



# Conceptos y Aplicaciones de Big Data

MapReduce

Desarrollo en Java y en Python

Prof. Waldo Hasperué  
[whasperue@lidi.info.unlp.edu.ar](mailto:whasperue@lidi.info.unlp.edu.ar)

# Temario

- MapReduce
  - Combiner
  - Jobs
  - Pasaje de información

## MAPREDUCE

### Java

- Función combiner
- Ejecución de varios jobs
- Pasaje de información a los TaskTrackers
- Lectura de archivos en el HDFS
- Tipos de datos personalizados

# Combiner - Implementación

```
public class WCCombiner extends Reducer<Text, LongWritable, Text, LongWritable>
{
    public void reduce(Text key, Iterable<LongWritable> values, Context context)
        throws IOException, InterruptedException {

        int times = 0;
        for (@SuppressWarnings("unused") Object val : values) {
            times++;
        }

        context.write(key, new LongWritable(times));
    }
}
```

# Combiner - Implementación

```
public class WCCombiner extends Reducer<Text, LongWritable, Text, LongWritable>
{
    public void reduce(Text key, Iterable<LongWritable> values, Context context)
        throws IOException, InterruptedException {

        int times = 0;
        for (@SuppressWarnings("unused") Object val : values) {
            times++;
        }

        context.write(key, new LongWritable(times));
    }
}
```

En Java la función combiner se implementa como  
subclase de Reducer

# Combiner - Implementación

```
public class Worker extends Configured implements Tool {  
    private Job setupJob() throws IOException{  
        Configuration conf = getConf();  
        Job job = new Job(conf, "WordCount");  
        job.setJarByClass(Worker.class);  
  
        //configure Mapper  
        ...  
  
        //configure combiner  
        job.setCombinerClass(WCCCombiner.class);  
  
        //configure Reducer  
        ...  
    }  
}
```

# Combiner - Implementación

```
public class Worker extends Configured implements Tool {  
    private Job setupJob() throws IOException{  
        Configuration conf = getConf();  
        Job job = new Job(conf, "WordCount");  
        job.setJarByClass(Worker.class);  
  
        //configure Mapper  
        ...  
  
        //configure combiner  
        job.setCombinerClass(WCReducer.class);  
  
        //configure Reducer  
        ...  
    }  
}
```

En la mayoría de los problemas, como función combiner, se usa la misma implementación que el reducer

# Ejemplo - Búsqueda del máximo

```
public class MaxMapper extends Mapper<LongWritable, Text,  
                                     Text, LongWritable>  
{  
    private static Text one = new Text("UniqueKey");  
  
    public void map(LongWritable key, Text value, Context context) {  
        DoubleWritable double_value = new DoubleWritable();  
        double_value.set(Double.valueOf(value.toString()));  
        context.write(one, double_value);  
    }  
}
```



# Ejemplo - Búsqueda del máximo

```
public class MaxMapper extends Mapper<LongWritable, Text,  
                                     Text, LongWritable>  
{  
    private static Text one = new Text("UniqueKey");  
  
    public void map(LongWritable key, Text value, Context context) {  
        DoubleWritable double_value = new DoubleWritable();  
        double_value.set(Double.valueOf(value.toString()));  
        context.write(one, double_value);  
    }  
}
```

El mapper solo escribe el valor con una clave única: para que el mismo reduce reciba todos los valores

# Ejemplo - Búsqueda del máximo

```
public class MaxReducer extends Reducer<Text, DoubleWritable,  
                                     Text , DoubleWritable >  
{  
  
    public void reduce(Text key, Iterable<DoubleWritable> values,  
                       Context context) {  
  
        double maximo = -99999;  
        for (DoubleWritable val : values) {  
            double double_value = val.get();  
            if(double_value > maximo)  
                maximo = double_value;  
        }  
        context.write(new Text("Maximo"), new DoubleWritable(maximo));  
    }  
}
```

# Ejemplo - Búsqueda del máximo

```
public class MaxReducer extends Reducer<Text, DoubleWritable,  
                                     Text , DoubleWritable >  
{  
  
    public void reduce(Text key, Iterable<DoubleWritable> values,  
                       Context context) {  
  
        double maximo = -99999;  
        for (DoubleWritable val : values) {  
            double double_value = val.get();  
            if(double_value > maximo)  
                maximo = double_value;  
        }  
        context.write(new Text("Maximo"), new DoubleWritable(maximo));  
    }  
}
```

