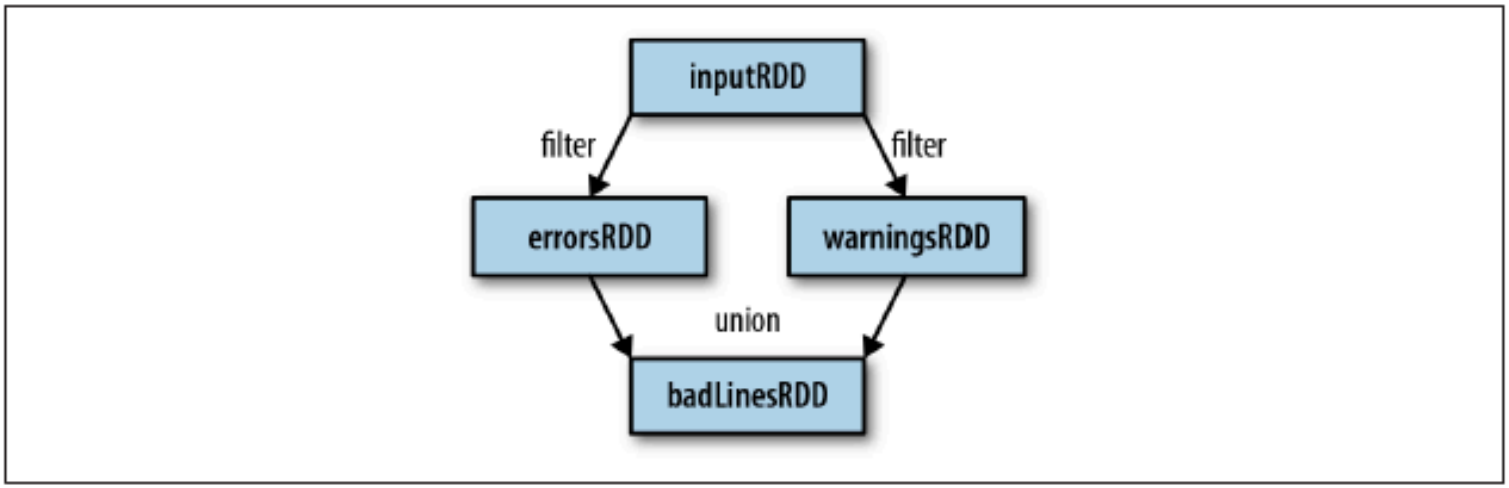


Figure 3-1. RDD lineage graph created during log analysis

Actions: Collect can do a lot of damage

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

Example 3-15. Python error count using actions

```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
    print line
```

Example 3-16. Scala error count using actions

```
println("Input had " + badLinesRDD.count() + " concerning lines")
println("Here are 10 examples:")
badLinesRDD.take(10).foreach(println)
```

Example 3-17. Java error count using actions

```
System.out.println("Input had " + badLinesRDD.cou
System.out.println("Here are 10 examples:")
for (String line: badLinesRDD.take(10)) {
  System.out.println(line);
}
```

In most cases RDDs can't just be collect()ed to the driver because they are too large.

Persisting results on

```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
    print line
```

In most cases RDDs can **NOT** just be collect()ed to the driver because they are too large. In these cases, it's common to write data out to a distributed storage system such as HDFS or Amazon S3.

- You can save the contents of an RDD using the saveAsTextFile() action, saveAsSequenceFile(), or any of a number of actions for various built-in formats. We will cover the different options for exporting data in Chapter 5.

cache()
It is important to note that each time we call a new action, the entire RDD must be computed "from scratch." To avoid this inefficiency, users can persist intermediate results, as we will cover in "Persistence (Caching)" on page 44.

count() action

- The act of creating a RDD does not cause any distributed computation to take place on the cluster.
- Rather, RDDs define logical datasets that are intermediate steps in a computation.
- Distributed computation occurs upon invoking an action on an RDD. For example, the count action returns the number of objects in an RDD.

```
rdd.count()
14/09/10 17:36:09 INFO SparkContext: Starting job: count at <console>:15
...
14/09/10 17:36:09 INFO SparkContext: Job finished: count at <console>:15,
  took 0.18273803 s
res0: Long = 4
```

Get data from Cluster to client (first, collect, take)

- RDDs have a number of methods that allow us to read data from the cluster into the Scala REPL on our client machine. Perhaps the simplest of these is first, which returns the first element of the RDD into the client:

```
rawblocks.first
...
res: String = "id_1","id_2","cmp_fname_c1","cmp_fname_c2",...

val head = rawblocks.take(10)
...
head: Array[String] = Array("id_1","id_2","cmp_fname_c1",...

head.length
...
res: Int = 10
```

collect() → array in local memory

- The collect action returns an Array with all the objects from the RDD.
- This Array resides in local memory, not on the cluster.

- Be careful: don't collect too much data; this will cause your driver to crash

```
rdd.collect()
14/09/29 00:58:09 INFO SparkContext: Starting job: collect at <console>:17
...
14/09/29 00:58:09 INFO SparkContext: Job finished: collect at <console>:17,
  took 0.531876715 s
res2: Array[(Int, Int)] = Array((4,1), (1,1), (2,2))
```

saveAsTextFile()

- Actions need not only return results to the local process. The saveAsTextFile action saves the contents of an RDD to persistent storage like HDFS.

```
rdd.saveAsTextFile("hdfs:///user/ds/mynumbers")
14/09/29 00:38:47 INFO SparkContext: Starting job: saveAsTextFile at <console>:15
...
14/09/29 00:38:49 INFO SparkContext: Job finished: saveAsTextFile at <console>:15,
  took 1.818305149 s
```

- The action creates a directory and writes out each partition as a file within it. From the command line outside of the Spark shell:

```
hadoop fs -ls /user/ds/mynumbers

-rw-r--r--   3 ds supergroup          0 2014-09-29 00:38 myfile.txt/_SUCCESS
-rw-r--r--   3 ds supergroup          4 2014-09-29 00:38 myfile.txt/part-00000
-rw-r--r--   3 ds supergroup          4 2014-09-29 00:38 myfile.txt/part-00001
```

Foreach(): Print-friendly

- **To make it easier t0 read the contents of an array, we can use the foreach method in conjunction with println (print in Python) to print each value in the array out on its own line:**

```
val head = rawblocks.take(10)
...
head: Array[String] = Array("id_1","id_2","cmp_fname_c1",...

head.length
...
res: Int = 10
head.foreach(println)
...
"id_1","id_2","cmp_fname_c1","cmp_fname_c2","cmp_lname_c1","cmp_lname_c2","cmp_sex","cmp_bd","cmp_
37291,53113,0.833333333333333,?,1,?,1,1,1,1,0,TRUE
39086,47614,1,?,1,?,1,1,1,1,1,TRUE
70031,70237,1,?,1,?,1,1,1,1,1,TRUE
84795,97439,1,?,1,?,1,1,1,1,1,TRUE
36950,42116,1,?,1,1,1,1,1,1,1,TRUE
42413,48491,1,?,1,?,1,1,1,1,1,TRUE
25965,64753,1,?,1,?,1,1,1,1,1,TRUE
49451,90407,1,?,1,?,1,1,1,1,0,TRUE
```

functional programming pattern

- **Summary**

- **Spark**
  - Transformations
  - Actions

  - On Base RDDs

  - Starting to get excited about spark?

Part 2: Spark Intro and Basics

- **Part 2: Spark Intro and basics**
  - Base RDD
  - Fault tolerance (and lineage)
  - Transformations and Actions
  - Persistance
  - Animated Example
  - Pair RDDs
  - Word count example

Example: filter Error lines from logfile

HDFS file

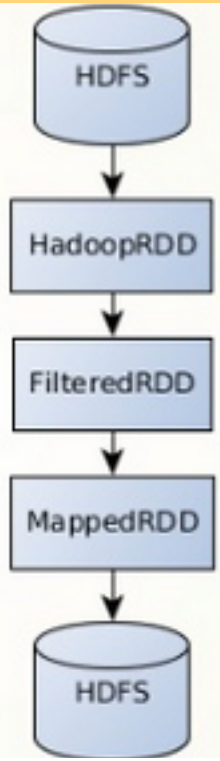
```
INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle
```

pySpark

```
sc.textFile("hdfs://<some input path or any local file")  \
        .filter(lambda x: 'ERROR' in x) \          # Error anywhere in the line
        .map(lambda line: line.split(" ")[1])      #second word and the original line
        .saveAsTextFile("hdfs://<output>  or any local directory")
```

Scala

```
sc.textFile("hdfs://<input>")
  .filter(_.startsWith("ERROR"))
  .map(_.split(" ")(1))
  .saveAsTextFile("hdfs://<output>")
```



Data on loca
nodes or will
be transferre
in

Example: filter Error lines from logfile

HDFS file

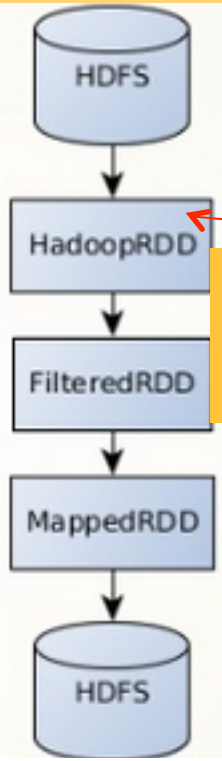
```
INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle
```

pySpark

```
sc.textFile(“hdfs://<some input path or any local file”)  \
        .filter(lambda x: 'ERROR' in x) \          # Error anywhere in the line
        .map(lambda line: line.split(" ")[1])      #second word and the original line
        .saveAsTextFile(“hdfs://<output>  or any local directory”)
```

