Week 5-2

# Spark programming


## Big Data

Prof. Hwanjo Yu
POSTECH

# Spark Overview

## Goal : easily work with large scale data in terms of transformations on distributed data

- Traditional distributed computing platforms scale well but have limited APIs (map/reduce)
- Spark has an expressive data focused API which makes writing large scale programs easy
- Spark supports 3 languages
  - Scala, Python, Java
- ***Spark*** is written in ***Scala*** language.
- Spark provides *interactive* Scala shell.
- Scala material
  - https://twitter.github.io/scala_school/
  - http://docs.scala-lang.org/index.html

# Practice environment (Just for practice)

- Download Spark ver.2.2.0 from http://spark.apache.org/downloads.html
- Unzip downloaded file.
- Execute “cmd” to execute spark-shell.
- Change your current directory to Spark folder
  - `>cd C:\spark-2.2.0-bin-hadoop2.7 (example)`
- Go to bin directory
  - `>cd bin`
- Execute Spark Shell
  - `>spark-shell`
- If your computer is not installed JDK8
  - Install JDK
  - http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

# About Scala

- High-level language for the JVM
  - Object oriented + functional programming
- Interoperates with Java
  - Can use any Java class (inherit form, etc.)
  - Can be called from Java code

# Quick Tour of Scala Part 1

**Declaring variables :**

```
var x: Int = 7
var x = 7    //type inferred
val y = "hi" //read-only
```

**Functions:**

```
def square(x: Int) : Int = x*x
def square(x: Int) : Int = {
  x*x
}
def announce(text: String) = {
  println(text)
}
```

**Java equivalent:**

```
int x = 7;

final String y = "hi"
```

**Java equivalent:**

```
int square(int x) {
  return x*x;
}

void announce(String text) {
  System.out.println(text);
}
```

# Scala functions

```
// Anonymous functions

(x:int) => x + 2    // full version

x => x + 2          // type inferred

_ + 2               // placeholder syntax (each argument must be used exactly once)

x => {              //body is a block of code

  val numberToAdd = 2

  x + numberToAdd

}

// Regular functions

def addTwo(x: Int): Int = x + 2
```

# Quick Tour of Scala Part 2

**Processing collections with functional programming**

```
val list = List(1, 2, 3)

list.foreach(x => println(x))     // prints 1, 2, 3
list.foreach(println)             // same


list.map(x => x+2)                // returns a new List(3, 4, 5)
list.map(_ + 2)                   // same


list.filter(x => x % 2 ==1)       // returns a new List(1, 3)
list.filter(_ % 2 == 1)           // same


list.reduce((x,y) => x + y)       // => 6
list.reduce(_ + _)                // same
```

# Functional methods on collections

There are a lot of methods on Scala collections, just ***google Scala Seq***.

| Method on Seq[T] | Explanation |
|---|---|
| map(f: T=>U):Seq[U] | Each element is result of f |
| flatMap(f: T=>Seq[U]):Seq[U] | One to many map |
| filter(f: T=>Boolean): Seq[T] | Keep elements passing f |
| exists(f: T=>Boolean) : Boolean | True if one element passes f |
| forall(f: T=>Boolean):Boolean | True if all elements pass |
| reduce(f: (T,T) =>T):T | Merge elements using f |
| groupBy(f:T=>K):Map[K, List[T]] | Group elements by f |
| softBy(f: T=>K):Seq[T] | Sort elements |
| … | … |

# About Spark

Write programs in terms of transformations on distributed datasets

- **Resilient Distributed Datasets**
  - Immutable, partitioned collections of objects spread across a cluster, stored in RAM or on Disk
  - Built through lazy parallel transformations
  - Automatically rebuilt on failure
- **Operations**
  - Transformations
    - map, filter, groupBy
  - Actions
    - count, collect, save

# API for working with RDDs

map

filter

groupBy

sort

union

join

leftOuterJoin

rightOuterJoin

reduce

count

fold

reduceByKey

groupByKey

cogroup

cross

zip

sample

take

first

partitionBy

mapWith

pipe

save

**Blue** operations are Action
**Black** operations are Transformation

More operations listed in online at http://spark.apache.org/docs/latest/programming-guide.html

# Creating RDDs

```
# Turn a scala collection into an RDD

>sc.parallelize(List(1,2,3))

# Load text file from local FS or HDFS

>sc.textFile("file.txt")

>sc.textFile("hdfs://namenode:9000/path")
```

sc is SparkContext, which is automatically generated in Spark-Shell

# Basic Transformations

```
>val nums = sc.parallelize(List(1,2,3))

// pass each element through a function

>val squares = nums.map(x => x*x) // {1, 4, 9}

// Keep elements passing a predicate

>val even = squares.filter(x => x % 2 == 0) // {4}

// Map each element to zero or more others

>nums.flatMap(x=> 0.to(x))    // => {0, 1, 0, 1, 2, 0, 1, 2, 3}
```

# Basic Actions

```
>val nums = sc.parallelize(List(1,2,3))

// Retrieve RDD contents as a local collection

>nums.collect() // => List(1,2,3)

// Return first k elements

>nums.take(2)   // => List(1,2)

// Count number of elements

>nums.count()  // => 3

// Merge elements with an associative function

>nums.reduce{case (x,y) => x + y} // =>6

// Write elements to a text file

>nums.saveAsTextFile("hdfs://namenode:9000/file.txt")
```

# Working with Key-Value Pairs

A few special operations are only available on RDDs of key-value pairs.

