Northeastern University

ALY 6110
Data Management and Big Data

Week 4

Agenda:

- Introduction to Spark;
- Key Modules;
- How does Spark work?
- Spark vs Map Reduce;
- Spark vs Hadoop;
- Installation of Spark;
- Load data into Spark;
- Building Spark Application using Maven;

What is Spark?

Apache Spark is an open-source distributed general-purpose cluster-computing framework

Introduction to Spark

Introduction to Spark

- Spark extends MapReduce model to efficiently support more types of computations, including interactive queries and stream processing;
- Spark is designed to cover a wide range of workloads that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries, and streaming;
- By supporting these workloads in the same engine, Spark makes it easy and inexpensive to combine different processing types.

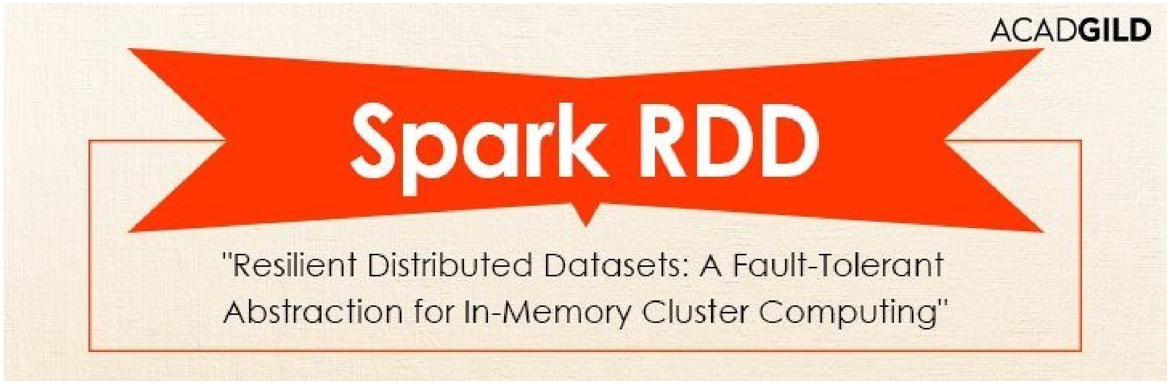
Introduction to Spark

- Spark is designed to be highly accessible, offering simple APIs in:
  - Python;
  - Java;
  - Scala;
  - SQL;
  - rich built-in libraries;
- Spark integrates closely with other Big Data tools;
- Spark can run in Hadoop clusters and access any Hadoop data source, including Cassandra.

Introduction to Spark

- Spark is an open-source software solution that performs rapid calculations on in-memory distributed datasets;

- In-memory distributed datasets are referred to as RDDs;


ACADGILD
Spark RDD
"Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing"

Introduction to Spark

- RDDs are Resilient Distributed Datasets;
- RDD is the key Spark concept and the basis for what Spark does;
- RDD is a distributed collections of objects that can be cached in memory across cluster and can be manipulated in parallel;
- RDD could be automatically recomputed on failure;

Introduction to Spark

- RDD is resilient – can be recreated on the fly from known state;
- Immutable – already defined RDDs can be used as a basis to generate derivative RDDs but are never mutated;
- Distributed – the dataset is often partitioned across multiple nodes for increased scalability and parallelism;

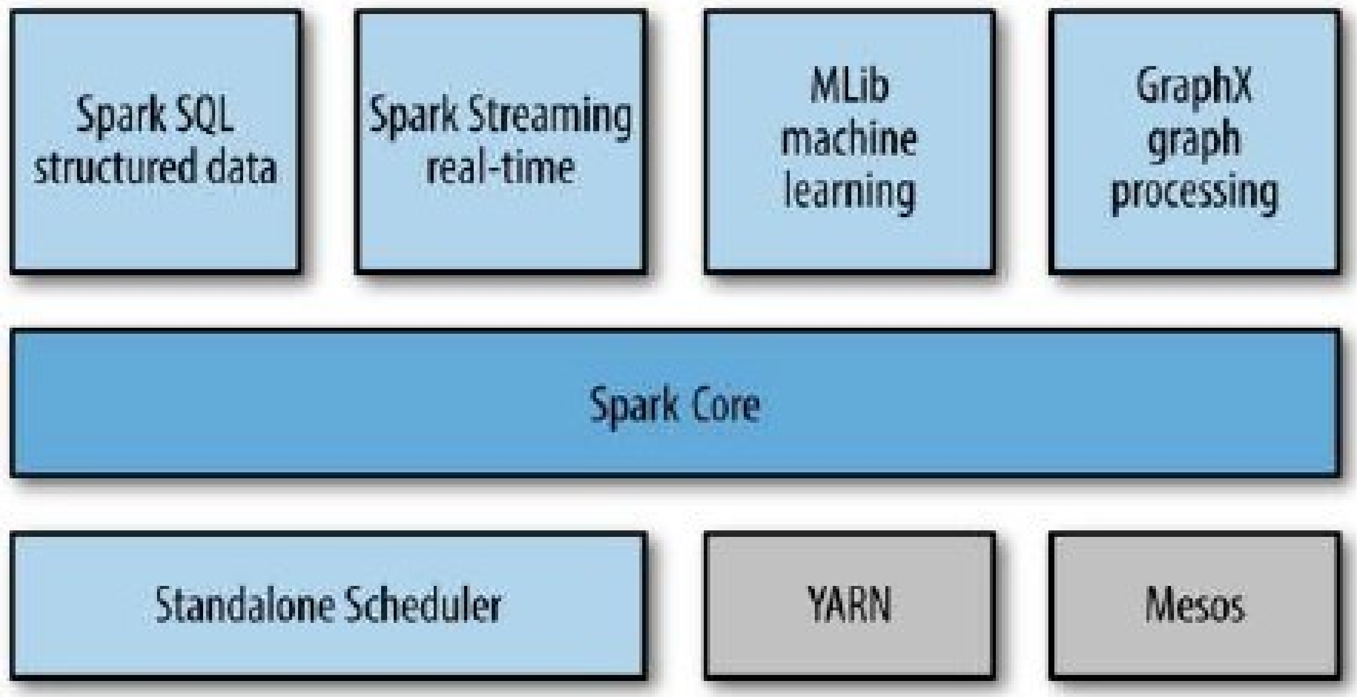
Introduction to Spark

- Spark is a fast general processing engine for large scale data processing;
- Spark is designed for iterative computations and interactive data mining;
- Spark supports use of well known languages:
  - Python;
  - Scala;
  - Java;
  - R
- With Spark streaming the same code can be used on data at rest and data in motion

Key Modules

Key Modules

Spark has several key modules



Key Modules

First:

- **Spark Core** contains the basic functionality of Spark, including components:
  - Task scheduling;
  - Memory management;
  - Fault recovery;
  - Interacting with storage systems;
  - … and others;

- **Spark Core** is the home to the API that defines resilient distributed datasets (RDDs), which are Spark's main programming abstraction;

- RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel.

Key Modules

Second:

- Spark SQL is Spark's package for working with structured data;
- Spark SQL allows querying data via SQL as well as the Apache Hive variant of SQL—Hive Query Language (HQL)—and it supports many sources of data, including Hive tables, Parquet, and JSON;
- Spark SQL allows developers to intermix SQL queries with the programmatic data manipulations supported by RDDs in Python, Java, and Scala, all within a single application, thus combining SQL with complex analytics.

Key Modules

Third:

- Spark Streaming is a Spark component that enables processing of live streams of data;
- Data streams include log files of web servers, or queues of messages.
- Spark Streaming API for manipulating data streams closely matches the Spark Core's RDD API, making it easy to move between apps that manipulate data in memory, on disk, or arriving in real time;
- Spark Streaming provides the same degree of fault tolerance, throughput, and scalability as Spark Core;

Key Modules

Fourth:

- MLlib is a library containing common machine learning (ML) functionality;
- MLlib provides multiple types of machine learning algorithms, including classification, regression, clustering, and collaborative filtering, as well as supporting functionality such as model evaluation and data import;
- MLlib provides some lower-level ML primitives, including a generic gradient descent optimization algorithm;
- All of ML methods are designed to scale out across a cluster.

Key Modules

Fifth:

- GraphX is a library for manipulating graphs (e.g., a social network's friend graph) and performing graph-parallel computations;
- GraphX extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge;
- GraphX also provides various operators for manipulating graphs (e.g., subgraph and mapVertices) and a library of common graph algorithms (e.g., PageRank and triangle counting).

How does Spark work?

How does Spark work?

- RDD
  - Your data is loaded in parallel into structured collections
- Actions
  - Manipulate the state of the working model by forming new RDDs and performing calculations upon them
- Persistence
  - Long-term storage of an RDD's state

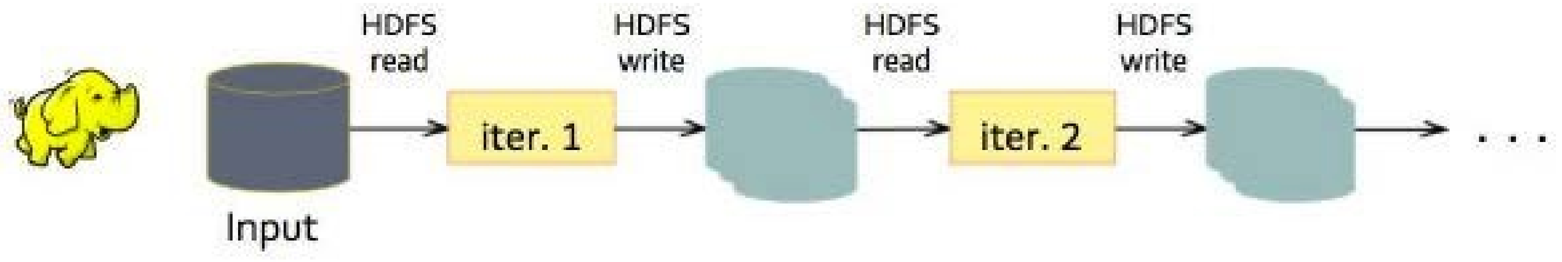
How does Spark work?

- Spark Application is a definition in code of
  - RDD creation
  - Actions
  - Persistence
- Spark Application results in the creation of a DAG (Directed Acyclic Graph)
  - Each DAG is compiled into stages
  - Each Stage is executed as a series of Tasks
  - Each Task operates in parallel on assigned partitions
  - It all starts with the SparkContext ‘sc’