Scala

```
sc.textFile("hdfs://<input>")
  .filter(_.startsWith("ERROR"))
  .map(_.split(" ")(1))
  .saveAsTextFile("hdfs://<output>")
```



Data on local nodes or will b transferred in

Example: filter Error lines from logfile

HDFS file

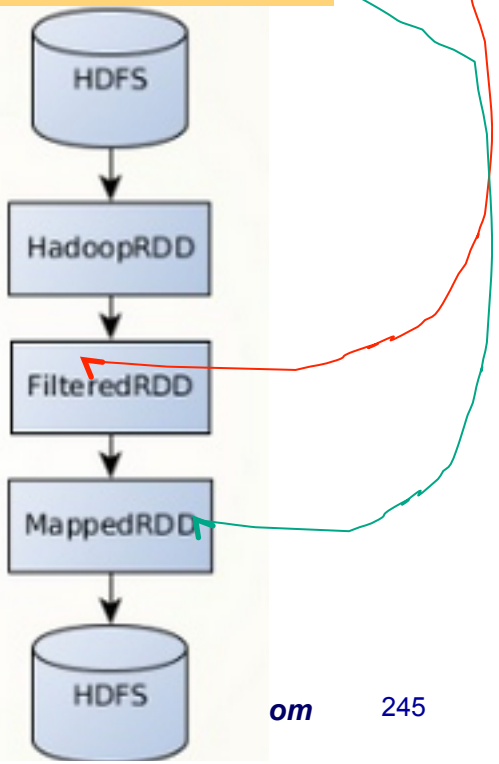
```
INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle
```

pySpark

```
sc.textFile(“hdfs://<some input path or any local file”)  \
        .filter(lambda x: 'ERROR' in x) \          # Error anywhere in the line
        .map(lambda line: line.split(" ")[1])      #second word and the original line
        .saveAsTextFile(“hdfs://<output>  or any local directory”)
```

Scala

```
sc.textFile("hdfs://<input>")
  .filter(_.startsWith("ERROR"))
  .map(_.split(" ")(1))
  .saveAsTextFile("hdfs://<output>")
```



saveAsTextFile Action: Step by Step

HDFS file

```
INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle
```

pySpark

```
sc.textFile("hdfs://<some input path or any local file")  \
        .filter(lambda x: 'ERROR' in x) \          # Error anywhere in the line
        .map(lambda line: line.split(" ")[1])      #second word and the original line
        .saveAsTextFile("hdfs://<output>  or any local directory")
```

The RDD.saveAsTextFile() action triggers a job. Tasks are started on scheduled executors.

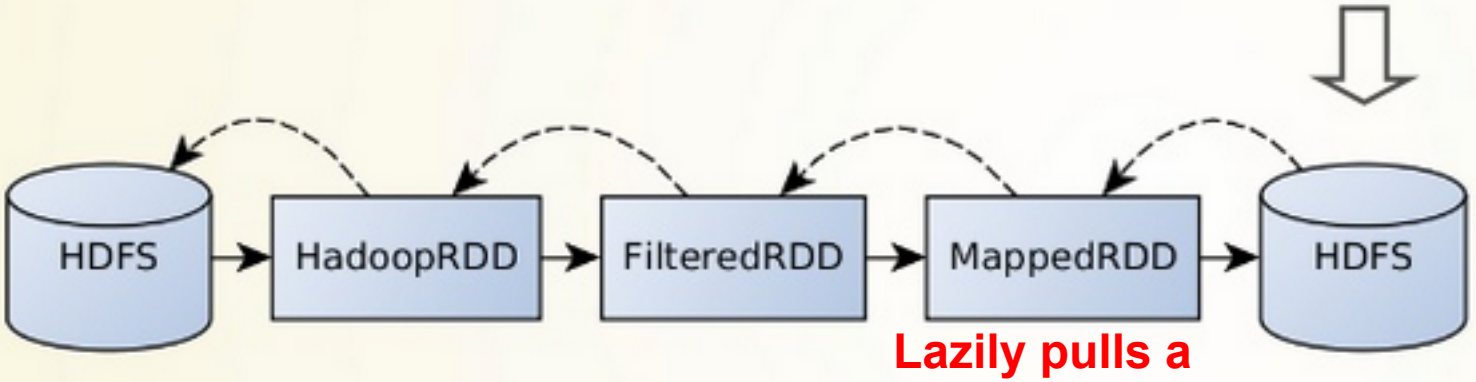
saveAsTextFile has a domino effect

HDFS file

```
INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle
```

pySpark

```
sc.textFile("hdfs://<some input path or any local file")  \
        .filter(lambda x: 'ERROR' in x) \          # Error anywhere in the line
        .map(lambda line: line.split(" ")[1])      #second word and the original line
        .saveAsTextFile("hdfs://<output>  or any local directory")
```

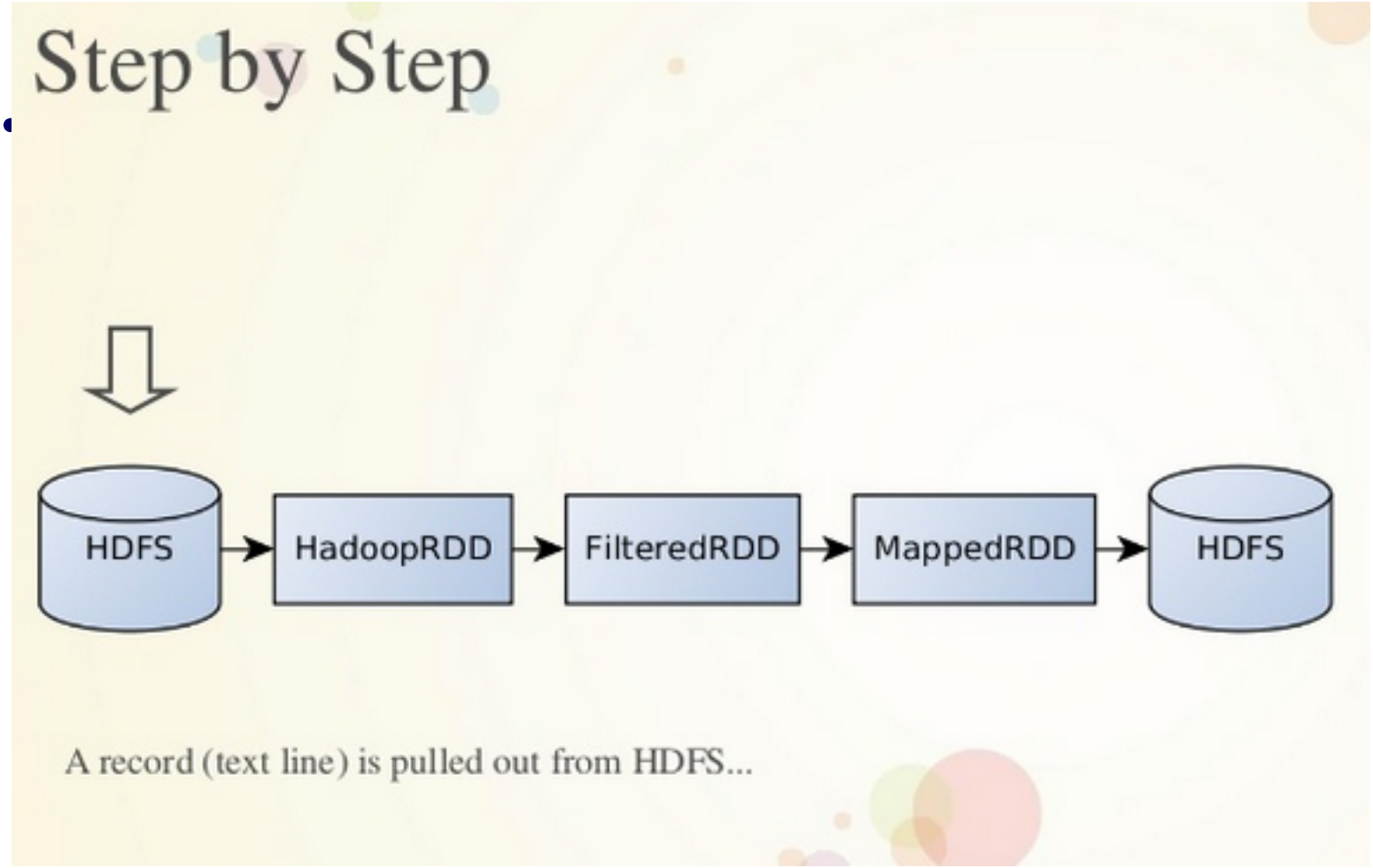


Lazily pulls a record from MappedRDD

Save a record

The task pulls data lazily from the final RDD (the MappedRDD here).