El reducer recibe  
todos los valores y  
devuelve el máximo  
de ellos

# Ejemplo - Búsqueda del máximo

```
public class Worker extends Configured implements Tool {  
    private Job setupJob() throws IOException{  
        ...  
  
        job.setMapperClass(MaxMapper.class);  
        job.setReducerClass(MaxReducer.class);  
        job.setCombinerClass(MaxReducer.class);  
  
        ...  
    }  
}
```


En este ejemplo, el combiner es exactamente el mismo que el reducer

# Ejecutando varios jobs

```
public class Worker extends Configured implements Tool {  
    ...  
    public int run(String[] args) throws Exception {  
        Job job;  boolean success;  
  
        job = setupSucProdJob(args);  
        success = job.waitForCompletion(true);  
        if (!success){  
            System.out.println("Error SucProd job"); return -1;  
        }  
  
        job = setupSucJob(args);  
        success = job.waitForCompletion(true);  
        if (!success){  
            System.out.println("Error Suc job"); return -1;  
        }  
        return 0;  
    }  
}
```

# Ejecutando varios jobs

```
public class Worker extends Configured implements Tool {  
    ...  
    public int run(String[] args) throws Exception {  
        Job job;  boolean success;  
  
        job = setupSucProdJob(args);  
        success = job.waitForCompletion(true);  
        if (!success){  
            System.out.println("Error SucProd job"); return -1;  
        }  
  
        job = setupSucJob(args);  
        success = job.waitForCompletion(true);  
        if (!success){  
            System.out.println("Error Suc job"); return -1;  
        }  
        return 0;  
    }  
}
```



El segundo job se ejecutará una vez que finalice el primero

# Ejecutando varios jobs

```
public class Worker extends Configured implements Tool {
    private Job setupSucProdJob(String[] args) throws IOException{
        ...
        String inputDir = args[0]; , outputDir = "intermediateDir";
        if(fs.exists(new Path(outputDir)) fs.delete(new Path(outputDir),true);  }
        FileInputFormat.addInputPath(job, new Path(inputDir));
        FileOutputFormat.setOutputPath(job, new Path(outputDir));
        return job;
    }
    private Job setupSucJob(String[] args) throws IOException{
        ...
        String inputDir = "intermediateDir", outputDir = "; args[1];
        if(fs.exists(new Path(outputDir))) { fs.delete(new Path(outputDir),true);  }
        FileInputFormat.addInputPath(job, new Path(inputDir));
        FileOutputFormat.setOutputPath(job, new Path(outputDir));
        return job;
    }
    ...
}
```

# Ejecutando varios jobs

```
public class Worker extends Configured implements Tool {  
    private Job setupSucProdJob(String[] args) throws IOException{  
        ...  
        String inputDir = args[0]; , outputDir = "intermediateDir";  
        if(fs.exists(new Path(outputDir)) fs.delete(new Path(outputDir),true);  }  
        FileInputFormat.addInputPath(job, new Path(inputDir));  
        FileOutputFormat.setOutputPath(job, new Path(outputDir));  
        return job;  
    }  
    private Job setupSucJob(String[] args) throws IOException{  
        ...  
        String inputDir = "intermediateDir", outputDir = "; args[1];  
        if(fs.exists(new Path(outputDir))) fs.delete(new Path(outputDir),true);  }  
        FileInputFormat.addInputPath(job, new Path(outputDir));  
        FileOutputFormat.setOutputPath(job, new Path(outputDir));  
        return job;  
    }  
    ...  
}
```

El primer job lee el directorio de entrada  
y escribe la salida en un directorio  
temporal intermedio