Spark vs Map Reduce

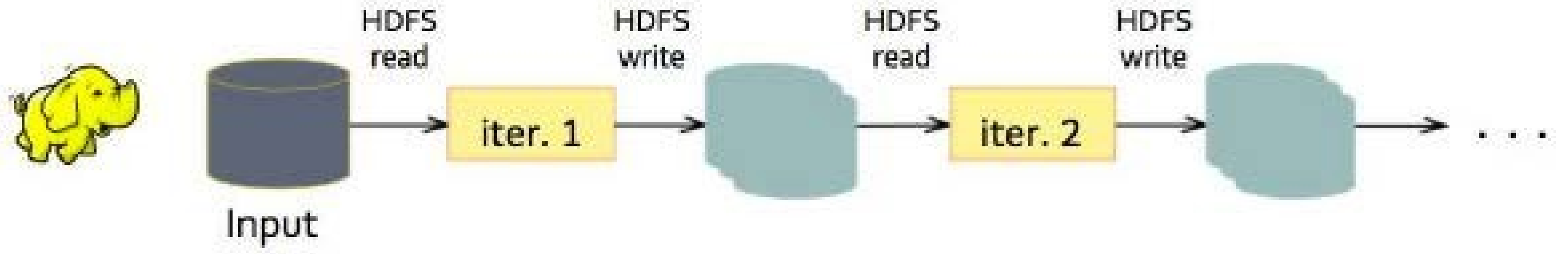
Spark vs Map Reduce

Map Reduce places every result on a disk

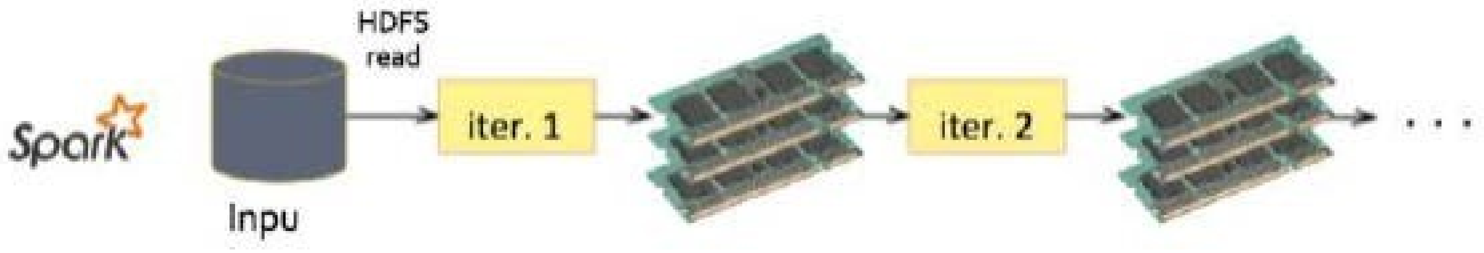


Spark vs Map Reduce

Map Reduce places every result on a disk



Spark is much smarter, it keeps all results in memory



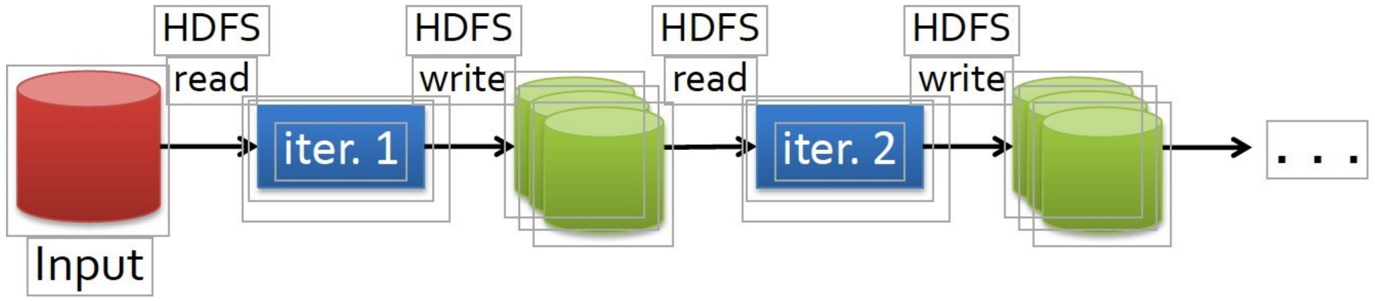
Spark vs Map Reduce

- MapReduce greatly simplified big data analysis;
- But as soon as it got popular, users wanted more:
    - More complex, multi-pass analytics (e.g. ML, graph);
    - More interactive ad-hoc queries;
    - More real-time stream processing;
    - All need faster data sharing across parallel jobs;

Spark vs Map Reduce

Map Reduce is slow due to replication, serialization and disk IO

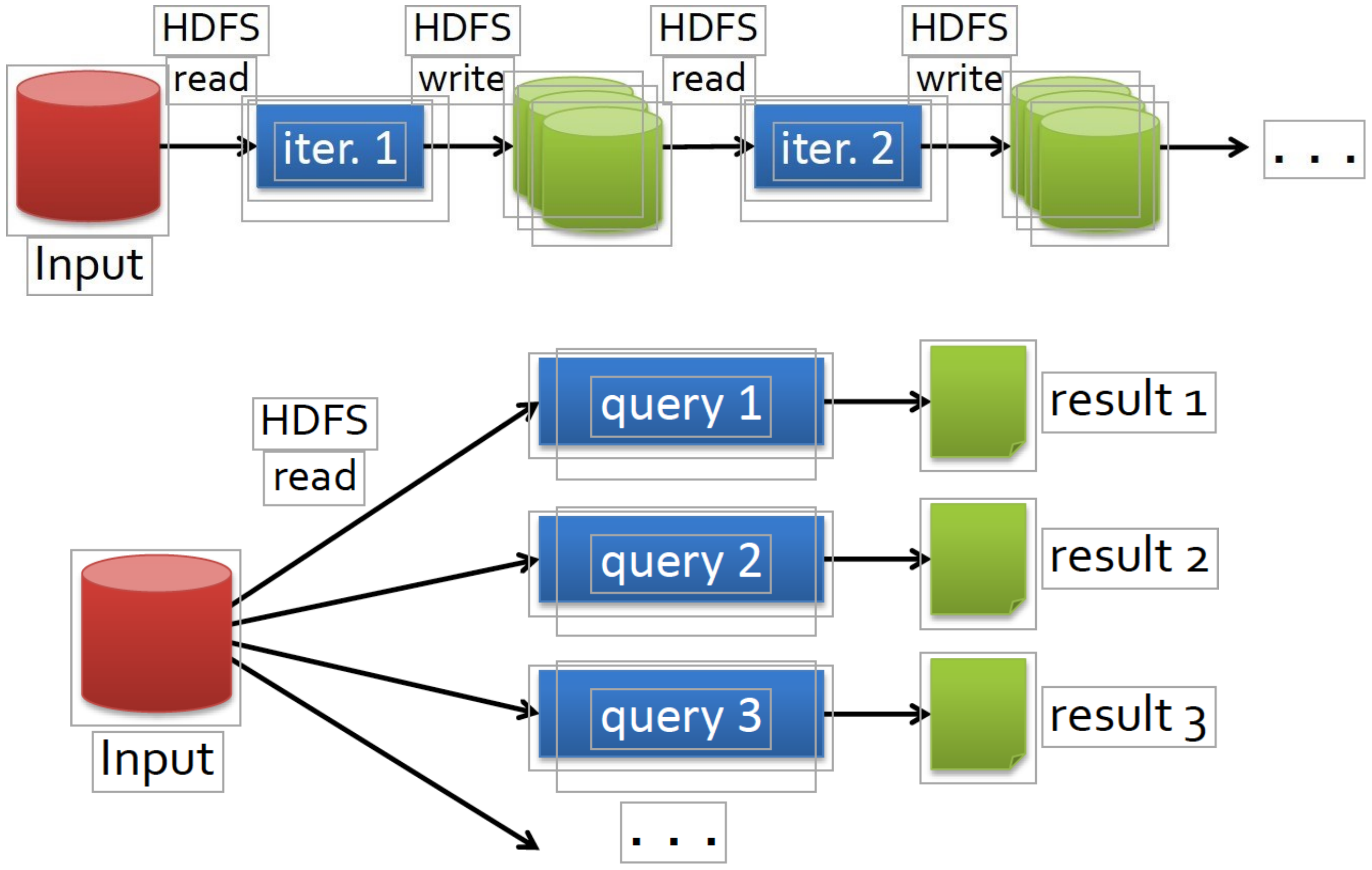
Data Sharing in Map Reduce



Spark vs Map Reduce

Map Reduce is slow due to replication, serialization and disk IO

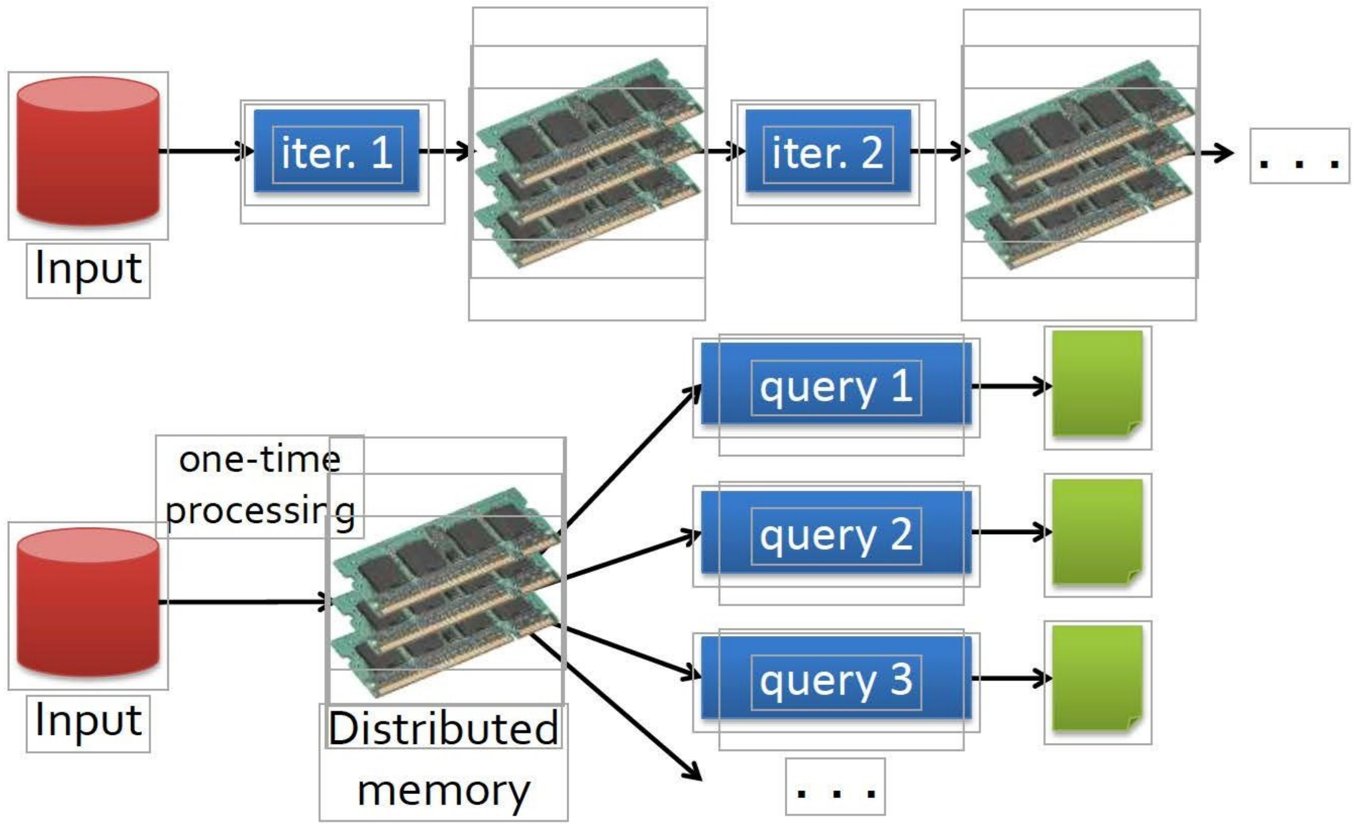
Data Sharing in Map Reduce



Spark vs Map Reduce

Distributed memory is 10-100× faster than network and disk

Data Sharing in Spark



Spark vs Map Reduce

- Spark extends the popular MapReduce model to efficiently support more types of computations, including interactive queries and stream processing;

- It is said that inventors of Spark noticed that Hadoop (MapReduce) was inefficient for the iterative workflows and they extended Hadoop architecture to make it more efficient;

- Speed is important in processing large datasets, as it means the difference between exploring data interactively and waiting minutes or hours;

Spark vs Map Reduce

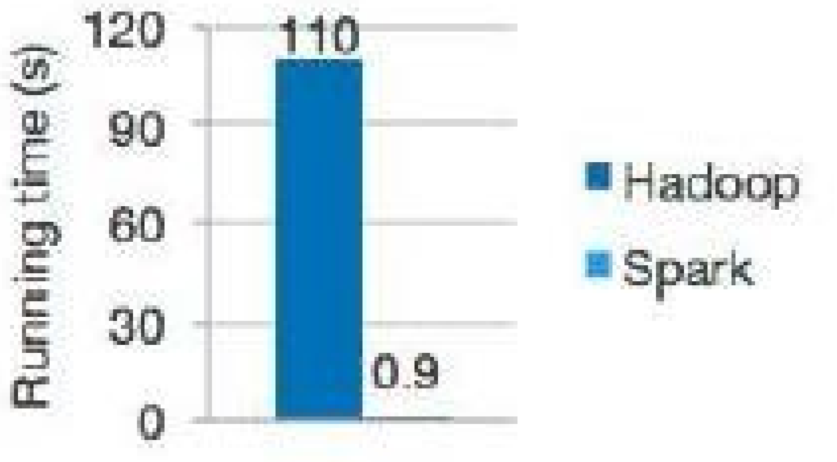
- Spark's speed comes from its ability to run computations in memory;
- Spark appears to be more efficient than MapReduce for complex applications running on disk;
- Spark retains the attractive properties of MapReduce:
    - fault tolerance;
    - data locality;
    - Scalability;

Spark vs Hadoop

Spark vs Hadoop

- Note and do not take them too seriously;

- Hadoop people will show you the results below which compare Impala, a Hadoop SQL engine, and Spark. Impala is 4 or 6 X faster,….