Chain effect right back to the source



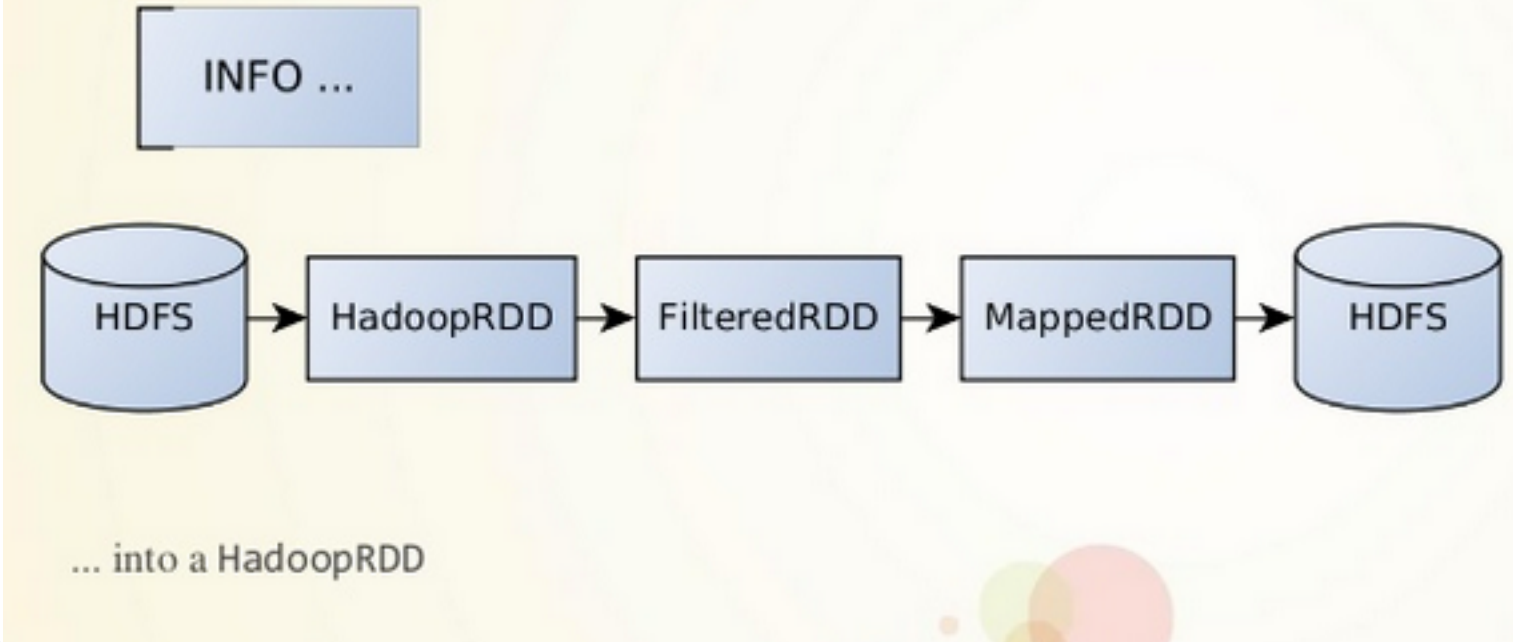
Materializes a record that is passed forward in the pipeline

HDFS file

```
INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle
```

pySpark

```
sc.textFile("hdfs://<some input path or any local file")  \
        .filter(lambda x: 'ERROR' in x) \          # Error anywhere in the line
        .map(lambda line: line.split(" ")[1])      #second word and the original line
        .saveAsTextFile("hdfs://<output>  or any local directory")
```



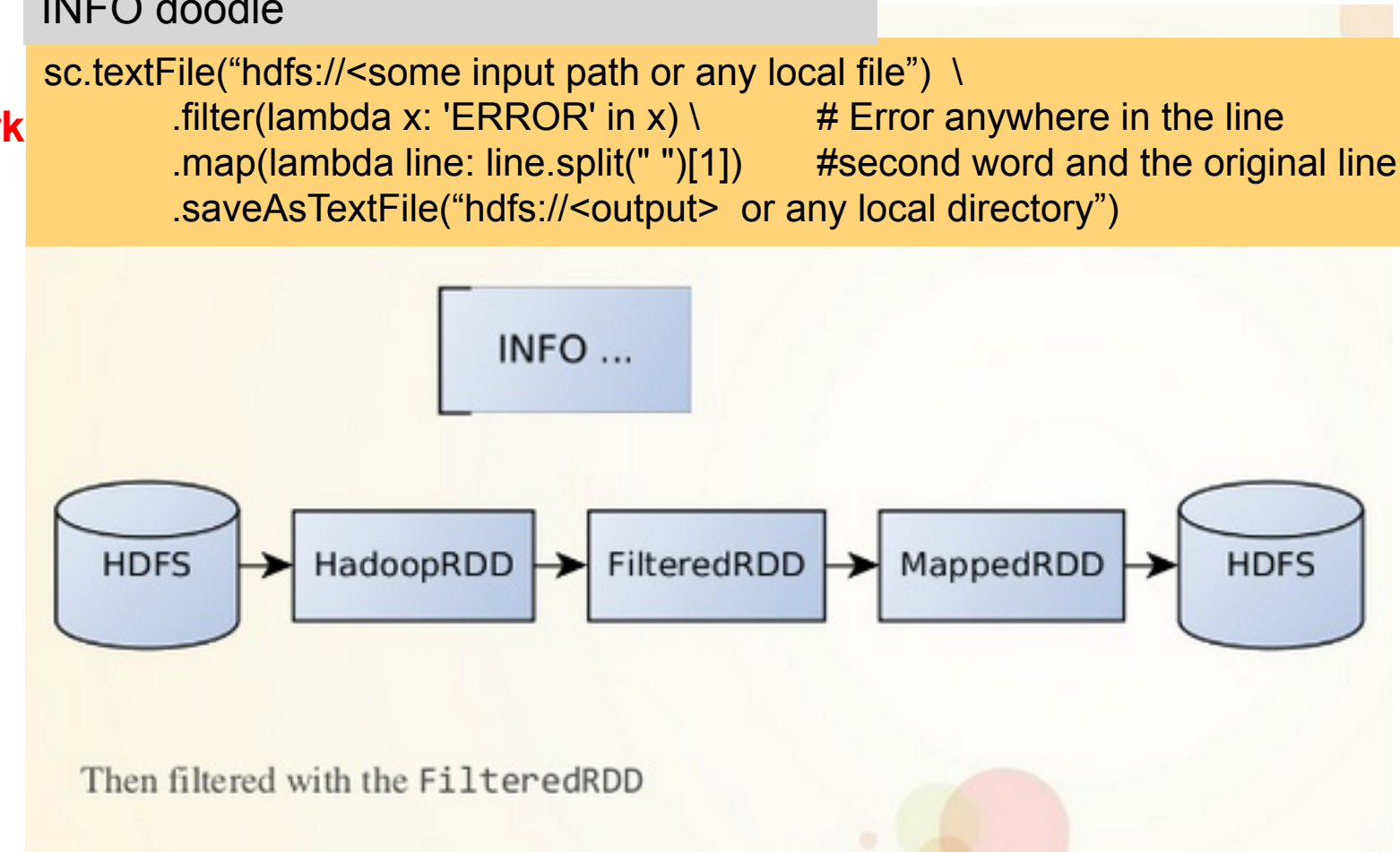
Record is passed thru the filter

HDFS file

INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle

pySpark

```
sc.textFile("hdfs://<some input path or any local file")  \
        .filter(lambda x: 'ERROR' in x) \          # Error anywhere in the line
        .map(lambda line: line.split(" ")[1])      #second word and the original line
        .saveAsTextFile("hdfs://<output>  or any local directory")
```