# Ejecutando varios jobs

```
public class Worker extends Configured implements Tool {
    private Job setupSucProdJob(String[] args) throws IOException{
        ...
        String inputDir = args[0]; , outputDir = "intermediateDir";
        if(fs.exists(new Path(outputDir))      fs.delete(new Path(outputDir),true);  }
        FileInputFormat.addInputPath(job, new Path(inputDir));
        FileOutputFormat.setOutputPath(job, new Path(outputDir));
        return job;
    }
    private Job setupSucJob(String[] args) throws IOException{
        ...
        String inputDir = "intermediateDir", outputDir = "; args[1];
        if(fs.exists(new Path(outputDir))){      fs.delete(new Path(outputDir),true);  }
        FileInputFormat.addInputPath(job, new Path(inputDir));
        FileOutputFormat.setOutputPath(job, new Path(outputDir));
        return job;
    }
    ...
}
```

El segundo job lee el directorio intermedio y escribe la salida en el directorio de salida especificado

# Ejecutando varios jobs

```
public class Worker extends Configured implements Tool {  
    ...  
    public int run(String[] args) throws Exception {  
        Job job;  boolean success, continuar = true;  
  
        job = setupIterableJob(args);  
        while (continuar){  
            success = job.waitForCompletion(true);  
            if (!success){  
                System.out.println("Error job"); return -1;  
            }  
            continuar = evaluarCondicionFin();  
        }  
        return 0;  
    }  
}
```

# Ejecutando varios jobs

```
public class Worker extends Configured implements Tool {
```

```
    ...
```

```
    public int run(String[] args)
```

```
    {
        Job job;  boolean suc
```

```
        job = setupIterableJob
```

```
        while (continuar){
```

```
            success = job.waitForCompletion(true);
```

```
            if (!success){
```

```
                System.out.println("Error job"); return -1;
```

```
            }
```

```
            continuar = evaluarCondicionFin();
```

```
        }
```

```
        return 0;
```

```
    }
```

```
}
```

Esta evaluación involucra leer en HDFS la salida parcial del MapReduce para saber si hay que continuar o no con el Job

# Parametrizando jobs

```
public class Worker extends Configured implements Tool {  
    ...  
    public int run(String[] args) throws Exception {  
        Job job; boolean success, continuar = true;  
  
        job = setupIterableJob(args); int ite = 0;  
        while (continuar){  
            conf.setInt("ite", ++ite);  
  
            success = job.waitForCompletion(true);  
            if (!success){  
                System.out.println("Error job"); return -1;  
            }  
            continuar = eval  
        }  
        return 0;  
    }  
}
```

Mediante el objeto *Configuration* podemos pasarle datos a los mapper y los reducers. Recibe un "label" y un valor asociado

# Parametrizando jobs

```
public class MaxMapper extends Mapper<LongWritable, Text,  
                                     Text, LongWritable>  
{  
    int iteracion;  
    protected void setup(Context context){  
        Configuration conf = context.getConfiguration();  
        iteracion = conf.getInt("ite", 0);  
    }  
}
```

Luego podemos usar la variable iteración en cada una de la llamadas a la función map.

# Parametrizando jobs

```
public class MaxMapper extends Mapper<LongWritable, Text,  
                                     Text, LongWritable>  
{  
    int iteracion;  
    protected void setup(Context context){  
        Configuration conf = context.getConfiguration();  
        iteracion = conf.getInt("ite", 0);  
    }  
}
```

Si el contexto no tiene el label "ite"  
devuelve el valor por defecto  
pasado como argumento.

# Leyendo archivos en el FSD

- Para leer un archivo alojado en el FSD

```
FileSystem fs = FileSystem.get(new
    URI("hdfs://localhost:54310"), conf);
FSDataInputStream fin = fs.open(new
    Path("DirSalida\part-r-00000"));
String line = fin.readLine();
while(line != null){
    System.out.println(line);
    line = fin.readLine();
}
fin.close();
```

# Clave-valor personalizados

- Muchas veces como clave o como valor necesitamos tuplas de datos.
  - Ejemplo del producto más vendido por sucursal.  
 $\langle \text{suc\_prod}, \text{cant} \rangle$  ,  $\langle \text{suc}, (\text{prod\_max}) \rangle$
- Todo el tiempo hay que hacer concatenación y splits de strings.
- Es más "elegante" usar tipos writable personalizados.



# Clave-valor personalizados

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableUtils;

public class SucProdWritable implements Writable {
    String sucursal, producto;

    public SucProdWritable () {}
    public SucProdWritable (String s, String p) {
        this.sucursal = s;
        this.producto = p;
    }

    ...
}
```

# Clave-valor personalizados

...