The most common ones are distributed “shuffle” operations.(groupBy, reduceByKey …)

```
val pair = (a,b)
  pair._1 // => a
  pair._2 // => b
```

# Some Key-Value Operations

```
>val pets = sc.parallelize(List((“cat”,1), (“dog”, 1), (“cat”, 2)))
>pets.reduceByKey(_+_) // => ((“cat”,3), (“dog”,1))
>pets.groupByKey()  // => {(“cat”, [1,2]), (“dog”, [1])}
>pets.sortByKey()  // => {(“cat”, 1), (“cat”, 2), (“dog”, 1)}
```

# Other Key-Value Operations

```
>val visits = sc.parallelize(List((“index.txt”, “1.2.3.4”),
                                  (“about.txt”, “3.4.5.6”),
                                  (“index.txt”, “1.3.3.1”)))
>val pageNames = sc.parallelize(List((“index.txt”, “Home”),
                                    (“about.txt”, “About”)))
>visits.join(pageNames)
// (“index.txt”, (“1.2.3.4”, “Home”))
// (“index.txt”, (“1.3.3.1”, “Home”))
// (“about.txt”, (“3.4.5.6”, “About”))
>visits.cogroup(pageNames)
//(“index.txt”, (Seq(“1.2.3.4”, “1.3.3.1”), Seq(“Home”)))
//(“about.txt”, (Seq(“3.4.5.6”), Seq(“About”)))
```

# Persistence or Cache

By default, each transformed RDD may be recomputed each time you run an action on it.

However, you may also persist RDD in memory using the *persist (or cache)* method.

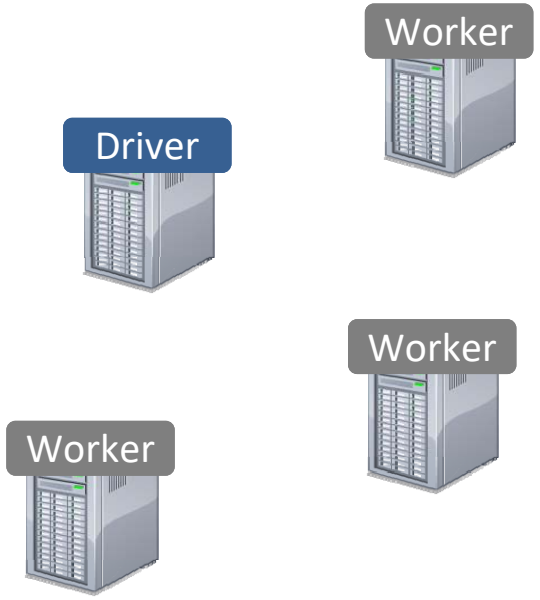
```
>val lines = sc.textFile("hdfs://…")
>val lineLengths = lines.map(s => s.length)
>linelengths.cache() // if you also wanted to use linelength again later
>val totalLength = lineLengths.reduce((a,b) => a + b)
```

※ Difference between persistence and cache : **storage level**

- cache() : memory only
- persistence() : support other storage level (memory and disk)

# Log Mining Example

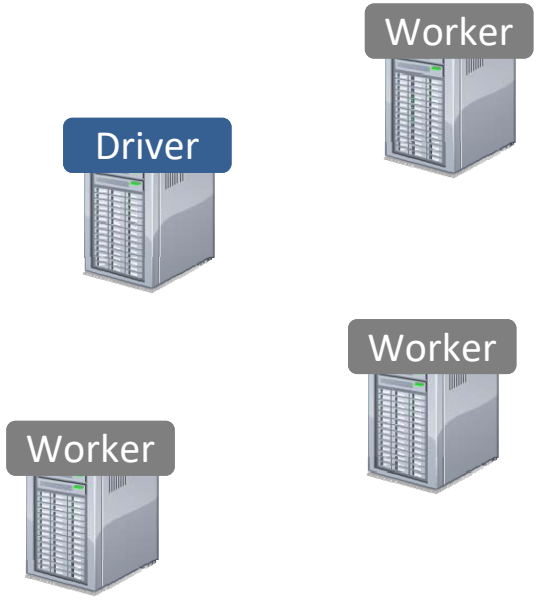
Load error messages from a log into memory, then interactively search for various patterns



# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
```

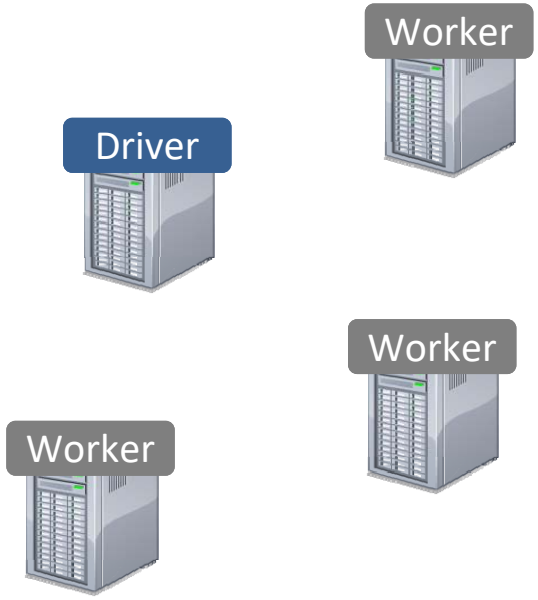


# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

Base RDD