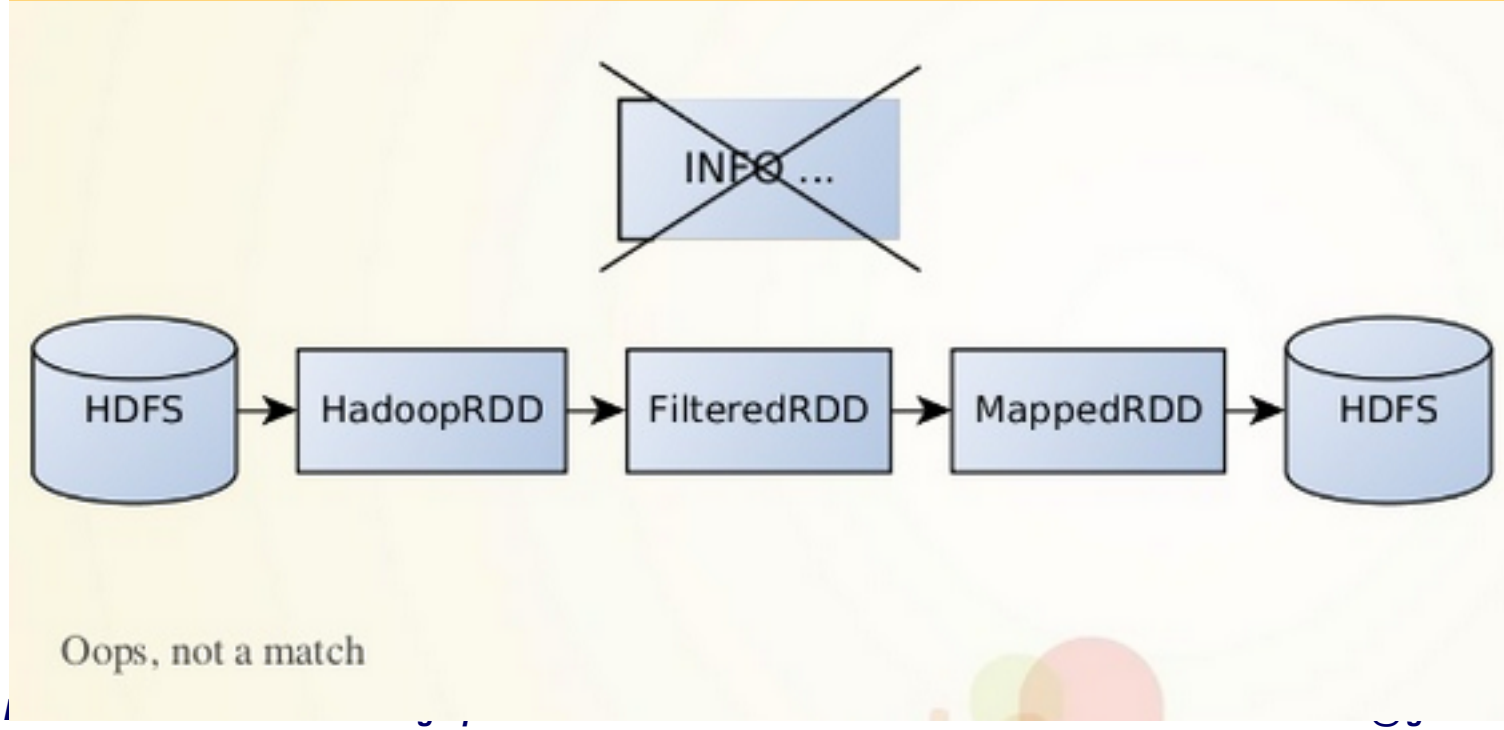
Record is passed thru the filter but Rejected

HDFS file

INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle

pySpark

```
sc.textFile("hdfs://<some input path or any local file")  \
        .filter(lambda x: 'ERROR' in x) \          # Error anywhere in the line
        .map(lambda line: line.split(" ")[1])      #second word and the original line
        .saveAsTextFile("hdfs://<output>  or any local directory")
```



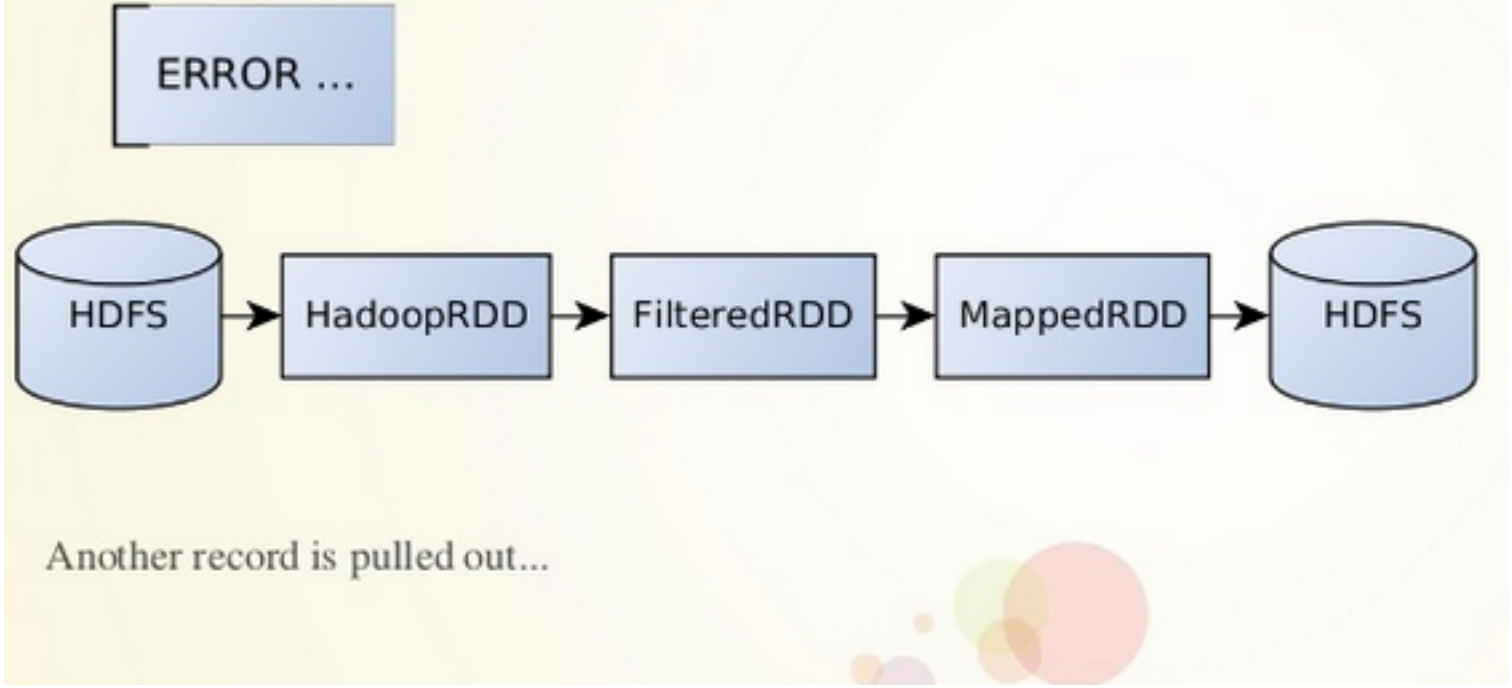
Action: Take 2

HDFS file

```
INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle
```

pySpark

```
sc.textFile("hdfs://<some input path or any local file")  \
        .filter(lambda x: 'ERROR' in x) \          # Error anywhere in the line
        .map(lambda line: line.split(" ")[1])      #second word and the original line
        .saveAsTextFile("hdfs://<output>  or any local directory")
```



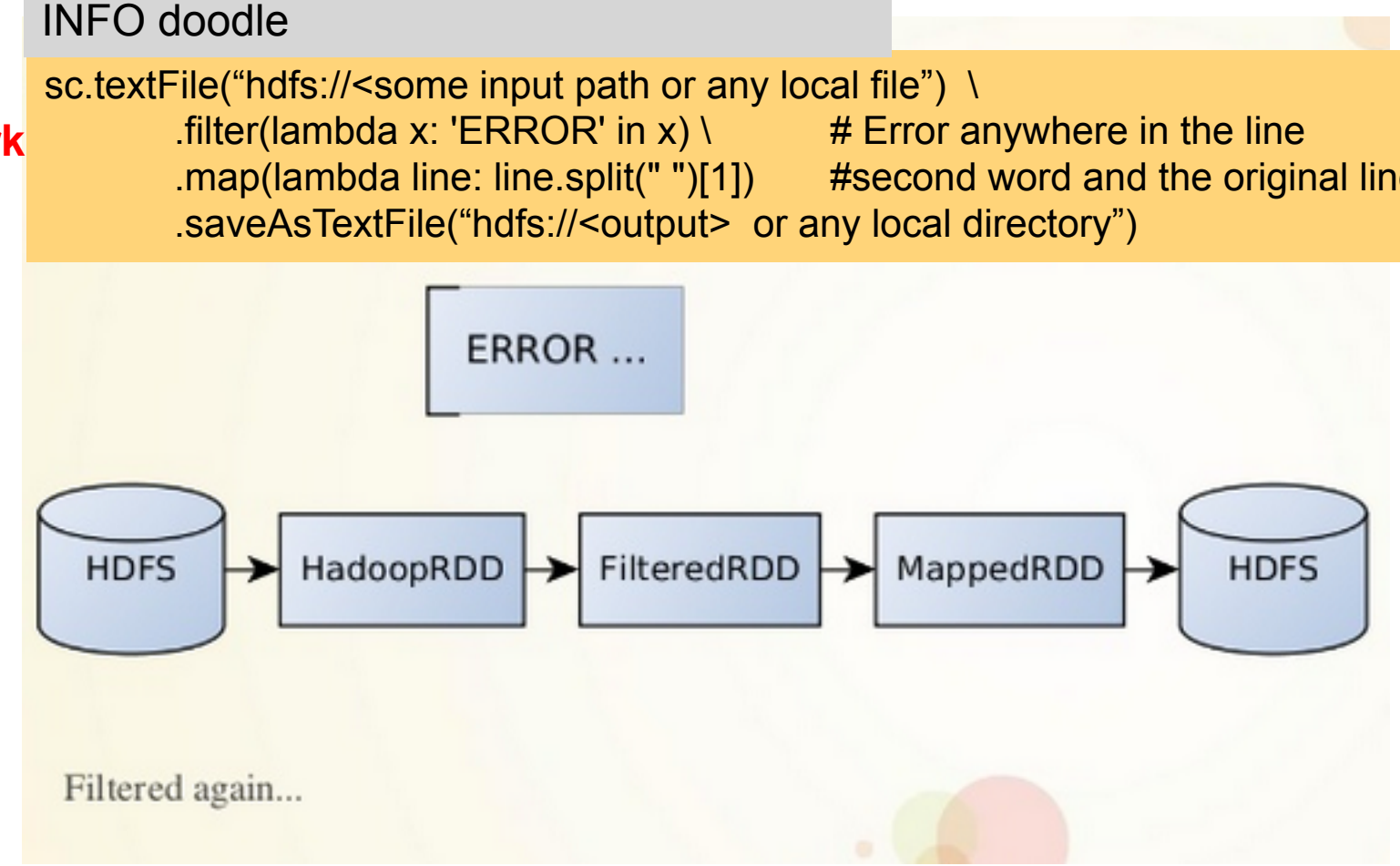
Pass second record thru filter

HDFS file

INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle

pySpark

```
sc.textFile("hdfs://<some input path or any local file")  \
        .filter(lambda x: 'ERROR' in x) \          # Error anywhere in the line
        .map(lambda line: line.split(" ")[1])      #second word and the original line
        .saveAsTextFile("hdfs://<output>  or any local directory")
```