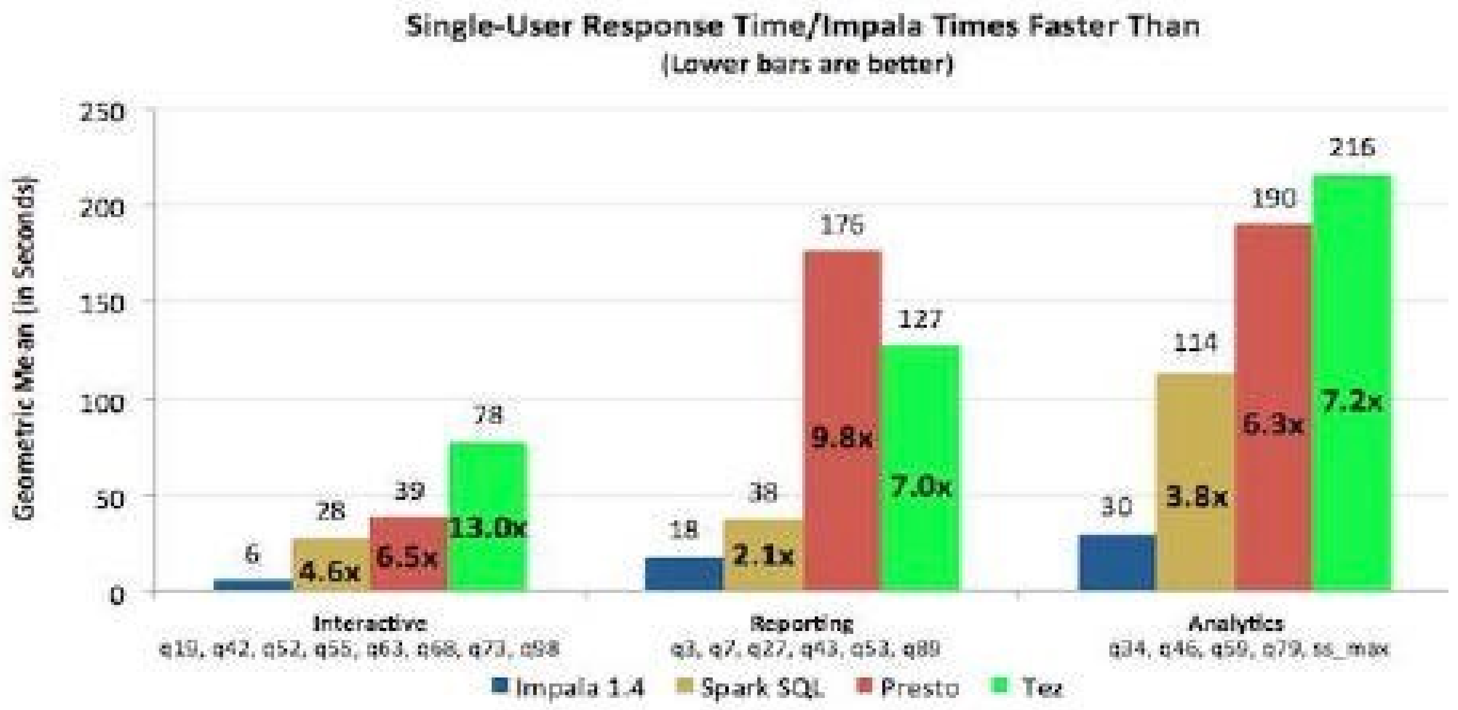


Logistic regression in Hadoop and Spark

Spark vs Hadoop

- Engineers developing Hadoop are learning from Spark just like Spark engineers learned from Hadoop;

- Cycles keep repeating!

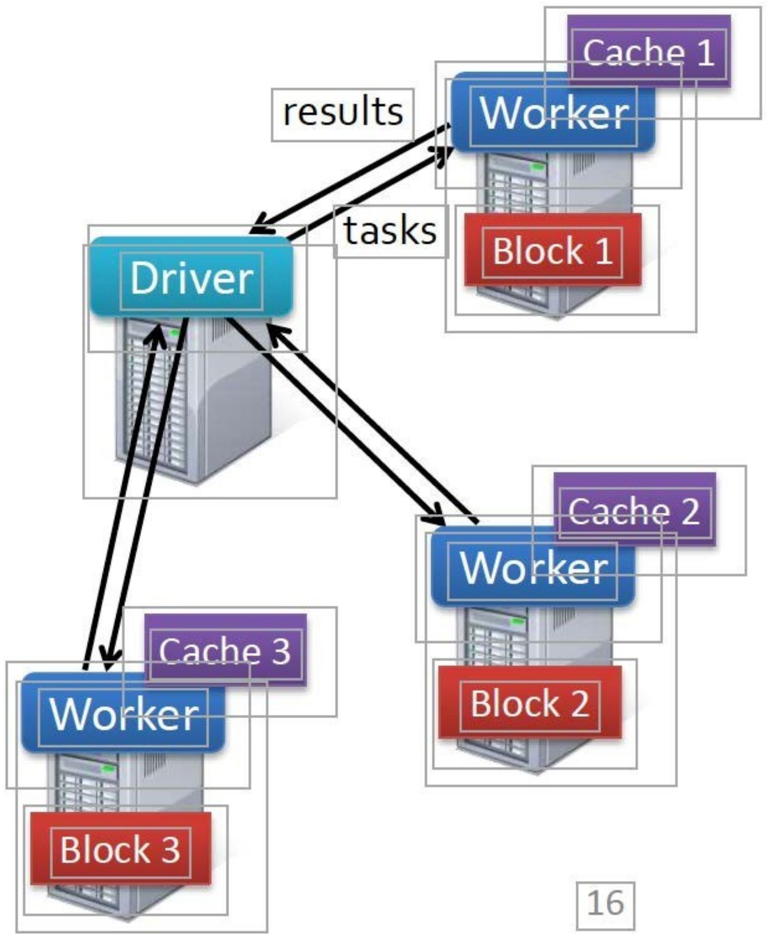


Spark vs Hadoop

Distributed Memory Processing

- Just like MapReduce (Hadoop), Spark has a central controller, called Driver (App Master) which distributes tasks to Workers;

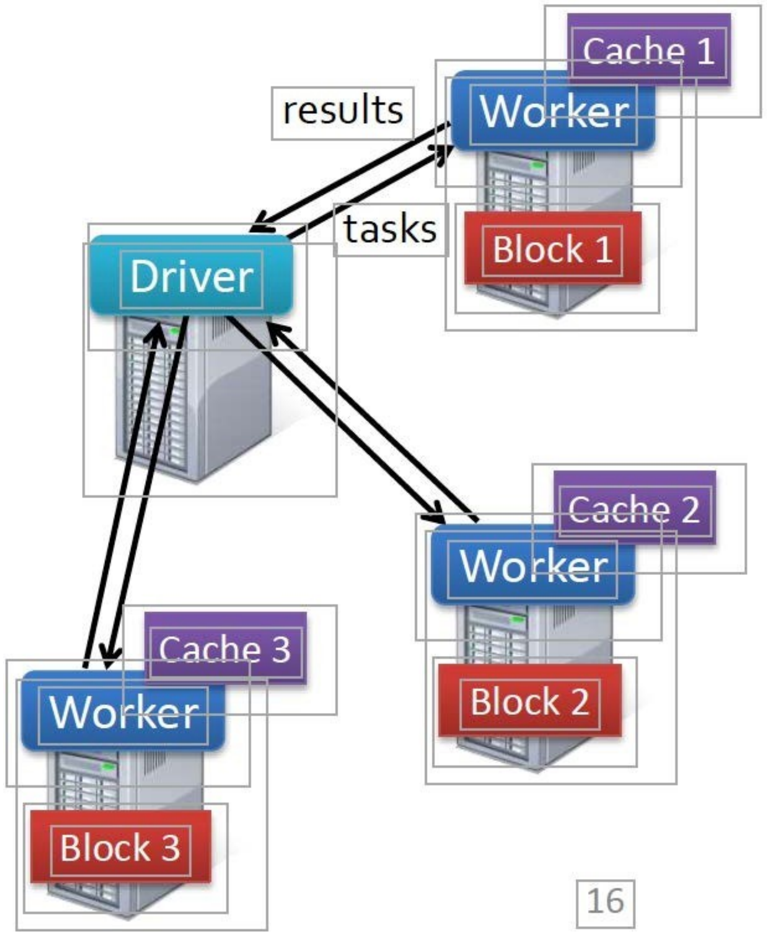
- Data (RDD) is split among memories of many workers.



Spark vs Hadoop

Distributed Memory Processing

- Data is originally sourced from the disk (HDFS, S3, regular File System);
- During processing data is shared only between memories (if possible);
- Driver, if it detects that a worker is slacking or not working at all could make the worker redo its work or could move processing to another worker.
- Spark is fault tolerant just like Hadoop;

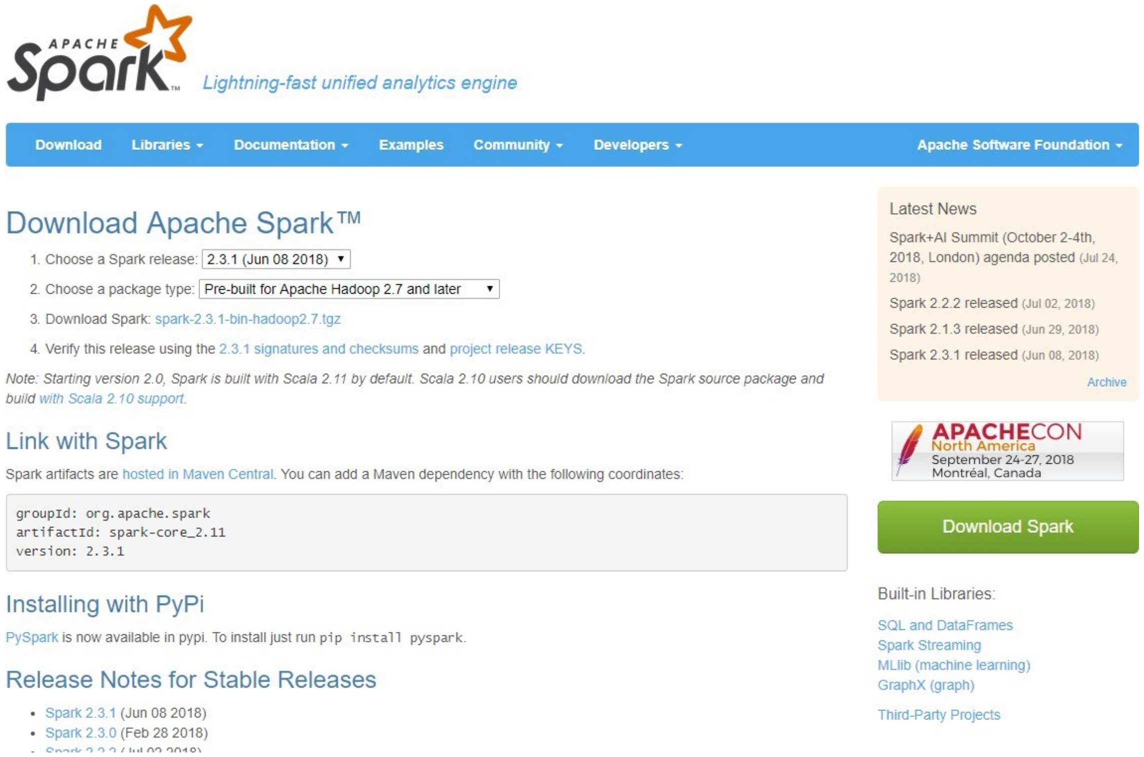


Installation of Spark

Installation of Spark

- We can go to http://spark.apache.org/downloads and fetch the newest and greatest release of Spark;

- There are a few options on what can be downloaded:
    - The source code;
    - Binaries;



Download Apache Spark™

1. Choose a Spark release: 2.3.1 (Jun 08 2018)
2. Choose a package type: Pre-built for Apache Hadoop 2.7 and later
3. Download Spark: spark-2.3.1-bin-hadoop2.7.tgz
4. Verify this release using the 2.3.1 signatures and checksums and project release KEYS.