```
val lines = spark.textFile("hdfs://...")
```

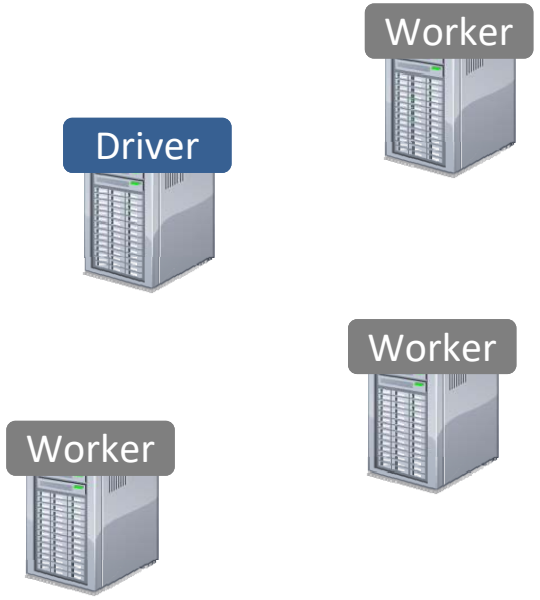


# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

Transformed RDD

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
```

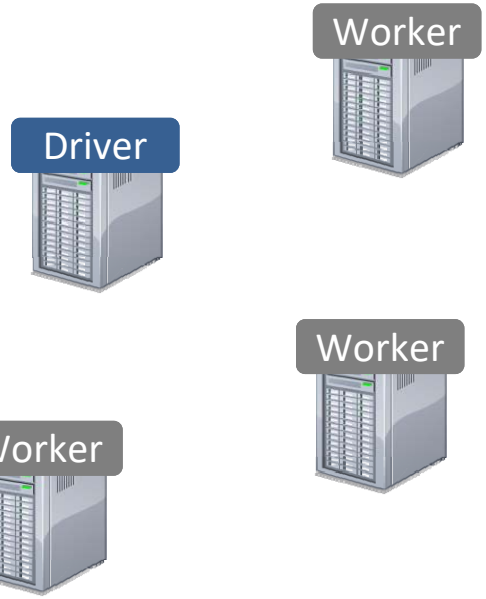


# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))

messages.filter(_.contains("mysql")).count()
```



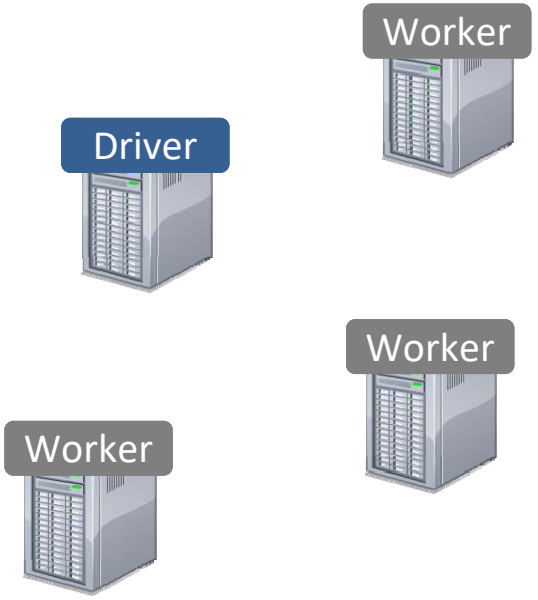
# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile(“hdfs://...”)
val errors = lines.filter(_.startsWith(“ERROR”))
val messages = errors.map(_.split(“\t”)(2))
messages.cache()

messages.filter(_.contains(“mysql”)).count()
```

Cache the RDD



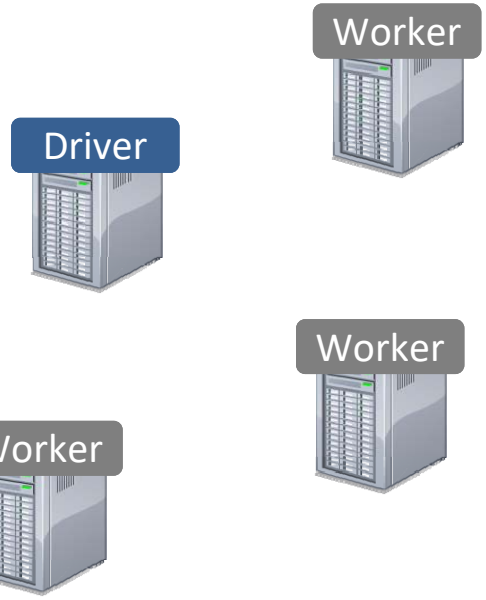
# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
```

Action

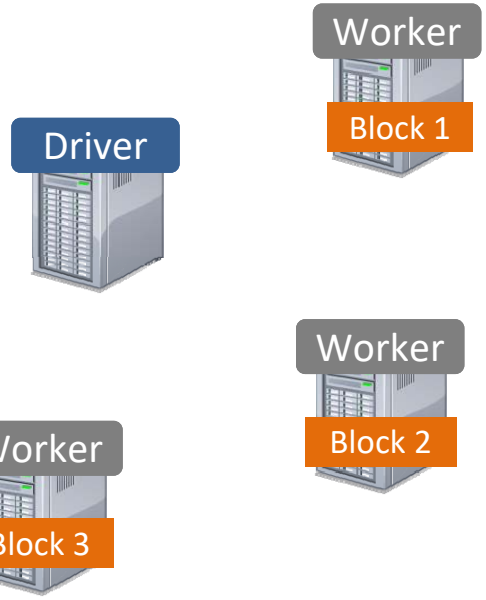


# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
```