Passes file and write to HDFS

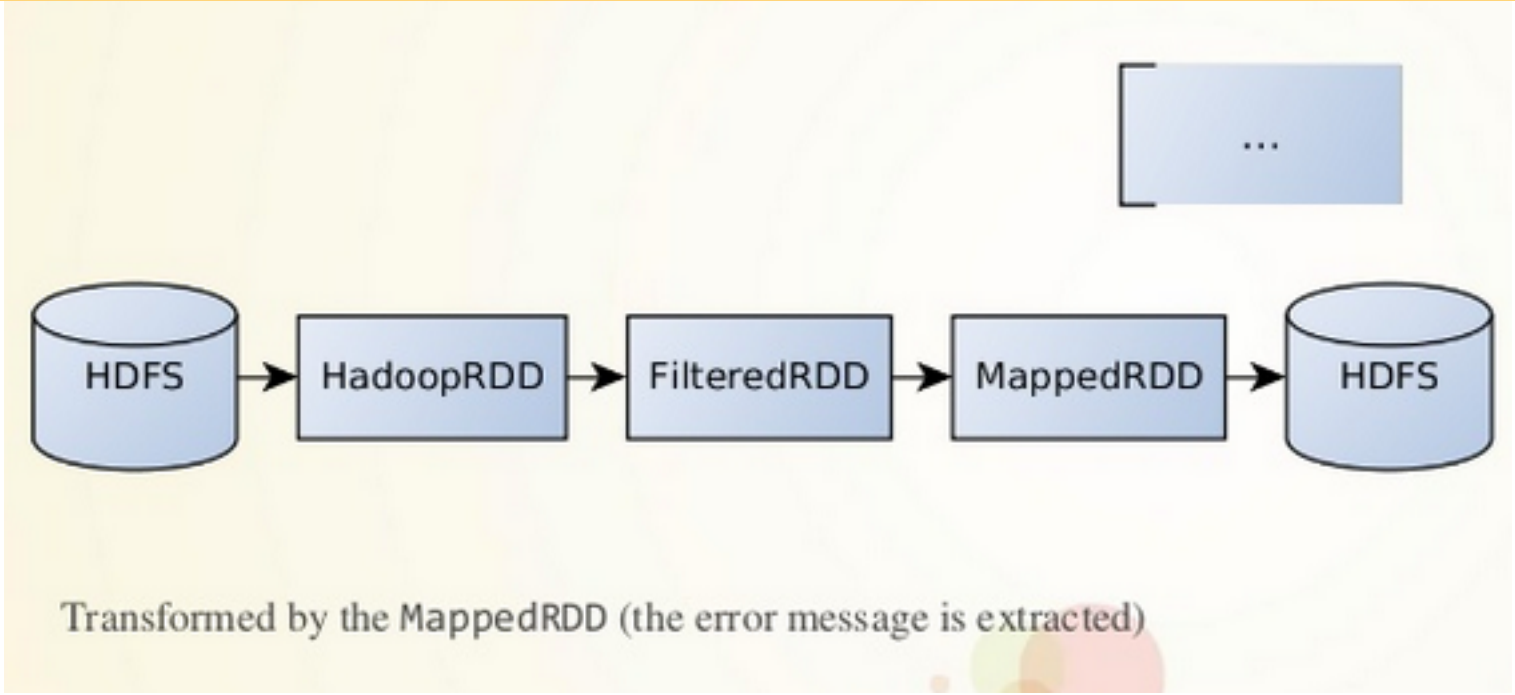
HDFS file

INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle

ERROR 30330 task aborted with no status

pySpark

```
sc.textFile("hdfs://<some input path or any local file")  \
        .filter(lambda x: 'ERROR' in x) \          # Error anywhere in the line
        .map(lambda line: line.split(" ")[1])      #second word and the original line
        .saveAsTextFile("hdfs://<output>  or any local directory")
```



A Synthesized Iterator

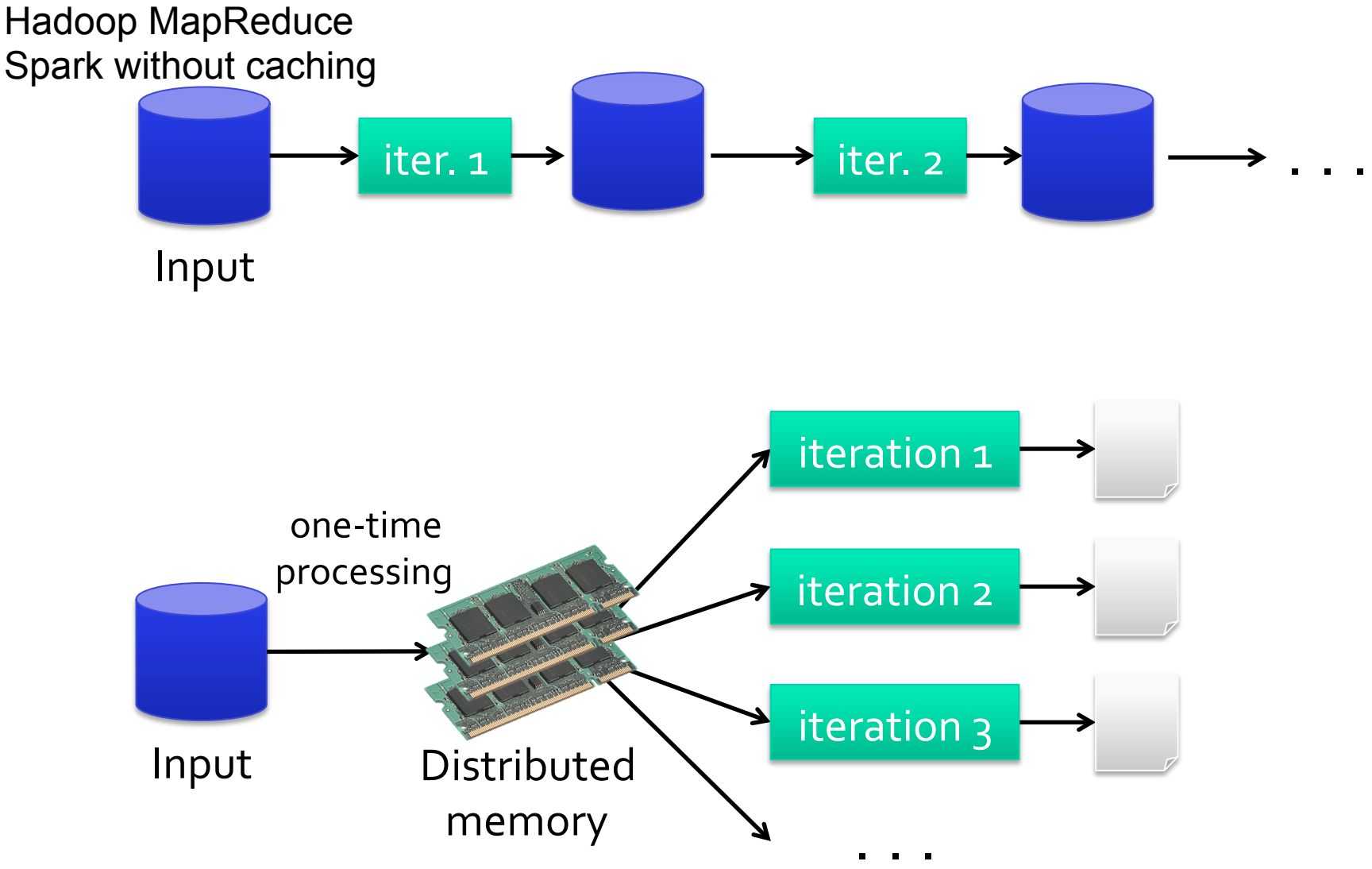

```
new Iterator[String] {
  private var head: String = _
  private var headDefined: Boolean = false

  def hasNext: Boolean = headDefined || {
    do {
      try head = readOneLineFromHDFS(...)      // (1) read from HDFS
      catch {
        case _: EOFException => return false
      }
    } while (!head.startsWith("ERROR"))        // (2) filter closure
    true
  }

  def next: String = if (hasNext) {
    headDefined = false
    head.split(" ")(1)                         // (3) map closure
  } else {
    throw new NoSuchElementException("...")
  }
}
```

- **Caching**

Goal: Keep Working Set in RAM



Caching

While the contents of RDDs are transient by default, Spark provides a mechanism persisting the data in an RDD. After the first time an action requires computing such an RDD's contents, they are stored in memory or disk across the cluster. The next time an action depends on the RDD, it need not be recomputed from its dependencies. Its data is returned from the cached partitions directly.

```
cached.cache()
cached.count()
cached.take(10)
```

The call to `cache` indicates that the RDD should be stored the next time it's computed. The call to `count` computes it initially. The `take` action returns the first 10 elements the RDD as a local `Array`. When `take` is called, it accesses the cached elements of `cached` instead of recomputing them from their dependencies.