Note: Starting version 2.0, Spark is built with Scala 2.11 by default. Scala 2.10 users should download the Spark source package and build with Scala 2.10 support.

Link with Spark

Spark artifacts are hosted in Maven Central. You can add a Maven dependency with the following coordinates:

```
groupId: org.apache.spark
artifactId: spark-core_2.11
version: 2.3.1
```

Installing with PyPi

PySpark is now available in pypi. To install just run `pip install pyspark`.

Release Notes for Stable Releases

- Spark 2.3.1 (Jun 08 2018)
- Spark 2.3.0 (Feb 28 2018)

Latest News

Spark+AI Summit (October 2-4th, 2018, London) agenda posted (Jul 24, 2018)

Spark 2.2.2 released (Jul 02, 2018)

Spark 2.1.3 released (Jun 29, 2018)

Spark 2.3.1 released (Jun 08, 2018)

Archive

APACHECON North America September 24-27, 2018 Montréal, Canada

Built-in Libraries:

SQL and DataFrames
Spark Streaming
MLlib (machine learning)
GraphX (graph)

Third-Party Projects

Installation of Spark

- To use the source code you need Maven;

- Working with Cloudera's VM-s we will do the following:
  - Make sure we have JDK 7+ installed;
  - Make sure we have hadoop-client package installed;

Installation of Spark

- For example, you could do:

  $ sudo yum install hadoop-client

- yum will either install the package or tell you it is already there;
- Make sure all HDFS and YARN services are installed and running;
- Stop them and start them all;

Installation of Spark

- Use yum to run installation of Spark, by typing all on one line:

$ sudo yum install spark-core spark-master spark-worker spark-history-server spark-python

- Stay positive. Answer yes to all the questions;

- For Linux versions other than CentOS, please consult Cloudera documentation;

Installation of Spark

- Set your SPARK_HOME env. variable to /usr/lib/spark and add $SPARK_HOME/bin to your PATH environmental variable. …. And you are all set now!

- yum made sure that you have Spark version which matches your Hadoop (Yarn) version;

- Without any further changes we will run Spark locally on this single machine;

Installation of Spark

- Spark comes with interactive shells that enable ad hoc data analysis.
- Unlike most other shells, which let you manipulate data using the disk and memory on a single machine, Spark's shells allow interaction with data distributed on disks or in memory across many machines.
- Spark can load data into memory on the worker nodes, and distributed computations, even ones that process large volumes of data across many machines, can run in a few seconds;
- This makes iterative, ad hoc, and exploratory analysis commonly done in shells a good fit for Spark;

Installation of Spark

- Spark provides both (and only) Python and Scala shells that have been augmented to support access to a cluster of machines.
- Python shell opens with pyspark command;
- Scala shell is very similar and opens with spark-shell command;
- You can program Spark from R using package SparkR;

Northeastern University

Installation of Spark

Pyspark

```
[cloudera@localhost conf]$ pyspark
Python 2.8.1 (r231:90122, Aug 23 2025, 23:20:16)
[GCC 4.4.7 20201212 (Red Hat 7.4.1-11)] on linux2
Type "help", "copyright", "credits" or "license" for more information. SLF4J: Class path contains multiple
SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/lib/zookeeper/lib/slf4j-log4j12-
1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/lib/flume-ng/lib/slf4j-log4j12-
1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory] Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.5.0-cdh5.5.1
      /_/

Using Python version 2.6.6 (r266:84292, Jul 23 2015 15:22:56)
SparkContext available as sc, HiveContext available as sqlContext.
>>>
```

Installation of Spark

- The output is long and annoying. It would have been even longer had we not created log4j.properties file in the directory $SPARK_HOME/conf;

- You create that file by copying provided file log4j.properties.template and by changing line:

log4j.rootCategory=INFO, console

to:

log4j.rootCategory=ERROR, console

- That lowered the logging level so that we see only the ERROR messages, and above;

Installation of Spark

- Another option is WARN, which is more verbose that ERROR but less than INFO;
- Before we proceed, let’s see what kind of files and how many lines we have in HDFS

```
hadoop fs –ls ulysis
-rw-r--r-- 1 cloudera 5258688 2015-04-01 14:32 input/4300.txt
```

```
hadoop fs -cat ulysis/4300.txt | wc
```

```
33056 267980 1573079
```

Load data into Spark

Load data into Spark

Load Data (RDD) from HDFS

- In Spark, we express our computation through operations on distributed collections that are automatically parallelized across the cluster;
- These collections are called resilient distributed datasets, or RDDs;
- When we load some data, i.e. a file into a shell variable, we are creating an RDD, like:

```
>>> lines = sc.textFile("ulysis/4300.txt")
>>> lines.count()
33056
```

Load data into Spark

Load Data (RDD) from HDFS

- What we’ve just done is create and populate an RDD named lines using a mysterious object “***sc***” and its method ***textFile***();
- We populated that RDD with data in HDFS file ***ulysis/4300.txt***;
- “***sc***” stands for an implicit SparkContext;
- SparkContext allows us to communicate with the execution environment;
- It appears that RDD-s are also (Object Oriented) objects and have methods, such as ***count()*** which gave use the exact number of lines in file ***4300.txt;***

Load data into Spark

Load Data (RDD) from Local File

We could load the same data from the local operating system;

We happen to have the ***4300.txt*** file in /***home/cloudera*** directory and could do the following:

```
>>> blines = sc.textFile("file:///home/cloudera/4300.txt")
>>> blines.count()

33056

>>> blines.first()
```

Load data into Spark

Load Data (RDD) from Local File

- What we did above was create an RDD named ***blines*** and populate that RDD with data from the local file /***home/cloudera/4300.txt***;
- We also see in action another method of RDD-s, ***first***(), which tells us that the first line in RDD ***blines*** is some uninterested collection of characters (***u***'');
- Note that we are not terminating our commands with a semi-colon ***(";")*** or anything else aside from the carriage return;
- Savings on typing all those semi-colons is one of the greatest contributions of Python to the computer science;
- By the way, our commands are in ***Python***;

Load data into Spark

Load Data (RDD) from the Cloud (AWS S3 Bucket)

- A file was already uploaded (the same 4300.txt file) to the Amazon's AWS S3 bucket called class2025;

- On my Linux box (Cloudera VM) I created two new environmental variables in file .bash_profile:

  AWS_ACCESS_KEY_ID=AMYSECRETKEYIDISHERE and

  AWS_SECRET_ACCESS_KEY=mysecretaccesskeyisheretoo

- The source file:

source .bash_profile

- Then, after reopening Python Spark shell, issued this command:

```
>>> s3lines = sc.textFile("s3n:// class2025 /4300.txt")
>>> s3lines.count()
33056
```

Load data into Spark

Load Data (RDD) from the Cloud (AWS S3 Bucket)

- We did not lose a single line of text while brining the text down from the Cloud:

```
>>> Heaven = s3lines.filter(lambda line: "Heaven" in line)
>>> Heaven.count()
2

>>> heaven = s3lines.filter(lambda line: "heaven" in line)
>>> heaven.count()
50
```

Load data into Spark

Load Data (RDD) from the Cloud (AWS S3 Bucket)

• We also see in action another method of RDD-s, ***filter***(), which apparently let us inquire how many times is heaven mentioned in the ***Ulysis***;

- <u>Heaven</u> is mentioned 52 time, i.e. some 0.15% of the time (> Bible);
- If you do not have an AWS account, do filtering examples on local files!

Load data into Spark

Python lambda Syntax

- **Python** supports the creation of anonymous inline functions (i.e. functions that are not bound to a name) at runtime, using a construct called "lambda“;

- The following code shows the difference between a normal function definition ("f") and a lambda function ("g"):

```
def f(x): return x**2
 print f(8)
64

g = lambda
 x[1]:x**2
 print g(8)
64
```

Load data into Spark

Python lambda Syntax

- As you can see, ***f()*** and ***g()*** do exactly the same thing;
- The ***lambda*** definition does not include a "***return***" statement;
- The last expression is returned;
- You can put a ***lambda*** definition anywhere a function is expected, and you don't have to assign it to a variable at all;

Load data into Spark

filter() Method

- Method ***filter()*** takes a function returning True or False and applies it to a sequence (list) and returns only those members of the sequence for which the function returned True;

- So, in the Python code:

heavens = s3lines.filter(lambda line: "Heaven" in line)

- Method ***filter()*** acts on the collection ***s3lines***, and passes every element of that collection as the variable line as the argument to the anonymous function created using lambda construct;

Load data into Spark

filter() Method

- That anonymous function uses a simple regular expression to test whether string "***Heaven***" exists in variable line;

- If the regular expression returns ***True*** for a particular line, an element of collection ***s3lines***, the anonymous function will return ***True*** and for that particular line, ***filter()*** will return/add variable line building up a new collection called heavens;

- You can accomplish the same in Java 1.7 and older with several lines of code;

- In Java 1.8 you have similarly efficient lambda constructs;

Load data into Spark

Passing Function to Spark in Python

- We want to convince ourselves that rather than using lambda constructs we could define functions and then pass their names to Spark;

- For example:

```
def hasLife(line):
return "life" in line     # At the beginning hit a tab
lines = sc.textFile("file:///home/cloudera/4300.txt")
lifeLines = lines.filter(hasLife)
lifeLines.count()
203

print lifeLines.first()
whom Mulligan was one, and Arius, warring his life long upon the'
```

Load data into Spark

- Passing Function to Spark in Python

- Function ***hasLife()*** returns ***True*** if the line of text in its argument contains string "***life***". We successfully passed that function's name to method ***filter***();

- Above, we have seen a few more crucial features of Python;

- Python accepted the second line of function definition only when we properly indented return "***life***"… statement;

- Also, to terminate function definition, we had to hit Carriage Return (***Enter***) twice;

- Most importantly, there are no semicolons in sight.