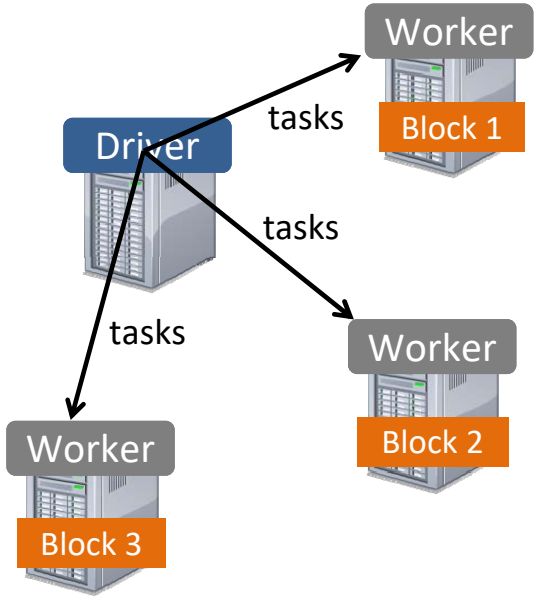


# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
```

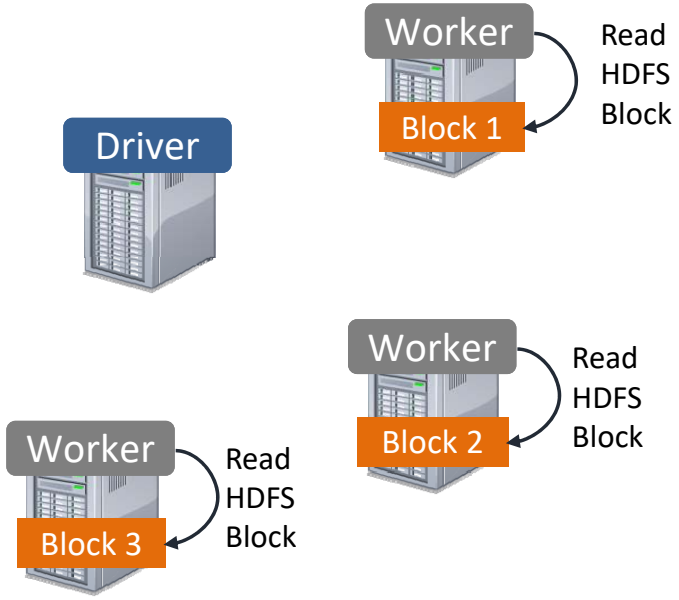


# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
```