Spark defines a few different mechanisms, or `StorageLevel` s, for persisting RDDs. `rdd.cache()` is shorthand for `rdd.persist(StorageLevel.MEMORY)`, which stores the RDD as unserialized Java objects. When Spark estimates that a partition will not fit in memory, it simply will not store it, and it will be recomputed the next time it's needed. This level makes the most sense when the objects will be referenced frequently and/or require low-latency access, as it avoids any serialization overhead. Its drawback is that it takes up larger amounts of memory than its alternatives. Also, holding on to many small objects puts pressure on Java's garbage collection, which can result in stalls and general slowness.

The In-Memory Magic

- With cache
  - One block per RDD partition
  - LRU cache eviction
  - Locality aware
  - Evicted blocks can be *recomputed in parallel* with the help of RDD lineage DAG

Cached intermediate (use by multiple downstream tasks if necessary instead of recalculating)

HDFS file

INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle

pySpark

```
cached= sc.textFile("hdfs://<some input path or any local file")  \
        .filter(lambda x: 'ERROR' in x).cache() \          # Error anywhere in the line

cached.map(lambda line: line.split(" ")[1])       #second word and the original line
        .saveAsTextFile("hdfs://<output>  or any local directory")
```

```
val cached = sc
  .textFile("hdfs://<input>")
  .filter(_.startsWith("ERROR"))
  .cache()

cached
  .map(_.split(" ")(1))
  .saveAsTextFile("hdfs://<output>")
```

ERRORs are cached

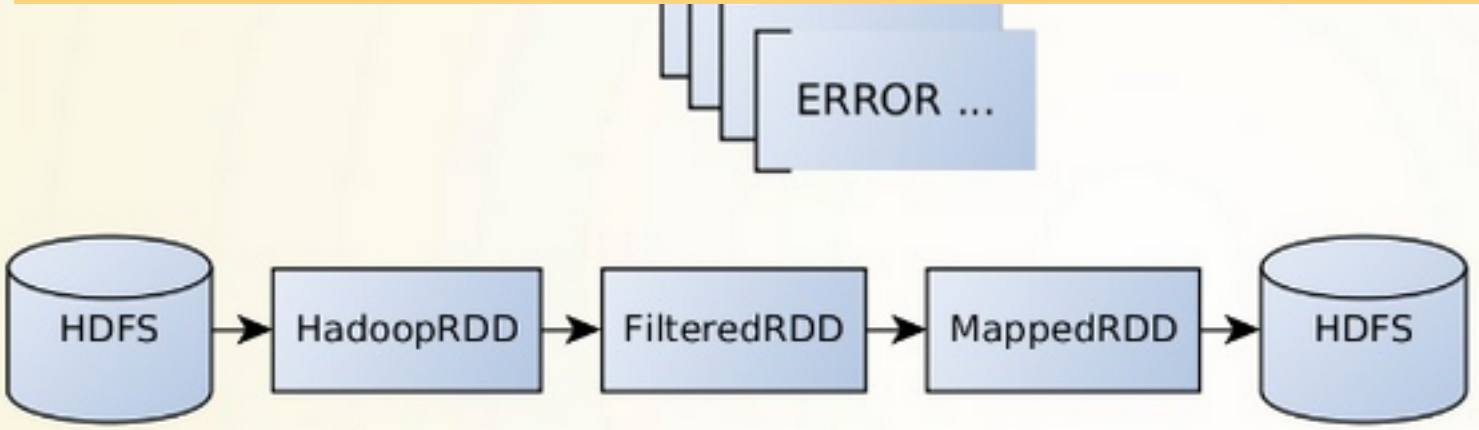
HDFS file

INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle

pySpark

```
cached= sc.textFile("hdfs://<some input path or any local file")  \
        .filter(lambda x: 'ERROR' in x).cache() \          # Error anywhere in the line

cached.map(lambda line: line.split(" ")[1])       #second word and the original line
        .saveAsTextFile("hdfs://<output>  or any local directory")
```



All filtered error messages are cached in memory before being passed to the next RDD. LRU cache eviction is applied when memory is insufficient

- **Documentation**

help(pyspark)

http://spark.apache.org/docs/latest/quick-start.html

- ..

Interactive Use

The `bin/pyspark` script launches a Python interpreter that is configured to run PySpark applications. To use `pyspark` interactively, first build Spark, then launch it directly from the command line without any options:

```
$ sbt/sbt assembly
$ ./bin/pyspark
```

The Python shell can be used explore data interactively and is a simple way to learn the API:

```
>>> words = sc.textFile("/usr/share/dict/words")
>>> words.filter(lambda w: w.startswith("spar")).take(5)
[u'spar', u'sparable', u'sparada', u'sparadrap', u'sparagrass']
>>> help(pyspark) # Show all pyspark functions
```

By default, the `bin/pyspark` shell creates SparkContext that runs applications locally on a single core. To connect to a non-local cluster, or use multiple cores, set the `MASTER` environment variable. For example, to use the `bin/pyspark` shell with a standalone Spark cluster:

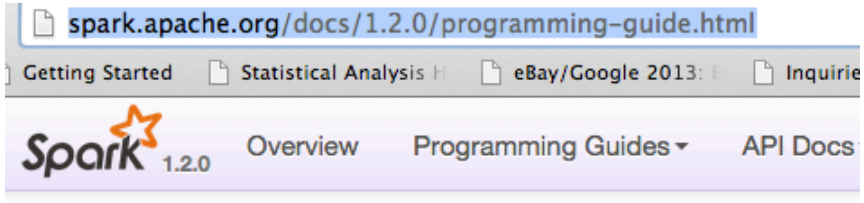
```
$ MASTER=spark://IP:PORT ./bin/pyspark
```

Or, to use four cores on the local machine:

```
$ MASTER=local[4] ./bin/pyspark
```

Spark in one-page!

- **http://spark.apache.org/docs/1.4.0/programming-guide.html**



Spark Programming Guide

Summary

- I hope this example was useful in helping you understand the power of Spark and to ground some of the concepts

bin/pyspark: Spark Shell

http://spark.apache.org/docs/latest/quick-start.html

Interactive Use

The `bin/pyspark` script launches a Python interpreter that is configured to run PySpark applications. To use `pyspark` interactively, first build Spark, then launch it directly from the command line without any options:

```
$ sbt/sbt assembly
$ ./bin/pyspark
```

The Python shell can be used explore data interactively and is a simple way to learn the API:

```
>>> words = sc.textFile("/usr/share/dict/words")
>>> words.filter(lambda w: w.startswith("spar")).take(5)
[u'spar', u'sparable', u'sparada', u'sparadrap', u'sparagrass']
>>> help(pyspark) # Show all pyspark functions
```

help(pyspark)

By default, the `bin/pyspark` shell creates SparkContext that runs applications locally on a single core. To connect to a non-local cluster, or use multiple cores, set the `MASTER` environment variable. For example, to use the `bin/pyspark` shell with a standalone Spark cluster:

```
$ MASTER=spark://IP:PORT ./bin/pyspark
```

Connect to a cluster versus a local

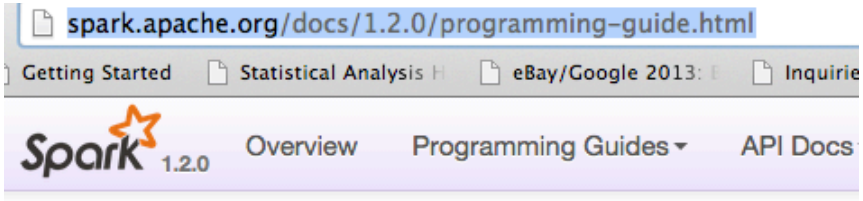
Or, to use four cores on the local machine:

```
$ MASTER=local[4] ./bin/pyspark
```

Connect to a 4 cores on the local machine

Spark in one-page!

- **http://spark.apache.org/docs/1.4.0/programming-guide.html**



Spark Programming Guide

Transformations

The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc (Scala, Java, Python) and pair RDD functions doc (Scala, Java) for details.

Transformation	Meaning
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).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
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.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numTasks*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `combineByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numTasks` argument to set a different number of tasks.
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 *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
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

Actions

The following table lists some of the common actions supported by Spark. Refer to the RDD API doc (Scala, Java, Python) and pair RDD functions doc (Scala, Java) for details.

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should 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. This is 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. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
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*) (Java and Scala)	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. This is available on RDDs of key-value pairs that either implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an accumulator variable (see below) or interacting with external storage systems.