Load data into Spark

Spark Sources and Destinations of Data

- Spark supports a wide range of input and output sources, partly because it builds on the ecosystem made available by Hadoop;

- In particular, Spark can access data through the InputFormat and OutputFormat interfaces used by Hadoop MapReduce, which are available for many common file formats and storage systems (e.g., S3, HDFS, Cassandra, HBase, etc.);

- For data stored in a local or distributed file system, such as NFS, HDFS, or Amazon S3, Spark can access a variety of file formats including Text, JSON, SequenceFiles, and Google's Protocol Buffers;

Load data into Spark

Spark Sources and Destinations of Data

- The Spark SQL module, provides an efficient API for structured data sources, including JSON and Apache Hive;
- Spark could also use third-party libraries for connecting to Cassandra, HBase, Elasticsearch, and JDBC databases;
- We will analyze some of the above techniques a bit later;

Load data into Spark

File Formats

- **Spark** makes it very simple to load and save data in a large number of file formats;
- **Spark** transparently handles compressed formats based on the file extension;
- We can use both **Hadoop's** new and old file **APIs** for keyed (or paired) data;
- We can use those only with key/value data;

Format Name	Structured	Comments
Text File	No	Plain old text files. Records assumed to be one per line.
JSON	Semi	Common text-based format, semi-structured; most libraries require one record per line.
CSV	YES	Very common text-based format, often used with spreadsheet applications.
SequenceFile	YES	A common Hadoop file format used for key/value data
Protocol buffer	YES	A fast, space-efficient multi-language format
Object file	YES	Useful for saving data from a Spark job to be consumed by shared code. Breaks if you change your classes, as it relies on Java Serialization.

Load data into Spark

Standalone Applications

- Spark can be linked into standalone applications in either Java, Scala, or Python;
- The main difference from using it in the shell is that you need to initialize your own SparkContext. After that, the API is the same;
- The process of linking to Spark varies by language;
- In Java and Scala, you give your application a Maven dependency on the spark-core artifact;
- Maven is a popular package management tool for Java-based languages that lets you link to libraries in public repositories;

Load data into Spark

Standalone Applications

- You can use Maven itself to build your project, or use other tools that can talk to the Maven repositories, including Scala's sbt tool or Gradle.;
- Eclipse also allows you to directly add a Maven dependency to a project;
- In Python, you simply write applications as Python scripts, but you must run them using the bin/spark-submit script included in Spark;
- The spark-submit script includes the Spark Python dependencies;
- We simply run your script with the

$SPARK_HOME/bin/spark-submit   your_script.py

Load data into Spark

Standalone Application in Python

- To create an application (Python script) we need to import some Python classes and create SparkContext object;

- The rest of the application is coded as if you are writing code in PySpark shell from pyspark import SparkConf, SparkContext

```
conf = SparkConf().setMaster("local").setAppName("MyApp")
sc = SparkContext(conf = conf)
lines = sc.textFile("/ulysis/4300.txt")
lifeLines = lines.filter(lambda line: "life" in line)
print lifeLines.first()
```

Load data into Spark

Standalone Application in Python

- If we invoke the above with:

```
spark-submit my_script.py
```

- The output will be:

```
py4j.protocol.Py4JJavaError: An error occurred while calling o25.partitions.
org.apache.hadoop.mapred.InvalidInputException: Input path
does not exist: hdfs://localhost:8020/ulysis/4300.txt
```

Load data into Spark

Python Applications use full HDFS path

- We have to provide the complete HDFS path to our files and the filter() line of the script should read:

lines = sc.textFile("/user/cloudera/ulysis/4300.txt")

- In HDFS, every user has a home directory, /user/cloudera in our case.

- Modified script will produce the expected output:

spark-submit your_script.py

whom Mulligan was one, and Arius, warring his life long upon the

Load data into Spark

Python Applications use full HDFS path

- If we want to read 4300.txt file from the local Linux directory /home/cloudera, we should change the above line to:

```
lines = sc.textFile("file:///home/cloudera/4300.txt")
```

Load data into Spark

To initialize SparkContext in Java we would write

```
import org.apache.spark.SparkConf;

import org.apache.spark.api.java.JavaSparkContext;

SparkConf conf = new SparkConf().setMaster("local").setAppName("MyApp");

JavaSparkContext sc = new JavaSparkContext(conf);
```

Load data into Spark

To initialize SparkContext in Python we would do

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("MyApp")

sc = SparkContext(conf = conf)
```

Load data into Spark

- Only one SparkContext may be active per JVM;
- You must stop() the active SparkContext before creating a new one;
- To **shutdown** SparkContext call **sc.stop()**

or

- **Exist** the application with **System.exit()** or **sys.exit()**

Load data into Spark

• Initializing SparkContext

- Previous examples show the minimal way to initialize a SparkContext, where we pass two parameters:
  - *Cluster_URL*, namely local in these examples, which tells Spark how to connect to a cluster;
  - *Local* is a special value that runs Spark on one thread on the local machine, without connecting to a cluster;

- An application name, namely MyApp in these examples. This will identify your application on the cluster manager's UI if you connect to a cluster.

- Additional parameters exist for configuring how your application executes or add code to be shipped to the cluster;

- SparkContext is the main entry point for Spark functionality.

Load data into Spark

SparkContext API

- ***Java*** class implementing SparkContext object is described here:

https://spark.apache.org/docs/1.6.0/api/java/org/apache/spark/SparkCon text.html

Load data into Spark

SparkContext API

- ***Python*** class implementing SparkContext object is described here:

https://spark.apache.org/docs/1.6.0/api/python/pyspark.html#pyspark.Sp arkContext

Load data into Spark

SparkContext API

- ***Scala*** class implementing SparkContext object is described here:

https://spark.apache.org/docs/1.6.0/api/scala/index.html#org.apache.spark.SparkContext

Load data into Spark

SparkContext API

- **R** package for working with Spark could be found at:

https://spark.apache.org/docs/1.6.0/api/R/index.html

Load data into Spark

- While the **functionality** is **almost identical** in all supported languages, there does not exist a one-to-one mapping between methods of respective classes in all four languages;

- Most methods one could find in Scala class **SparkContext** exist in Java class **SparkContext**;

- Python's **SparkContext** has fewer methods. It appears that some are moved to some other Python classes, like **RDD**

Northeastern University

Building Spark Application using Maven

(For Advanced Data Engineers/Developers)

Building Spark Application using Maven

- To build Spark Applications you need Maven for Java and sbt tool for Scala apps;
- You download Maven's binary tar.gz file from http://maven.apache.org/download.cgi;
- You untar or unzip the archive;
- On Linux you usually copy the untared folder to /usr/local;

Building Spark Application using Maven

- Then you add to your .bash_profile file

```
M2_HOME=/usr/local/apache-maven-3.3.9 export M2_HOME
PATH=$M2_HOME/bin:$PATH
export PATH
```

- Maven’s executable is called mvn, so you ask for it like:

```
which mvn
/usr/local/apache-maven-3.3.9/bin/mvn
```

Building Spark Application using Maven

- Maven does what ***ant*** does, or what ***make*** does in the world of C/C++;
- You use **Maven** to compile (build) your projects;
- **Maven** also maintains a public repository of all kinds of packages and their versions and if properly instructed it will fetch for your build process any and all software packages (dependencies) you need;
- **Maven** can do many other manipulations of your files/projects/packages and make little children, if you want it to;

Building Spark Application using Maven

- Maven build file **pom.xml**;
- Instructions for Maven, what to do and where to get what it needs, are written in a file called **pom.xml**;
- Initially, you copy and past content of that file;
- At one point you start writing your own;

Building Spark Application using Maven

This is a file that will help us compile WordCount.java

```
<project> <groupId>edu.neu.examples</groupId>
 <artifactId>spark-examples</artifactId>
  <modelVersion>5.1.2</modelVersion>
  <name>example</name>
  <packaging>jar</packaging> <version>1.3.4</version> <dependencies>
 <dependency> <!-- Spark dependency -->
 <groupId>org.apache.spark</groupId>
   <artifactId>spark-core_4.11</artifactId> <version>3.1.1</version>
   <scope>provided</scope>
 </dependency>
  </dependencies>
  <properties><java.version>1.6.6</java.version> </properties>
  <build>
 <pluginManagement>
 <plugins><plugin>
<groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-compiler-plugin</artifactId>
 <version>3.1</version> <configuration>
 <source>${java.version}</source> <target>${java.version}</target>

 </configuration> </plugin>
 </plugins>
 </pluginManagement>
 </build>
 </project>
```

Building Spark Application using Maven

Maven's pom.xml file

- Project Object Model (pom):
  - Describes a project
  - Name and Version
  - Artifact Type
  - Source Code Locations
  - Dependencies
  - Plugins
  - Profiles (Alternate build configurations)

Building Spark Application using Maven

- To build a project we need to create a directory hierarchy for our code artifacts. Below is a possible structure of that hierarchy;

- Directories /src and /project are at the same level. Initially, /project is empty.

```
/mini-examples
        build.sbt pom.xml /src
                /main
                        /java
                                /examples
                                        WordCount.java
/project
```

Building Spark Application using Maven

```
$ mkdir -p mini-examples/src/main/java/edu/hu/examples
$ mkdir -p mini-examples/src/main/scala/edu/hu/examples
```

- **Run Maven** from the directory where pom.xml resides, i.e. /mini-examples, like this:

```
$ mvn clean && mvn compile && mvn package
```

- Build process downloaded a bunch of “dependencies”, compiled class ***WordCount.java*** and created directory target

Building Spark Application using Maven

Structure of Maven’s Project Maven project structure

- target: Default work directory;
- src: All project source files go in this directory;
- src/main: All sources that go into primary artifact;
- src/test: All sources contributing to testing project;
- src/main/java: All java source files;
- src/main/webapp: All web source files;
- src/main/resources: All non compiled source files;
- src/test/java: All java test source files;
- src/test/resources: All non compiled test source files ;

Building Spark Application using Maven

Generated Artifacts

- Generated directory target contains directory classes with the compiled Java classes and a jar file ***spark-example-0.0.1.jar***;
- This ***jar*** contains compiled classes coming from the original ***src*** directory;
- This ***jar*** file, in a cluster with more than one machine, will be shipped to various worker nodes;
- The name of the ***jar*** file is a concatenation of pom.xml elements <***artifactId***> and <***version***>
- Directory maven-archiver contains file ***pom.properties*** which lists values of relevant elements from ***pom.xml*** file: version, ***groupId*** and ***artifactId***;

Building Spark Application using Maven

Generated Artifacts

- To run compiled code, in the directory where pom.xml file resides, we issue the following command, all on one line:

$ spark-submit –-class edu.hu.examples.WordCount ./target/spark-example-0.0.1.jar ulysis/4300.txt wordcounts

- In the above, string ***ulysis/4300.txt*** is the name of the ***HDFS*** input file (***WordCount.java inputFile*** variable) and ***wordcounts*** is the name of the ***WordCount.java*** outputFile ***variable***, i.e. ***HDFS*** output directory.

Building Spark Application using Maven

WordCount.java Spark's Heavenly Output

- Once the run is finished, in HDFS directory /user/cloudera/wordcounts we will find files named part-00000 and part-00001;

- If we open those files, we will see that they contain something that looks like words and counts:

…………………………

Building Spark Application using Maven

WordCount.java Spark's Heavenly Output

(supplied.,1)
(Ah!,14)
(reunion,2)
(bone,5)
(Justifiable,1)
(Hats,1)
(Apart.,1)
(blandly,1)
(fiction.,1)
(Friend,2)
(hem,2)
(stinks,3)
(boats?,1)
(flutiest,1)
(Lost.,1)
(fuller,2)
(jade,1)

- If we search for the words heaven or Heaven, we will find numbers like

  (heavens,5) (HeavenS_,1).

- There were some 50+ 2 = 52 heavens.
- They are there, except that neither of our searches was very sophisticated;
- If we do

egrep –i heaven part-00000

- We will get 56 of them;
- What is even better (more) than the result with simple filter() method;

(heavens,5)
(heaven._,1)
(Heavenly,1)
(heavens,,1)
(heavenbeast,,1)
(heavenborn,1)
(heavenly,2)
(heaven!,1)
(heavenly.,1)
(heavengrot,,1)
(heaven,,8)
(Heavens_,1)
(heavenman.,1)
(heaven.,8)
(heaventree,1)
(heaven,17)
(heaventree,,1) (heaven:,1)
(heaven's,1)
(heavenworld,1)
(heavenward.)_,1)

Building Spark Application using Maven

Linux tee Command

Unix-like Systems

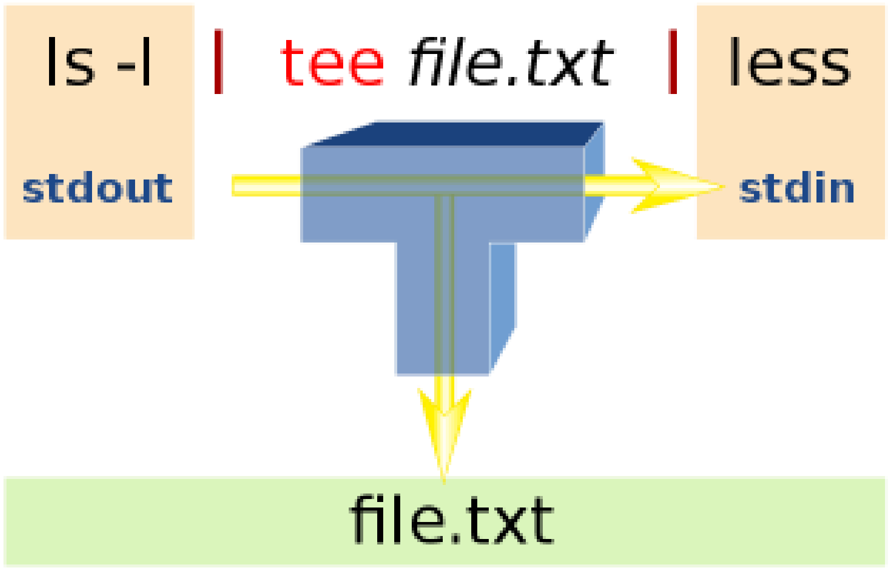
tee [-a] [-i] [File ...] Arguments:

File One or more files that will receive the "tee-d" output. Flags:

- *a Appends the output to the end of File instead of writing over it.*
- *i Ignores interrupts.*

The command returns the following exit values (exit status):

- 0 The standard input was successfully copied to all output files;
- 0 An error occurred;



Building Spark Application using Maven

WordCount in Python

• You can do WordCounting in Python as well:

```
file = sc.textFile("/user/cloudera/ulysis/4300.txt")
counts = lines.flatMap(lambda line: line.split(" ")) \
 .map(lambda word: (word, 1)).reduceByKey(lambda a, b: a +b)
>>> counts.saveAsTextFile("pycounts")
```

• In HDFS directory ***pycounts*** you will find file ***part-00000*** and ***part-00001***

Building Spark Application using Maven

Programming with RDDs

- An **RDD** in **Spark** is simply an immutable distributed collection of objects;
- Each **RDD** is split into multiple partitions, which may be computed on different nodes of the cluster;
- **RDDs** can contain any type of **Python**, **Java**, or **Scala objects**, including user defined classes;
- Users create **RDDs** in two ways: by loading an external dataset, or by distributing a collection of objects (e.g., a list or set) in their driver program;
- We have already seen loading a text file as an **RDD** of strings using **SparkContext.textFile**();

Building Spark Application using Maven

Programming with RDDs

- 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;
- It can be **use** to create a new **RDD** holding just the strings that contain the word Heaven;
- **Transformations** always return an **RDD**;
- **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**.

Building Spark Application using Maven

Lazy Compute

- **Spark** computes **RDDs** only in a lazy fashion—that is, the first time they are used in an action;

- If **Spark** were to load and store all the lines in the file as soon as we define an **RDD**, it would waste a lot of storage space, given that we might filter out many lines. Instead, once Spark sees the whole chain of transformations, it can compute just the data needed for its result. In fact, for the **first()** action, Spark scans the file only until it finds the first matching line; it doesn't even read the whole file;

- Finally, **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();**

Building Spark Application using Maven

Lazy Compute

- We can ask **Spark** to **persist** our **data** in a **number** of different **places**;
- After computing it the first time, **Spark** will store the **RDD** contents in memory (partitioned across the machines in your cluster), and reuse them in future actions;
- **Persisting RDDs** on disk instead of memory is also possible;
- In practice, you will often use persist() to load a subset of your data into memory and query it repeatedly.

```
>>> pythonLines.persist
>>> pythonLines.count()
>>> pythonLines.first()
```

Building Spark Application using Maven

Creating RDDs

- We already know how to create **RDDs** by loading data from external files;
- Another way to create **RDDs** is to take an existing collection in your program and pass it to **SparkContext's parallelize**() method;
- Like **parallelize**() method in **Python**

lines = sc.parallelize(["Heaven", "Earth"])

parallelize() method in Scala

val lines = sc.parallelize(List ("Heaven", "Earth"))

parallelize() method in Java

JavaRDD<String> lines = sc.parallelize(Arrays.asList ("Heaven", "Earth"));

Building Spark Application using Maven

Transformations

- Suppose that we have a logfile, log.txt, with a number of messages, and we want to select only the error messages;
- We can use the **filter**() transformation seen before;
- **filter**() transformation in **Python**;

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```

filter() transformation in Scala

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line =>
line.contains("error"))
```

filter() transformation in Java

```
JavaRDD<String> inputRDD = sc.textFile("log.txt");
JavaRDD<String> errorsRDD = inputRDD.filter(
new Function<String, Boolean>() {
 public Boolean call(String x) { return x.contains("error"); }
 } });
```

- **filter**() operation does not mutate the existing **inputRDD**. Instead, it returns a pointer to an entirely new **RDD**. **inputRDD** can still be reused later in the program

Building Spark Application using Maven

union() Transformation

- If we need to print the number of lines that contained either error or warning, we could use **union()** function, which is identical in all three languages;

- Text that follows is in **Python**:

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 operate on any number of input **RDDs**;

Building Spark Application using Maven

Actions

- At some point, we'll want to actually do something with our dataset;
- Actions are the second type of RDD operation;
- They are the operations that return a final value to the driver program or write data to an external storage system;
- Actions force the evaluation of the transformations required for the RDD they were called on, since they need to actually produce output;

Building Spark Application using Maven

Actions

- Continuing the log example from the previous section, we might want to print out some information about the **badLinesRDD**;

- To do that, we'll use two actions, **count()**, which returns the count as a number, and **take(),** which collects a number of elements from the **RDD**,.

Building Spark Application using Maven

Actions

- In the following example, we will use method take() to retrieve a small number of elements (sample of 10) in the **RDD** at the driver program;
- We then iterate over them locally to print out information at the driver;
- **RDDs** also have a **collect()** function to retrieve the entire **RDD**;
- This can be useful if your program **filters RDDs** down to a very small size and you'd like to deal with it locally;
- The entire dataset must fit in memory on a single machine to use **collect()** on it;

Building Spark Application using Maven

Actions, count(), take()

- ***Python*** error count and sample display using actions:

print "Input had " + badLinesRDD.count() + " concerning lines" print "Here are 10 examples:"

for line in badLinesRDD.take(10):

print line

Building Spark Application using Maven

Actions, count(), take()

- ***Java*** error count and sample display using actions

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

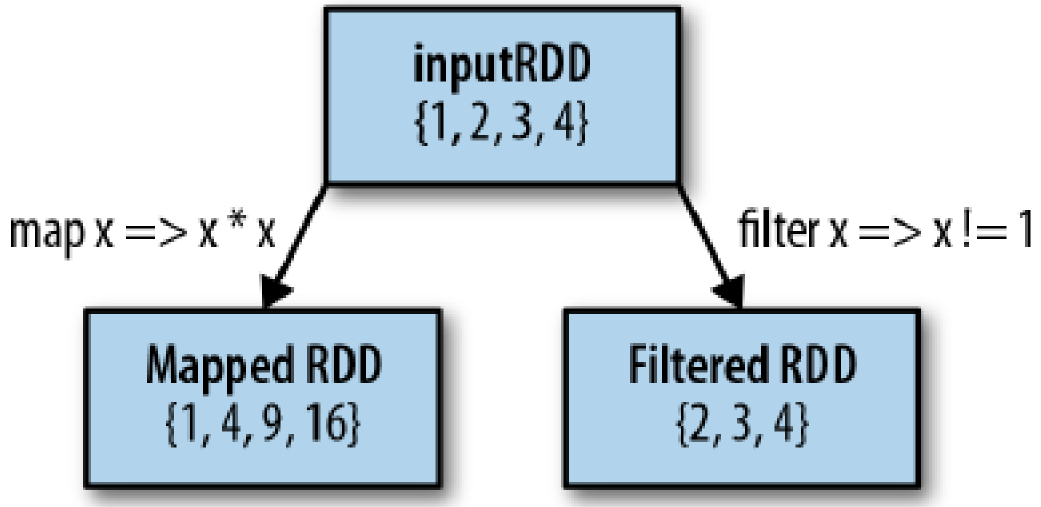
- If ***RDDs*** can't be ***collect()***ed to the driver because they are too large, it is common to write data out to a distributed storage system such as ***HDFS*** or Amazon ***S3***;

- 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;

Building Spark Application using Maven

Common Transformations and Actions

The two most **common transformations** we use are **map()** and **filter():**



Building Spark Application using Maven

Common Transformations and Actions

- 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;

Building Spark Application using Maven

map() Examples

- *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)
```

- *Java squaring the values in an RDD*

```
JavaRDD<Integer> rdd = sc.parallelize(
 Arrays.asList(1, 2, 3, 4)); JavaRDD<Integer> result = rdd.map(
 new Function<Integer, Integer>() {
 public Integer call(Integer x) { return x*x; } });
System.out.println(StringUtils.join(result.collect(), ","));
```

Building Spark Application using Maven

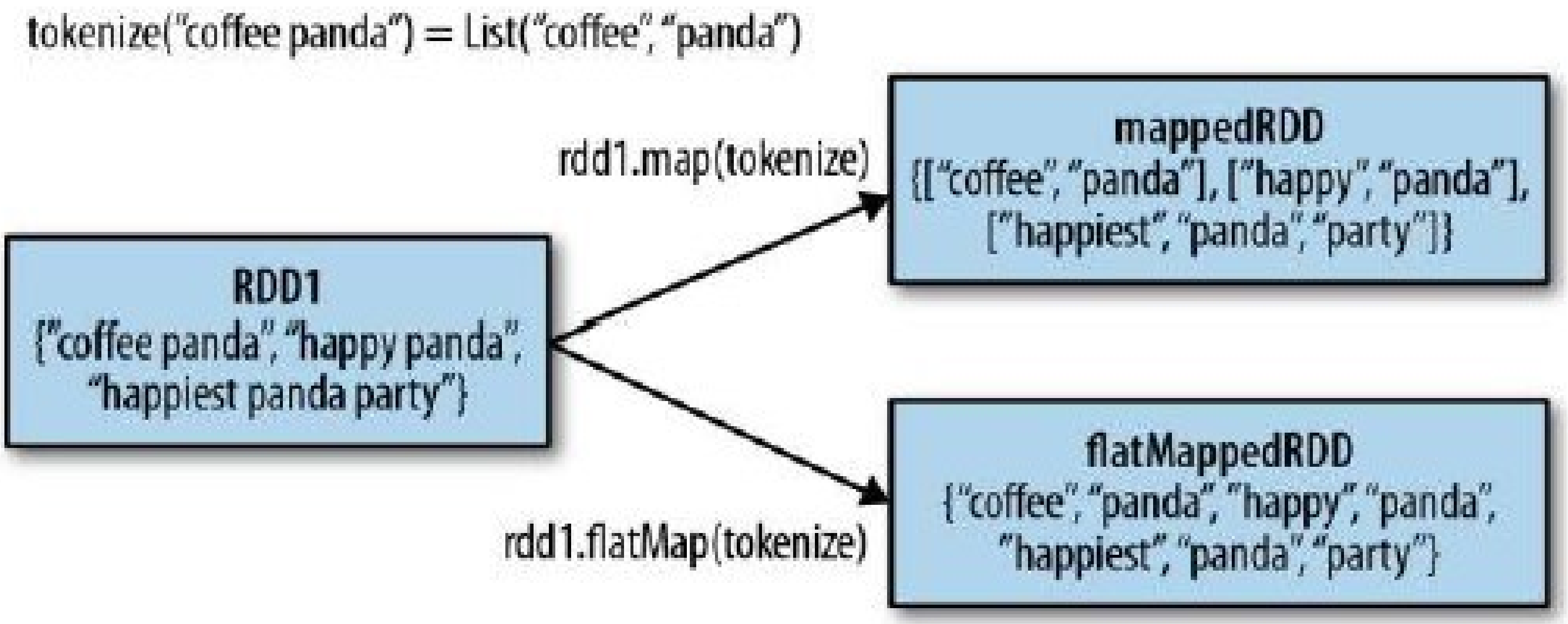
flatMap()

- 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;

Building Spark Application using Maven

flatMap()

The difference between map() and flatMap()


tokenize("coffee panda") = List("coffee", "panda")
rdd1.map(tokenize)
mappedRDD
{["coffee", "panda"], ["happy", "panda"], ["happiest", "panda", "party"]}
RDD1
{"coffee panda", "happy panda", "happiest panda party"}
rdd1.flatMap(tokenize)
flatMappedRDD
{"coffee", "panda", "happy", "panda", "happiest", "panda", "party"}

Building Spark Application using Maven

flatMap() Examples

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() in Java, splitting lines into multiple words