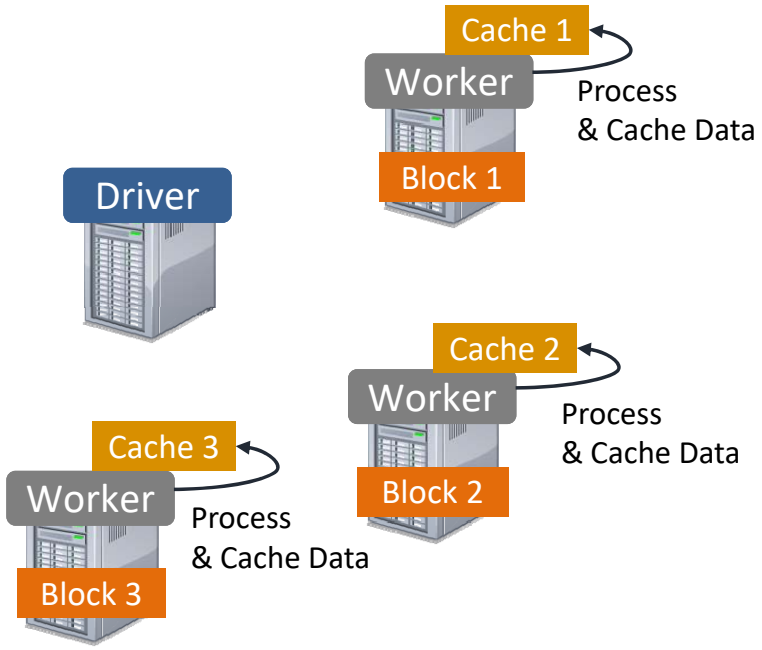


# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
```

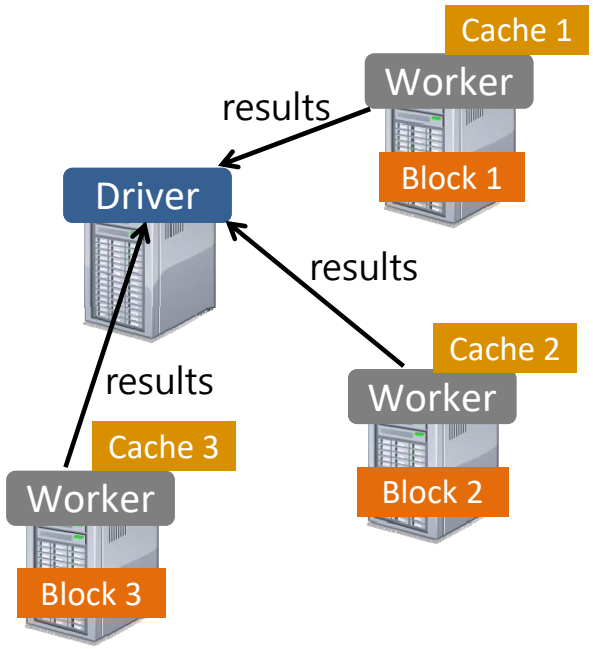


# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
```