Source Code available on GitHub

Part 2: Spark Intro and Basics

- **Part 2: Spark Intro and basics**
  - Base RDD
  - Fault tolerance (and lineage)
  - Transformations and Actions
  - Persistance
  - Animated Example
  - Pair RDDs
  - Word count example

PAIR RDDs

Key/value RDDs expose new operations (e.g., counting up reviews for each product, grouping together data with the same key, and grouping together two different RDDs).

Pair RDDs

This section covers how to work with RDDs of key/value pairs, which are a common data type required for many operations in Spark.

Key/value RDDs are commonly used to perform aggregations, and often we will do some initial ETL (extract, transform, and load) to get our data into a key/value format.

Key/value RDDs expose new operations (e.g., counting up reviews for each product, grouping together data with the same key, and grouping together two different RDDs).

Pair RDDs

We also discuss an advanced feature that lets users control the layout of pair RDDs across nodes: partitioning.

Using controllable partitioning, applications can sometimes greatly reduce communication costs by ensuring that data will be accessed together and will be on the same node.

JOINS: (see in later sections in this lecture and sunsequent lectures)

- This can provide significant speedups. We illustrate partitioning using the PageRank algorithm as an example.
- Choosing the right partitioning for a distributed dataset is similar to choosing the right data structure for a local one—in both cases, data layout can greatly affect performance.

Pair RDDs: Key value pairs

- **Spark provides special operations on RDDs containing key/ value pairs. These RDDs are called pair RDDs.**
- **Act on each key in parallel or regroup data across the network**
  - Pair RDDs are a useful building block in many programs, as they expose operations that allow you to act on each key in parallel or regroup data across the network.
- **reduceByKey(): aggregate data separately for each key**
  - For example, pair RDDs have a reduceByKey() method that can aggregate data separately for each key, and a join() method that can merge two RDDs together by grouping elements with the same key.
- **It is common to extract fields from an RDD (representing, for instance, an event time, customer ID, or other identifier) and use those fields as keys in pair RDD operations**

Pair RDDs Example

The way to build key-value RDDs differs by language. In Python, for the functions on keyed data to work we need to return an RDD composed of tuples (see Example 4-1).

Example 4-1. Creating a pair RDD using the first word as the key in Python

KEY VALUE

```
pairs = lines.map(lambda x: (x.split(" ")[0], x))
```

When creating a pair RDD from an in-memory collection in Scala and Python, we only need to call SparkContext.parallelize() on a collection of pairs.

Pair RDDs are allowed to use all the transformations available to standard RDDs. The same rules apply from "Passing Functions to Spark" on page 30 [Learning spark book].

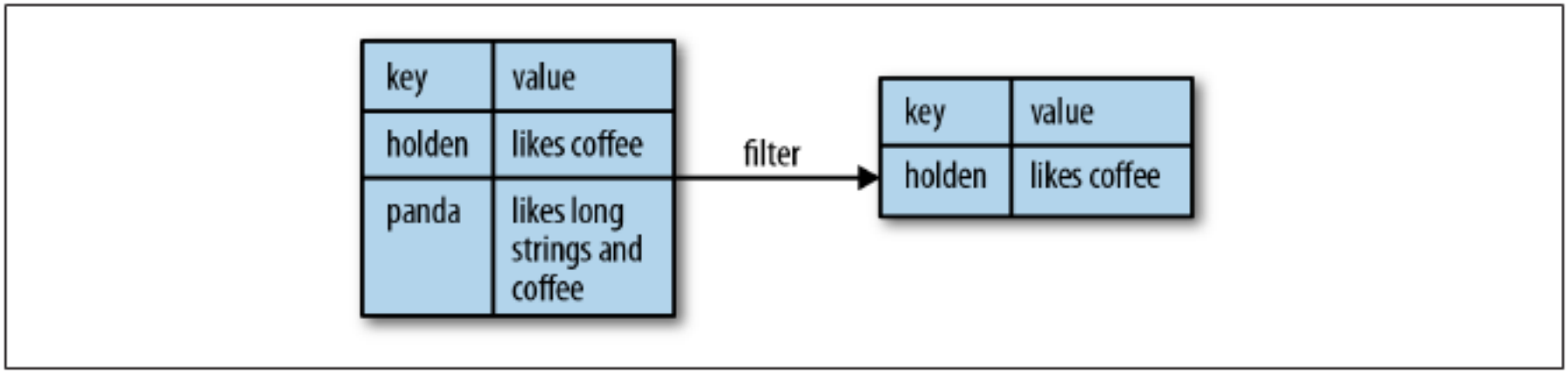


Figure 4-1. Filter on value

Pair RDDs are also still RDDs (of Tuple2 objects in Java/Scala or of Python tuples), and thus support the same functions as RDDs. For instance, we can take our pair RDD from the previous section and filter out lines longer than 20 characters, as shown in Examples 4-4 through 4-6 and Figure 4-1.

Filter on Value being < 20

Example 4-4. Simple filter on second element in Python

```
result = pairs.filter(lambda keyValue: len(keyValue[1]) < 20)
```

Example 4-5. Simple filter on second element in Scala

```
pairs.filter{case (key, value) => value.length < 20}
```

Part 2: Spark Intro and Basics

- **Part 2: Spark Intro and basics**
  - Base RDD
  - Fault tolerance (and lineage)
  - Transformations and Actions
  - Persistance
  - Animated Example
  - Pair RDDs
  - Word count example

Example 3

Simple Spark Apps: *WordCount*

Definition:

count how often each word appears in a collection of text documents

This simple program provides a good test case for parallel processing, since it:

- requires a minimal amount of code
- demonstrates use of both symbolic and numeric values
- isn't many steps away from search indexing
- serves as a "Hello World" for Big Data apps

```
void map (String doc_id, String text):
  for each word w in segment(text):
    emit(w, "1");


void reduce (String word, Iterator group):
  int count = 0;

  for each pc in group:
    count += Int(pc);

  emit(word, String(count));
```

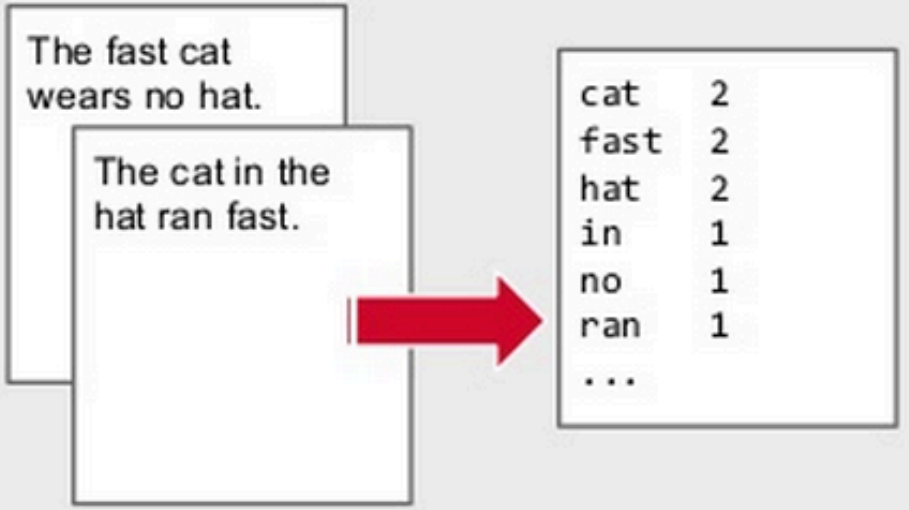
A distributed computing framework that can run WordCount **efficiently in parallel at scale** can likely handle much larger and more interesting compute problems

Word Count in Map-Reduce

```
def map(key, value):
    for word in value.split():
        emit(word, 1)

def reduce(key, values):
    count = 0
    for val in values:
        count += val
    emit(key, count)

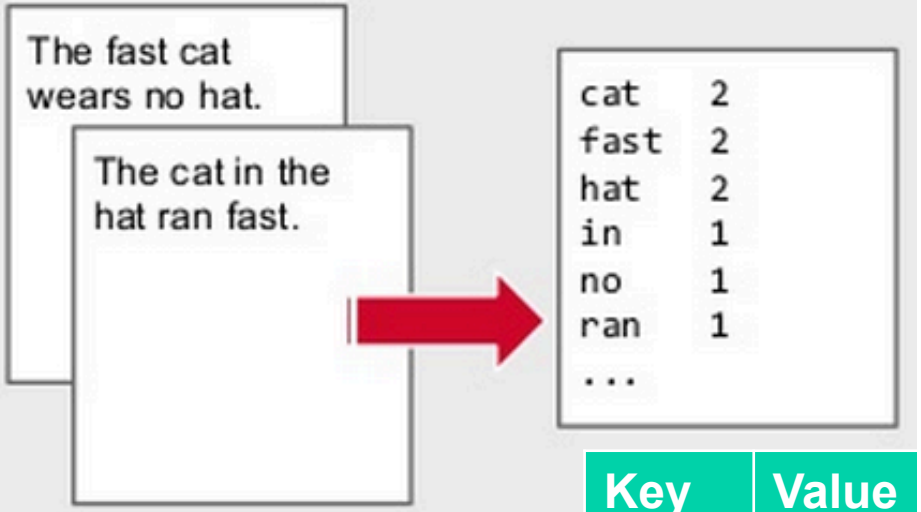
# emit is a function that performs distributed I/O
```



Each document is passed to a mapper, which does the tokenization. The output of the mapper is reduced by key (word) and then counted.