```
public void readFields(DataInput in) throws IOException {  
    sucursal = WritableUtils.readString(in);  
    producto = WritableUtils.readString(in);  
}
```

```
public void write(DataOutput out) throws IOException {  
    WritableUtils.writeString(out, sucursal);  
    WritableUtils.writeString(out, producto);  
}
```

...

# Clave-valor personalizados

...

```
@Override
public String toString() {
    return this.sucursal+ "\t" + this.producto ;
}

public String getSucursal() {
    return sucursal;
}

public String getProducto() {
    return producto;
}
}
```

# Formatos de entrada

- Todos los archivos se dividen en splits. Un split es lo que procesa un mismo proceso mapper.
- Cada split se divide en registros. Un registro es lo que procesa una invocación a la función map.
- Por lo general un split es de 64MB
- Hasta ahora vimos que un registro es una línea de texto.

# Formatos de entrada

- InputSplit es la interface que representa los split y tiene la longitud y la locación de un split.
- Es una referencia a los datos.

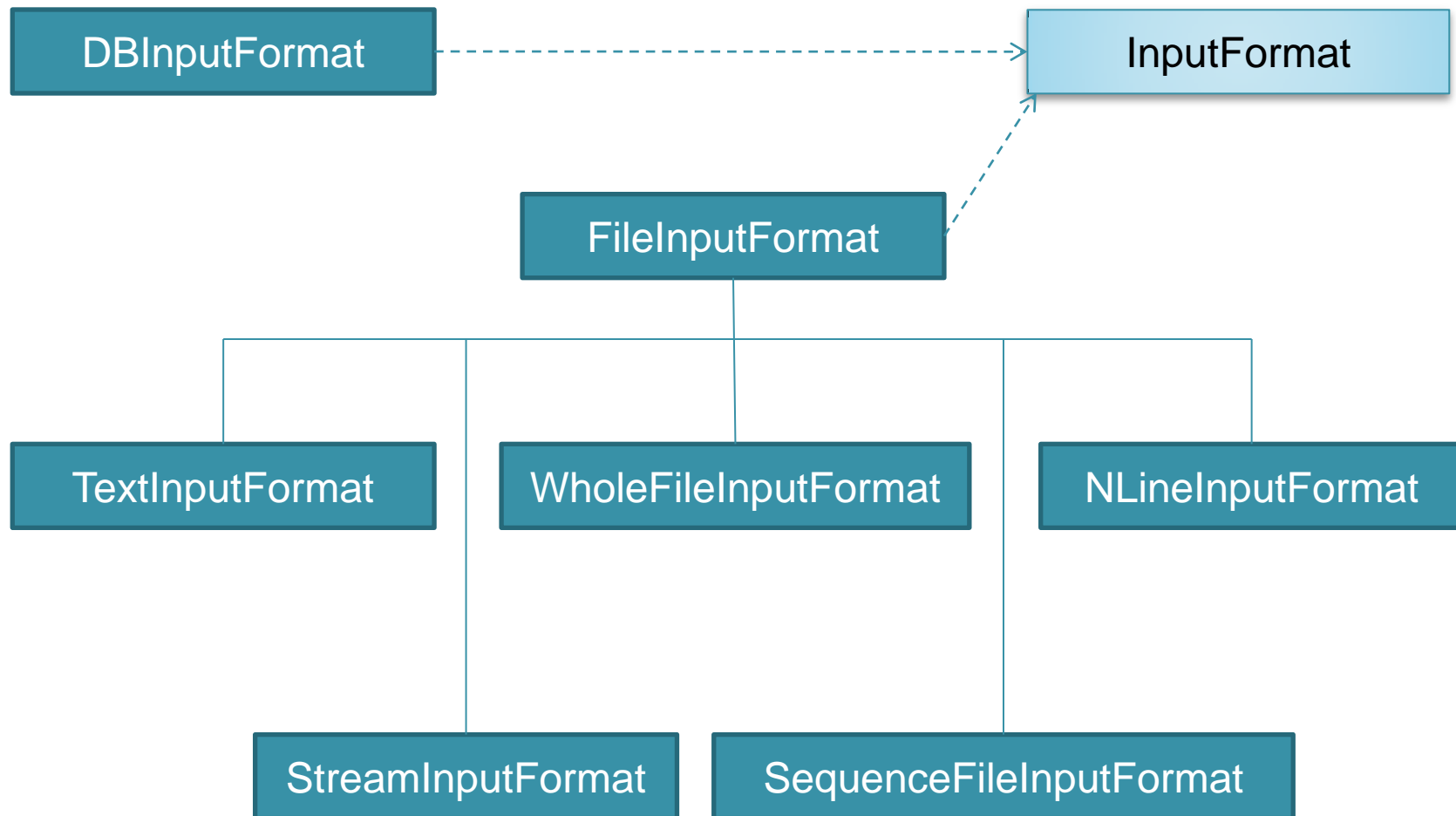
```
public interface InputSplit extends Writable {  
    long getLength();  
    String[] getLocations();  
}
```



# Formatos de entrada

- Hadoop puede procesar varios tipos de entrada
  - Archivos de texto
    - XML
  - Archivos binarios
    - Imágenes, videos, audios
  - Bases de datos

# Formatos de entrada





## MAPREDUCE

### Python

- Función combiner
- Ejecución de varios jobs
- Pasaje de información a los TaskTrackers
- Lectura de archivos en el HDFS

# Función combiner

- En python la función combiner es un archivo con extensión .py el cual se debe pasar como argumento al comando hadoop.