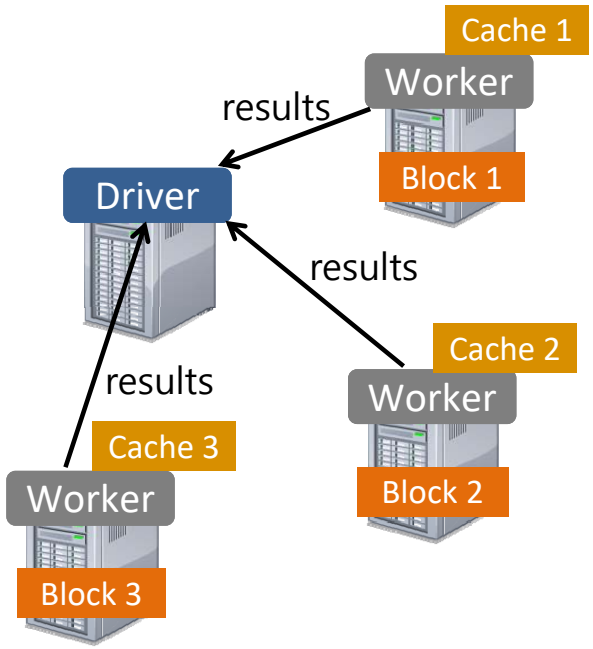


# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

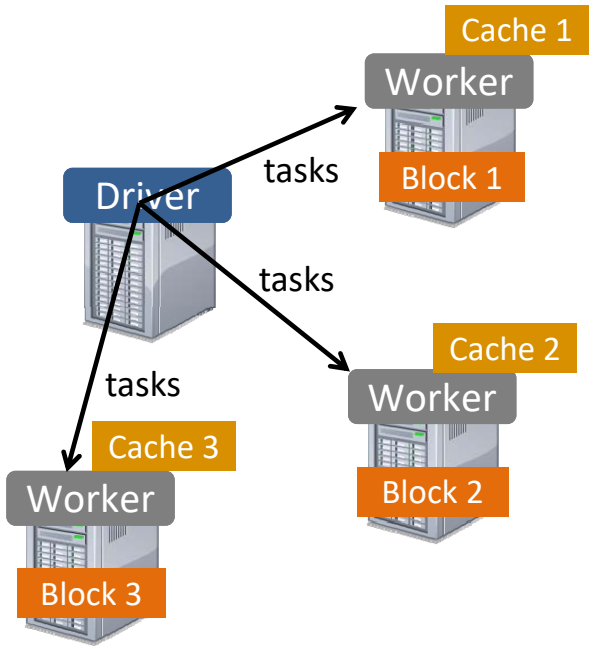


# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

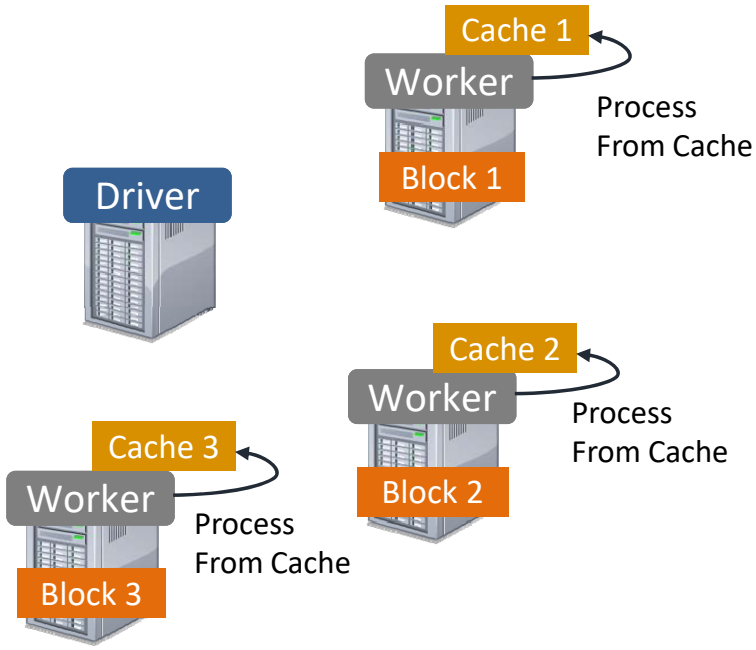


# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

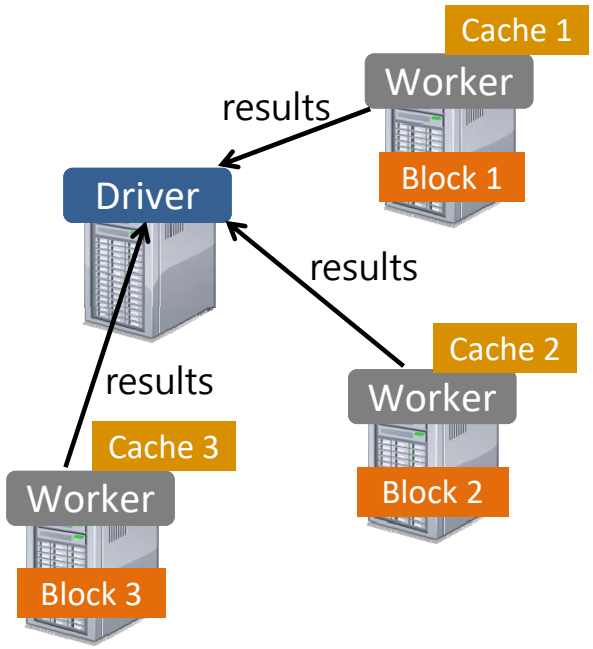


# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```



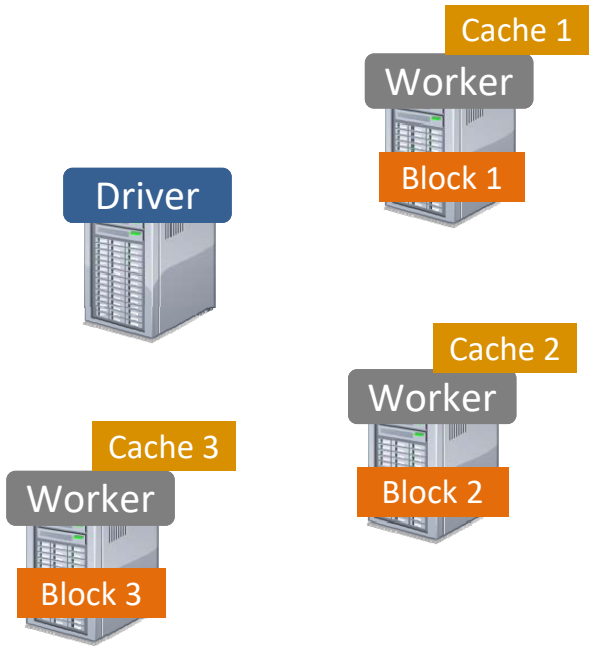
# Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

***Cache data ➔ Faster Results***
***1 TB of log data***
- ***5-7 sec from cache vs. 170s for on-disk***



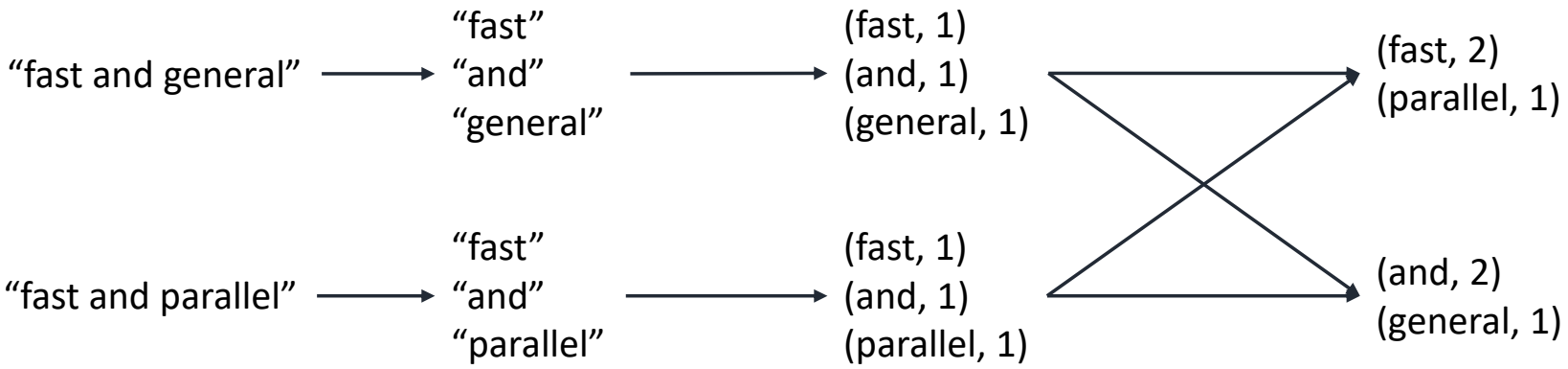
# Let's make Word Count

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

>val counts = text.flatMap(_.split(" "))
                 .map((_, 1))
                 .reduceByKey(_ + _)

>val result = counts.collect()

>result.foreach(println)
```



# Submit your Spark Application to Cluster

- To submit your application to cluster, you need ***Jar*** file.
- We use ***SBT***(Simple Build Tool) as Scala build tool.

Word Count Source code

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
object WordCount {
  def main(args: Array[String]) {
    val textFileAddress = “input_file_path"
    val conf = new SparkConf().setAppName(“WordCount").set("spark.cores.max","3")
    val sc = new SparkContext(conf)
    val textData = sc.textFile(textFileAddress)
    val words = textData.flatMap(line => line.split(“ “))
    val counts = words.map(word=>(word,1)).reduceByKey((x,y) => x+y)
    counts.saveAsTextFile(“your_output_path”)
  }
 }
```