What is the data flow for word count?

Word Frequency




```
from operator import add

def tokenize(text):
    return text.split()

text = sc.textFile("tolstoy.txt")    # Create RDD

# Transform
wc   = text.flatMap(tokenize)
wc   = wc.map(lambda x: (x,1)).reduceByKey(add)

wc.saveAsTextFile("counts")          # Action
```

Key	Value
the	1
fast	1
wears	1
cat	1
….	…

Word Count

See NOTEBOOK

Write some words into a file

```
%%writefile wordcount.txt
hello hi hi hallo
bonjour hola hi ciao
nihao konnichiwa ola
hola nihao hello
```

```
Overwriting wordcount.txt
```

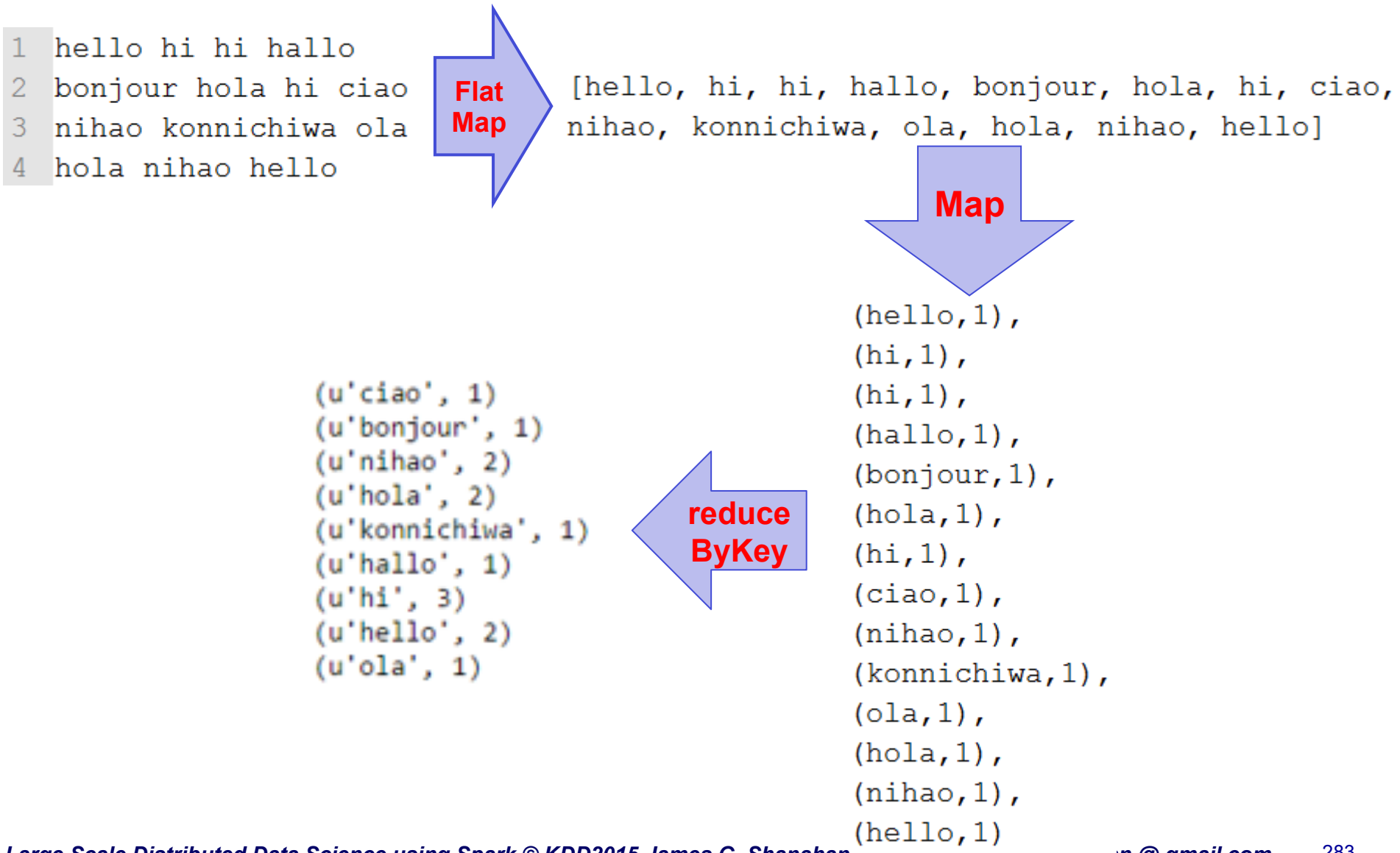
Word Count

```
#Count words in README.md
logFileNAME = 'wordcount.txt'
text_file = sc.textFile(logFileNAME)
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
for v in counts.collect():
    print v
```

```
(u'ciao', 1)
(u'bonjour', 1)
(u'nihao', 2)
(u'hola', 2)
(u'konnichiwa', 1)
(u'hallo', 1)
(u'hi', 3)
(u'hello', 2)
(u'ola', 1)
```

Attention:
flatMap works applying a function that returns a sequence for each element in the list, and flattening the results into the original list.

Word Count broken down



```
]: #Example 4-1. Creating a pair RDD using the first word as the key in Python

   lines = sc.parallelize(["Data line 1", "Mining line 2", "data line 3", "Data line 4", "Data Mining line 5"])
   pairs = lines.map(lambda x: (x.split(" ")[0], x))  #first word and the original line
   pairs.collect()
```

```
]: [('Data', 'Data line 1'),
    ('Mining', 'Mining line 2'),
    ('data', 'data line 3'),
    ('Data', 'Data line 4'),
    ('Data', 'Data Mining line 5')]
```

```
]:
```

In [15]:
```
#Paired RDD examples
logFileName='log.txt'
#Word count for error messages exercise
# READ Lines And PRINT
recs = sc.textFile(logFileName)
#recs = sc.textFile(logFileName).filter(lambda l: "ERROR" in l)
wcErrors = recs.flatMap(lambda l: l.split(" ")) \
                .map(lambda word: (word, 1)) \
                .reduceByKey(lambda x, y: x + y)
for (key, value) in wcErrors.collect():
    print "WordCount-->> %s:%d"%(key, value)
```

```
WordCount-->> :1
WordCount-->> dying:1
WordCount-->> for:1
WordCount-->> reasons:1
WordCount-->> ERROR     mysql:1
WordCount-->> angry:1
WordCount-->> cluster::1
WordCount-->> context:1
WordCount-->> spark:2
WordCount-->> with:1
WordCount-->> you:1
WordCount-->> me?:1
WordCount-->> WARN      dave,:1
WordCount-->> ERROR     php::1
WordCount-->> unknown:1
WordCount-->> it:1
WordCount-->> replace:1
WordCount-->> cluster:1
WordCount-->> missing...darn:1
WordCount-->> WARN      xylons:1
WordCount-->> at:1
WordCount-->> ERROR     :1
WordCount-->> approaching:1
WordCount-->> are:1
```

Word count notebook

Wordcount Notebook

DEMO

- **Download the following notebook**
  - https://www.dropbox.com/s/ul0l3q98w54dr8x/WordCount.ipynb?dl=0
- **Cd to the directory that contains the wordcount notebook**
  - cd /Users/jshanahan/Dropbox/NativeX-Internal/Publications/WSDM-2016
- **Launch Notebook**
  - /Users/jshanahan/anaconda/bin/ipython notebook&

localhost:8888/notebooks/Notebooks/WordCount/WordCount.ipynb#

jupyter WordCount (autosaved)

File Edit View Insert Cell Kernel Help Python 2

Code Cell Toolbar: None

In [2]:
```
import os
import sys
#spark_home = os.environ['SPARK_HOME'] = '/Users/liang/Downloads/spark-1.4.1-bin-hadoop2.6/'
spark_home = os.environ['SPARK_HOME'] = '/Users/jshanahan/Dropbox/Lectures-UC-Berkeley-ML-Class-2015/spark-1.5.0-bin-hadoop2.6'


if not spark_home:
    raise ValueError('SPARK_HOME enviroment variable is not set')
sys.path.insert(0,os.path.join(spark_home,'python'))
sys.path.insert(0,os.path.join(spark_home,'python/lib/py4j-0.8.2.1-src.zip'))
execfile(os.path.join(spark_home,'python/pyspark/shell.py'))
```

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.5.0
      /_/

Using Python version 2.7.11 (default, Dec  6 2015 18:57:58)
SparkContext available as sc, HiveContext available as sqlContext.
```

In [3]:
```
%%writefile wordcount.txt
hello hi hi hallo
bonjour hola hi ciao
nihao konnichiwa ola
hola nihao hello
```

```
Writing wordcount.txt
```

In [4]:
```
cat wordcount.txt
```

```
hello hi hi hallo
bonjour hola hi ciao
nihao konnichiwa ola
hola nihao hello
```

In [ ]:

In [3]:
```
#Count words in README.md
logFileNAME = 'wordcount.txt'
text_file = sc.textFile(logFileNAME)
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
for v in counts.collect():
    print v
```

```
(u'ciao', 1)
(u'bonjour', 1)
```