```
JavaRDD<String> lines = sc.parallelize(
Arrays.asList("hello world", "hi")); JavaRDD<String> words = lines.flatMap(
 new FlatMapFunction<String, String>() {
 public Iterable<String> call(String line) {
return Arrays.asList(line.split(" "));    }
});
words.first(); // returns "hello"
```

Building Spark Application using Maven

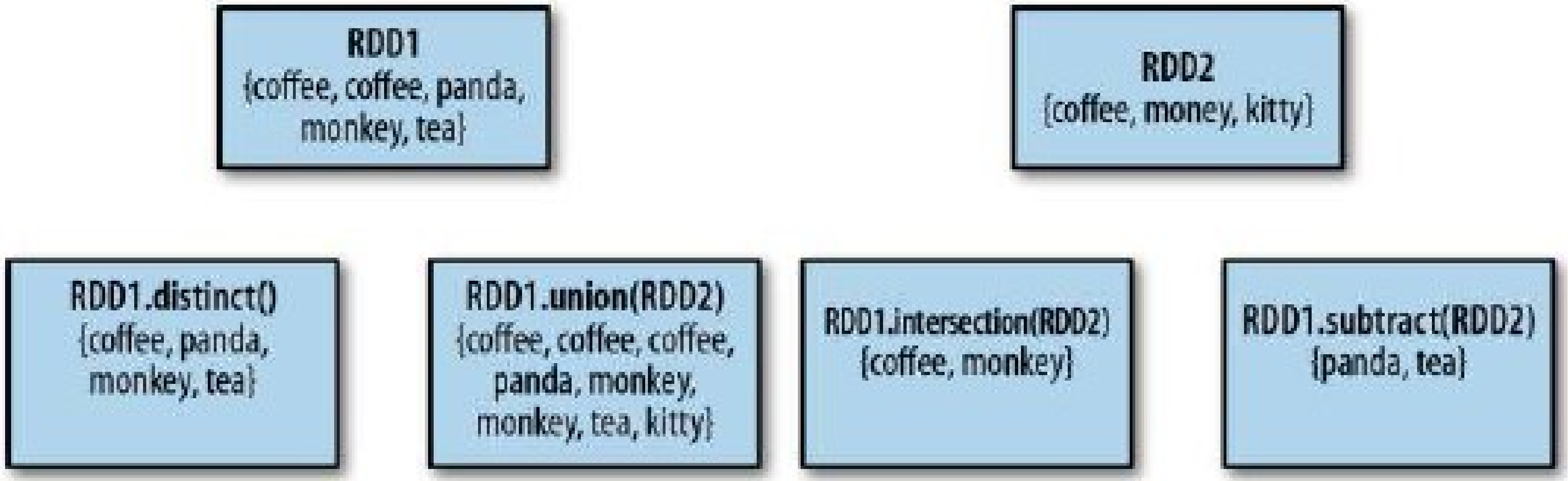
Pseudo set operations

- **RDDs** support many of the operations of mathematical sets, such as union and intersection, even when the **RDDs** themselves are not proper sets;
- **All set operations** require that the **RDDs** being operated on have elements of the same type;
- The **set property** most frequently missing from **RDDs** is the uniqueness of elements, as we often have duplicates;

Building Spark Application using Maven

Pseudo set operations

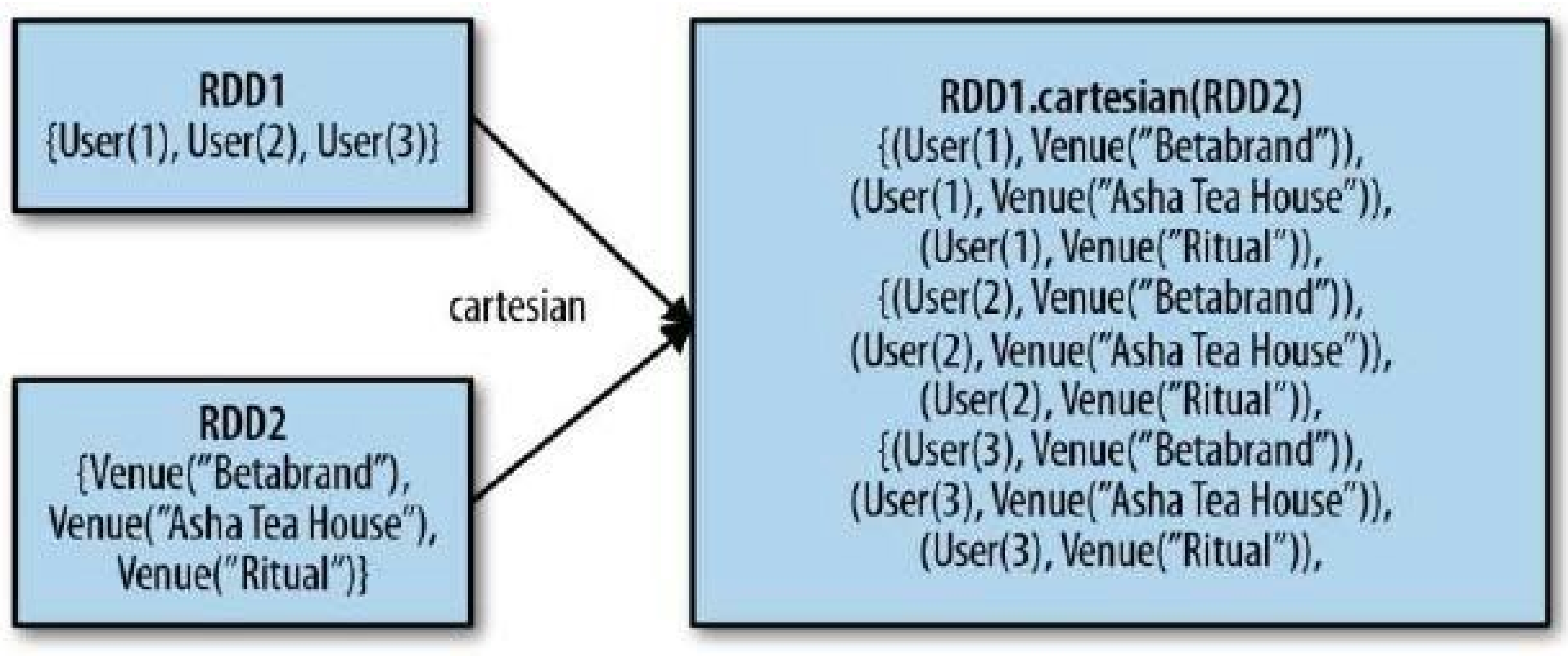
- Below, illustrated four set operations:
  - distinct();
  - union();
  - intersection();
  - subtract();



Building Spark Application using Maven

Cartesian Product between RDDs

- The cartesian(other) transformation returns all possible pairs of (a, b) where a is in the source RDD and b is in the other RDD.


RDD1
{User(1), User(2), User(3)}
RDD2
{Venue("Betabrand"), Venue("Asha Tea House"), Venue("Ritual")}
cartesian
RDD1.cartesian(RDD2)
{(User(1), Venue("Betabrand")),
(User(1), Venue("Asha Tea House")),
(User(1), Venue("Ritual")),
{(User(2), Venue("Betabrand")),
(User(2), Venue("Asha Tea House")),
(User(2), Venue("Ritual")),
{(User(3), Venue("Betabrand")),
(User(3), Venue("Asha Tea House")),
(User(3), Venue("Ritual")),

Building Spark Application using Maven

Actions, reduce()

- The most common action on basic **RDDs** you will likely use is **reduce(),** which takes a function that operates on two elements of the type in your **RDD** and returns a new element of the same type;

- A simple example of such a function is +, which we can use to **sum** our **RDD**;

- With **reduce(),** we can easily sum the elements of our **RDD**, count the number of elements, and perform other types of aggregations;

Northeastern University

Building Spark Application using Maven

Actions, reduce()

reduce() in Python

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

reduce() in Java