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

# Submit your Spark Application to Cluster

1. Make directories according to sbt directory layout

```
>cd ~

>mkdir –p WordCount/src/main/scala
```

2. Create sbt configuration file which explains dependencies of Spark

```
>cd WordCount

>vim build.sbt
```

3. Create source code

```
>vim src/main/scala/WordCount.scala
```

4. Compile and make jar file (takes some times…)

```
>sbt clean package
```

# Submit your Spark Application to Cluster

5. Submit the jar file and run the program

- *Client Mode*

```
$SPARK_HOME/bin/spark-submit \
--class WordCount \
--master spark://master:7077 \
wordcount_2.11-1.0.jar
```

- *Local Mode*

```
$SPARK_HOME/bin/spark-submit \
--class WordCount \
wordcount_2.11-1.0.jar
```

# HW 3

K-means clustering with Spark

# K-means Clustering

- Most popular clustering algorithm (Unsupervised learning)
- $K$ is the number of clusters
- Given a set of observations $(\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N)$, where each observation is a d-dimensional real vector, k-means clustering aims to partition the N observations into $\mathrm{K}(\leq N)$ sets $\boldsymbol{S} = \{S_1, S_2, \dots, S_K\}$ so as to minimize the within-cluster sum of squares

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|^2$$

- $r_{nk}$ is binary indicator variables $r_{nk} \in \{0, 1\}$, where $k = 1, \dots, K$ describing which of the $K$ clusters the data point $\boldsymbol{x}_n$ is assigned to.
- $\boldsymbol{\mu}_k$ represents the centroid of the cluster.

# K-means Clustering

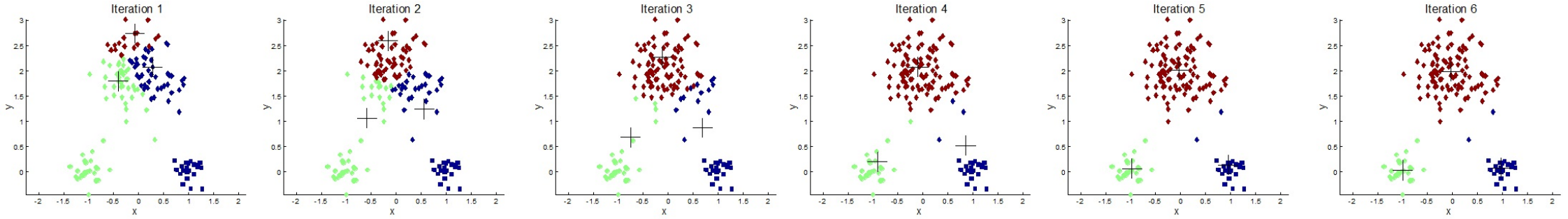
Select *K* points as the initial Centroids

**REPEAT**

Form K clusters by assigning all points to the closest Centroid

Re-compute the Centroids for each cluster

**UNTIL** "The Centroids don't change or all changes are below predefined threshold"




https://apandre.wordpress.com/visible-data/cluster-analysis/

# K-means Clustering

- How to initialize *K* centroids?
  - Pick *K* centroids among data randomly (sampling) or
  - Generate *K* centroids randomly

- What is the distance measure?
  - ***Euclidean distance***

    ex) (1,3,3) ~ (2,2,5) ➔ $\sqrt{(1-2)^2+(3-2)^2+(3-5)^2} = \sqrt{6}$

# Overview

- There are template & data file and output sample in attached zip file.
  - `hw3_template.tar.gz`

- Must
  - Utilize the **RDD operations**
  - Test on the server (Compilation, Execution), Compile error ➔ **Your score is Zero.**
  - All your activity should be done in your home directory
  - ***Do it yourself.***

# Overview

- Goal
  - Implement “K-means clustering algorithm” using **Scala** for Spark
- Submission
  - There must **only** 2 files, source code file and sbt file

    hw3/build.sbt ➔ sbt file, **Don’t modify this file!!**

    hw3/src/main/scala/Kmeans.scala ➔ source code file which you implement
- Input Data (/user/input/spark/kmeans_input.txt on HDFS)
  - 3-dimensional 100 vectors (Double data type, separated by ‘,’)
  - Each is Generated from the one of *three multivariate Gaussian distribution*
  - May contain *duplicate data* points ➔ you should remove duplicate data points first!!

# Overview

- Output
  - Your output files should be written on HDFS.
  - You have to show cluster number for each data point.
  - Your result should be sorted by cluster number.