```
hadoop jar hadoop-streaming-2.6.0.jar  
-input entrada  
-output salida  
-mapper /mapper.py  
-reducer /reducer.py  
-combiner /combiner.py
```

# Función combiner

- En python la función combiner es un archivo con extensión .py el cual se debe pasar como argumento al comando hadoop.

`hadoop jar hadoop`

`-input entrada`

`-output salida`

`-mapper /mapper.py`

`-reducer /reducer.py`

`-combiner /combiner.py`

El archivo para la función combiner podría ser el mismo archivo que para la etapa de reduce, si corresponde.

# Ejecutando varios jobs

- En python el proceso driver es un script bash o un programa en python. La ejecución de más de un job se realiza con la llamada a más de un comando hadoop.

```
#!/bin/bash
```

```
hadoop jar hadoop-streaming-2.6.0.jar  
    -input entrada      -output temporal  
    -mapper /map1.py    -reducer /red1.py
```

```
hadoop jar hadoop-streaming-2.6.0.jar  
    -input temporal     -output salida  
    -mapper /map2.py    -reducer /red2.py
```

# Ejecutando varios jobs

- En python el proceso driver es un script bash o un programa en python. La ejecución de cada job se hace con la llamada a `hadoop`.

El directorio salida del primer job es la entrada en el segundo job

```
#!/bin/bash
hadoop jar hadoop-streaming-2.6.0.jar
    -input entrada -output temporal
    -mapper /map1.py -reducer /red1.py
hadoop jar hadoop-streaming-2.6.0.jar
    -input temporal -output salida
    -mapper /map2.py -reducer /red2.py
```

# Ejecutando varios jobs

- Hay un proyecto que intenta resolver la ejecución de varios jobs en un único llamado del comando hadoop.

<https://github.com/hyonaldo/hadoop-multiple-streaming>

- Existe otro proyecto que permite la ejecución de varios jobs escribiendo solo código python.

<https://pythonhosted.org/mrjob/>

# Parametrizando jobs

- El mismo script bash que actúa como driver se utiliza para la parametrización de los jobs.

```
hadoop jar hadoop-streaming-2.6.0.jar  
  -input entrada  
  -output salida  
  -mapper /mapper.py  arg1 arg2  
  -reducer /reducer.py  arg3 arg4
```

# Parametrizando jobs

- Los argumentos enviados por línea de comando son recibidos en el script de python.

mapper.py

```
import sys
arg1 = sys.argv[1]
arg2 = sys.argv[2]
```



# Leyendo archivos en el DFS

- Para leer archivos que están en el DFS desde python, primero hay que copiarlo al FS local, ejecutando el comando hdfs.

driver.py

```
import os
while(True):
    os.system("hadoop ...")
    os.system("hdfs dfs -copyToLocal ...")
    f = open ("filename", "r")
    datos = f.readlines()
    f.close()
```