```
Integer sum = rdd.reduce(
new Function2<Integer, Integer, Integer>() {
public Integer call(Integer x, Integer y) { return x + y; }
 });
```

- ***reduce()*** requires that the return type of our result be the same type as that of the elements in the **RDD** we are operating over.

Building Spark Application using Maven

fold()

- Similar to **reduce()** is **fold(),** which also takes a function with the same signature as needed for **reduce(),** but in addition takes a "**zero valu**e" to be used for the initial call on each partition;

- The **zero value** you provide should be the identity element for your operation; that is, applying it multiple times with your function should not change the value (e.g., 0 for +, 1 for *, or an empty list for concatenation);

- **fold()** requires that the return type of our result be the same type as that of the elements in the **RDD** we are operating over;

- This works well for operations like **sum**, but sometimes we want to return a different type;

Building Spark Application using Maven

fold()

- For example, when computing a running average, we need to keep track of both the count so far and the number of elements, which requires us to return a pair;

- We could work around this by first using map() where we transform every element into the element and the number 1, which is the type we want to return, so that the reduce() function can work on pairs.

Building Spark Application using Maven

aggregate()

- The **aggregate()** function frees us from the constraint of having the return be the same type as the **RDD** we are working on;
- With **aggregate(),** like **fold(),** we supply an initial zero value of the type we want to return;
- We then supply a function to combine the elements from our **RDD** with the accumulator;
- Finally, we need to supply a second function to merge two accumulators, given that each node accumulates its own results locally;
- We can use **aggregate()** to compute the average of an **RDD**, avoiding a map() before the **fold();**

Building Spark Application using Maven

aggregate()

aggregate() in Python

sumCount = nums.aggregate((0, 0),

(lambda acc, value: (acc[0] + value, acc[1] + 1),

(lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1])))) return sumCount[0] / float(sumCount[1])

aggregate() in Scala

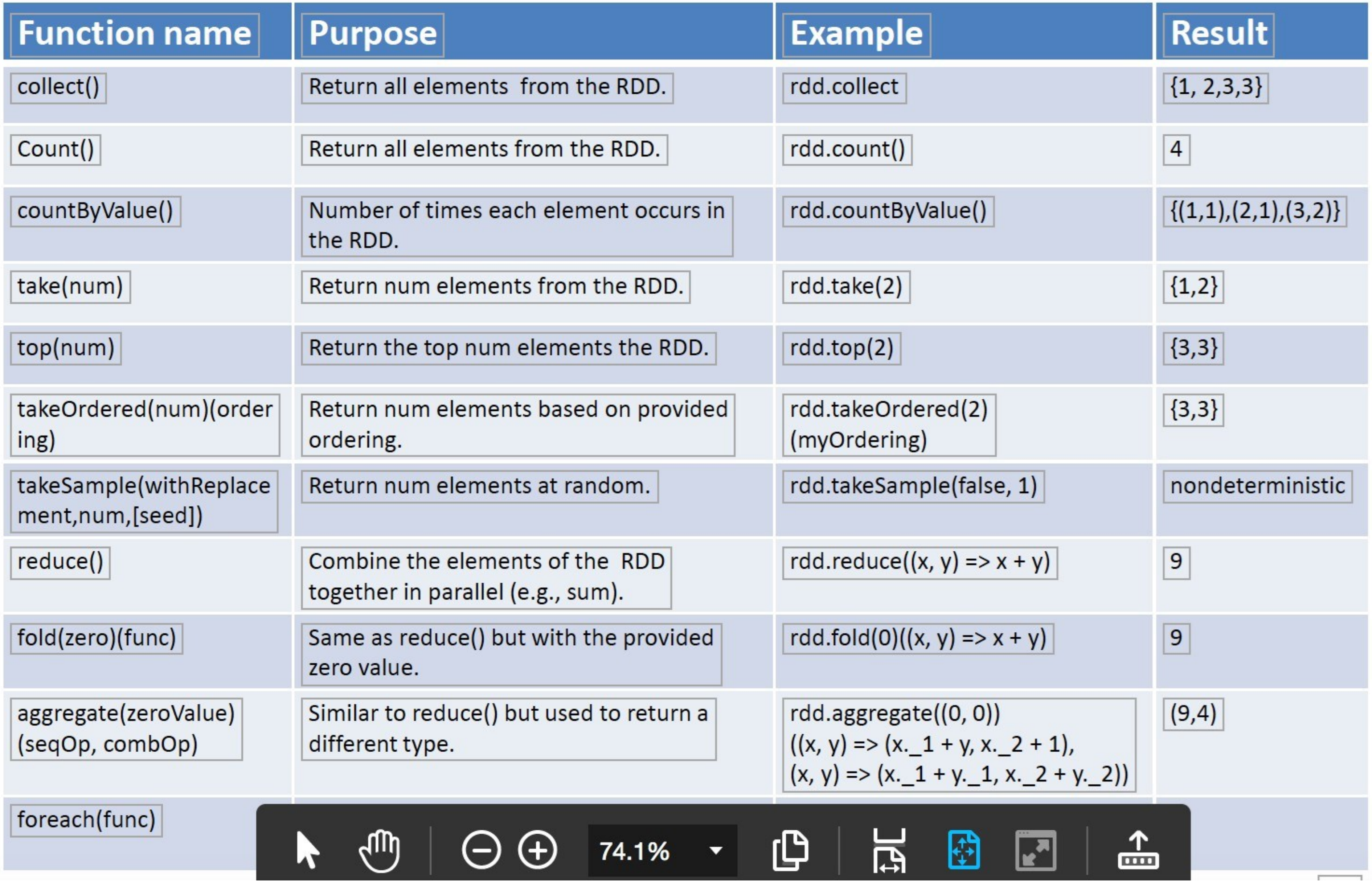
val result = input.aggregate((0, 0))(

(acc, value) => (acc._1 + value, acc._2 + 1),

(acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))

val avg = result._1 / result._2.toDouble

Northeastern University



Function name	Purpose	Example	Result
collect()	Return all elements from the RDD.	rdd.collect	{1, 2,3,3}
Count()	Return all elements from the RDD.	rdd.count()	4
countByValue()	Number of times each element occurs in the RDD.	rdd.countByValue()	{(1,1),(2,1),(3,2)}
take(num)	Return num elements from the RDD.	rdd.take(2)	{1,2}
top(num)	Return the top num elements the RDD.	rdd.top(2)	{3,3}
takeOrdered(num)(ordering)	Return num elements based on provided ordering.	rdd.takeOrdered(2)(myOrdering)	{3,3}
takeSample(withReplacement,num,[seed])	Return num elements at random.	rdd.takeSample(false, 1)	nondeterministic
reduce()	Combine the elements of the RDD together in parallel (e.g., sum).	rdd.reduce((x, y) => x + y)	9
fold(zero)(func)	Same as reduce() but with the provided zero value.	rdd.fold(0)((x, y) => x + y)	9
aggregate(zeroValue)(seqOp, combOp)	Similar to reduce() but used to return a different type.	rdd.aggregate((0, 0))((x, y) => (x._1 + y, x._2 + 1),(x, y) => (x._1 + y._1, x._2 + y._2))	(9,4)
foreach(func)			

Building Spark Application using Maven

Working with Key/Value Pairs

- While **Spark** tries to distinguish itself from **MapReduced**, most calculation in **Spark** rely on key/value pairs data types;
- Key/value **RDDs** are commonly used to perform aggregations;
- 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**);
- **Spark** introduces an advanced feature, partitioning, that lets users control the layout of pair **RDDs** across nodes;

Building Spark Application using Maven

Working with Key/Value Pairs

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

- Spark provides special operations on **RDDs** containing key/value pairs;

- These **RDDs** are called pair **RDDs**;

- 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;

- 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.

Building Spark Application using Maven

Creating Pair RDDs

- There are a number of ways to get pair **RDDs** in **Spark**;
- Many **import** formats will directly return pair **RDDs** for their key/value data;
- In other cases we have a **regular RDD** that we want to turn into a **pair RDD**;
- We can do this by running a **map()** function that returns key/value pairs;
- To illustrate, we show code that starts with an **RDD** of lines of text and keys the data by the first word in each line;
- The way to **build key-value RDDs** differs by language;

Building Spark Application using Maven

Creating Pair RDDs

- In **Python**, for the functions on keyed data to work we need to return an RDD composed of tuples;
- **Creating a pair RDD** using the first word as the key in Python:

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

- In **Scala**, for the functions on keyed data to be available, we also need to return tuples;
- An implicit conversion on **RDDs** of tuples exists to provide the additional key/value functions:

- **Creating a pair RDD** using the first word as the key in Scala:

val pairs = lines.map(x => (x.split(" ")(0), x))

Building Spark Application using Maven

Aggregations

- When **datasets** are **described** in terms of **key/value pairs**, it is common to want to aggregate statistics across all elements with the same key;
- We have looked at the **fold(), combine(),** and **reduce()** actions on basic **RDDs**, and similar **per-key** transformations exist on pair **RDDs**;
- **Spark** has a similar set of operations that combines values that have the same key;
- These operations return **RDDs** and thus are transformations rather than actions;
- **reduceByKey()** is quite similar to **reduce();**
- Both take a function and use it to combine value;

Building Spark Application using Maven

Aggregations

- **reduceByKey()** runs several parallel reduce operations, one for each key in the dataset, where each operation combines values that have the same key;

- Because datasets can have very large numbers of keys, **reduceByKey()** is not implemented as an action that returns a value to the user program;

- Instead, it returns a new **RDD** consisting of each key and the reduced value for that key;

- **foldByKey()** is quite similar to **fold();** both use a zero value of the same type of the data in out **RDD** and combination function.

Building Spark Application using Maven

Aggregation Examples

- The following examples demonstrate, we can use **reduceByKey()** along with **mapValues()** to compute the per-key average in a very similar manner to how **fold()** and **map()** can be used to compute the entire RDD average;

- Per-key average with **reduceByKey()** and **mapValues()** in ***Python***

rdd.mapValues(lambda x: (x, 1)).reduceByKey(

lambda x, y: (x[0] + y[0], x[1] + y[1]))

Per-key average with reduceByKey() and mapValues() in Scala rdd.mapValues(x => (x, 1)).reduceByKey(

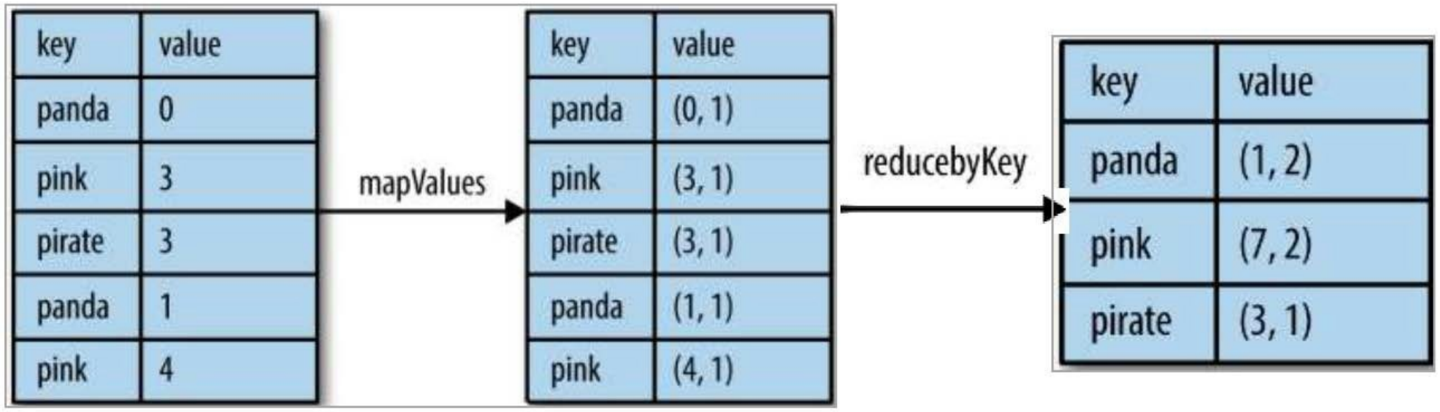
(x, y) => (x._1 + y._1, x._2 + y._2))

Building Spark Application using Maven

Aggregation Examples

Scala

rdd.mapValues(x => (x, 1)).reduceByKey((x, y) => (x._1 + y._1, x._2 + y._2))



key	value
panda	0
pink	3
pirate	3
panda	1
pink	4

key	value
panda	(0, 1)
pink	(3, 1)
pirate	(3, 1)
panda	(1, 1)
pink	(4, 1)

key	value
panda	(1, 2)
pink	(7, 2)
pirate	(3, 1)

Building Spark Application using Maven

Aggregation Example, Word count

- We will use ***flatMap()*** so that we can produce a pair ***RDD*** of words and the number 1 and then sum together all of the words using ***reduceByKey()***

- Word count in ***Python***

```
rdd =

sc.textFile("s3://...")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x + y)
```

Building Spark Application using Maven

Aggregation Example, Word count

- Word count in **Scala**

```
val input = sc.textFile("s3://...")
val words = input.flatMap(x => x.split(" "))
val result = words.map(x => (x, 1)).reduceByKey((x, y) => x + y)
```

Building Spark Application using Maven

Aggregation Example, Word count

- Word count in **Java**

```
JavaRDD<String> input = sc.textFile("s3://...")
JavaRDD<String> words = rdd.flatMap(new FlatMapFunction<String, String>() {
public Iterable<String> call(String x) { return Arrays.asList(x.split(" ")); } });
JavaPairRDD<String, Integer> result = words.mapToPair(
 new PairFunction<String, String, Integer>() {
public Tuple2<String, Integer> call(String x) { return new Tuple2(x, 1); }
}).reduceByKey(
new Function2<Integer, Integer, Integer>() {
public Integer call(Integer a, Integer b) { return a + b; }
});
```

References

1. “Learning Spark” by Holden Karau, Andy Konwinski, Patrick Wendell & Mathei Zaharia, O’Reilly 2015