# Overview

- Program parameter
  - [input path] [output path] [mode] [K = # of cluster]

- If your input file is `/user/input/spark/kmeans_input.txt` (on HDFS),

  **the output should be the same as the sample output**

  because the initial centroids are the same and distance measure is the same (Euclidean)

**0 : Randomly picks K initial centroids from data, third parameter (*K*) is needed**

**Otherwise : User-defined centroids which is stated in the source code, not given as parameters**

# Constraint

- **Don’t modify input file**
- **Don’t modify sbt file and don’t import another package.** It means that you can only use basic RDD operations and built-in data type.
- Distance measure among the data is the **Euclidean distance**
- **Remove duplicate data points** first. There may be the same data points.
- **Output must be the same as sample output** if the mode is 1 and initial centroids are the same as given in source code
- You **must use more than 5 types of RDD operations.**

  ➔This is because it’s Spark programming exercise, not Scala programming exercise.
- **Don’t modify the termination condition** which is the sum of moving distance of each centroid and the threshold is .001. Here, the moving distance is the Euclidean distance as well

# Recommendations

- **Cache RDDs** which is used and accessed several times
- **Use effective data structures**.
  - Scala provides various built-in data structures such as Array, List, Map, Tuple and Set.

# Compilation and Execution

- Compilation
  - Go into your homework directory and make sure the directory layout

```
cd ~/hw2_template
```

  - Put the commands

```
sbt clean package
```

  - You see 'jar' file in `./target/scala-2.11/kmeans_2.11-1.0.jar`

# Compilation and Execution

- Execution (submit your app to the cluster) *mode0*

```
$SPARK_HOME/bin/spark-submit \
--class Kmeans \
–-master spark://localhost:7077 \
target/scala-2.11/kmeans_2.11-1.0.jar \
hdfs://localhost:9000//user/input/spark/kmeans_input.txt \
hdfs://localhost:9000//user/your_id/output \
0 \
3
```

# Compilation and Execution

- Execution (submit your app to the cluster) *mode1*

```
$SPARK_HOME/bin/spark-submit \
--class Kmeans \
–-master spark://localhost:7077 \
target/scala-2.11/kmeans_2.11-1.0.jar \
hdfs://localhost:9000//user/input/spark/kmeans_input.txt \
hdfs://localhost:9000//user/your_id/output \
1
```

# Compilation and Execution

- Check output

```
hdfs dfs –cat your_output_file
```

➔ **I'll check and score the homework in similar way!**

# Appendix

Development Environment Setting

# Spark setting on Ubuntu

- Download Spark program from spark home page
  - https://spark.apache.org/downloads.html
- Move the file to /usr/local/
- Unzip spark file
  - tar xvzf spark-2.2.0-bin-hadoop2.7.tgz
- Change ownership of Spark directory
  - chown –R user:user spark-2.2.0-bin-hadoop2.7

# Spark setting on Ubuntu

- Environment Variables (~/.bash_profile)

  export SPARK_HOME=/user/local/spark-2.2.0-bin-hadoop2.7

  export PATH=$PATH:$SPARK_HOME/bin

- Spark Environment setting (conf/spark-env.sh)
  - SPARK_LOCAL_IP=localhost
  - SPARK_LOCAL_IP=localhost
  - SPARK_LOCAL_DIR="your_directory"
  - SPARK_MASTER_HOST=localhost

# Spark Development Environment (Scala)

- Set Linux environment

  VMware Workstation Player : http://www.vmware.com/products/player/playerpro-evaluation.html

  Ubuntu 16.04 : https://www.ubuntu.com/download/desktop

  Mac Users : Skip this part

- Install Java (Linux)

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
```

- Download and Install Spark and IDE (IntelliJ with Scala)

  Spark 2.2.0 : http://spark.apache.org/downloads.html

  IntelliJ : https://www.jetbrains.com/idea/download/download-thanks.html?code=IIC

# Spark Development Environment (Scala)

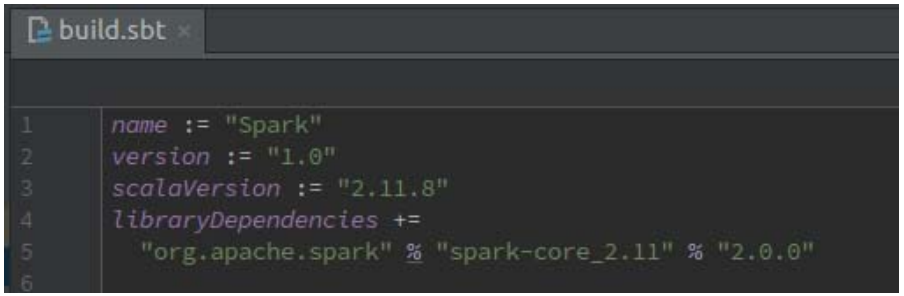
- Making Scala Project with Spark Dependencies
    1. *Execute IntelliJ*
    2. *File – new – project – Scala – sbt – next*
    3. *Check Project SDK, SBT version, Scala version*
        1. *Project SDK : JAVA 1.8*
        2. *SBT version : 0.13.8*
        3. *Scala version : 2.11.8*
    4. *Finish*
- At “build.sbt”

    *Add*

    *libraryDependencies +=*

    *“org.apache.spark” % “spark-core_2.11” % “2.2.0”*

    *Then save and refresh the project and wait……..*




- Make your scala file at src/main/scala/

# Spark Development Environment (Scala)

- Make Jar file
  - *Run - Edit Configurations – add button(+) – Name : sbt - Tasks : clean package – ok*
  - *Run with sbt*

- Check the Jar file
  - *target/scala-2.11/**your_projectname**_2.11-1.0.jar*
  - *The jar file is used to submit the job.*