

Dominando MapReduce

Un Viaje Práctico de Java a Python en Hadoop

El Objetivo de Nuestro Viaje

Al final de esta guía, habrás dominado el ciclo de vida completo de un job de MapReduce.

Nuestro objetivo es claro:

- **Desarrollar** dos aplicaciones fundamentales en Java.
- **Compilar y empaquetar** el código para su ejecución distribuida.
- **Lanzar y monitorizar** los jobs en un clúster Hadoop.
- **Verificar** los resultados directamente en HDFS.
- **Explorar** una alternativa ágil utilizando Python Streaming.



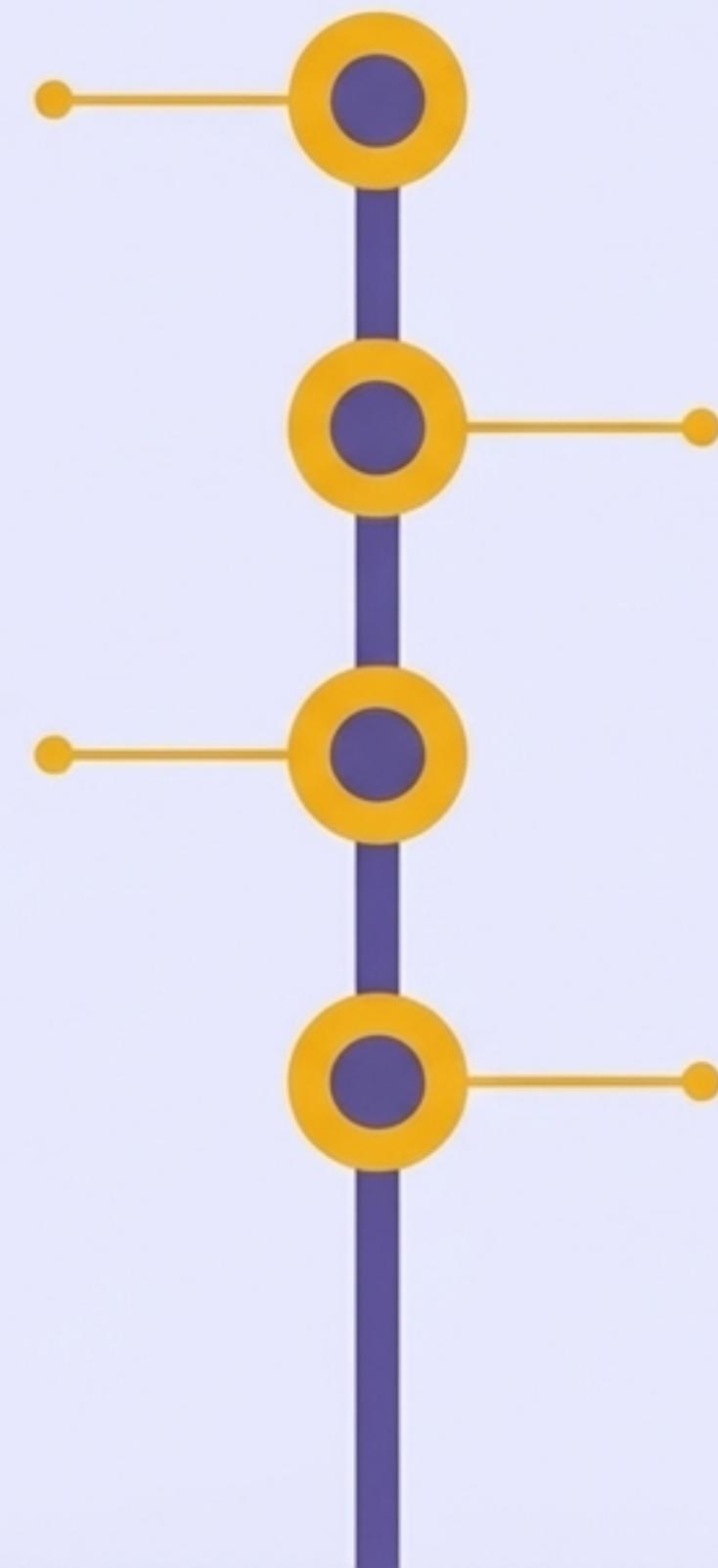
Nuestra Hoja de Ruta

Fase 1: Preparando el Terreno

Configuración esencial del entorno y los datos.

Fase 3: Subiendo de Nivel - Análisis de Logs

Una tarea más compleja que introduce la actualización de paquetes.



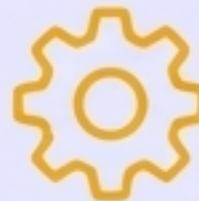
Fase 2: Primer Hito - El Contador de Palabras

Nuestro primer job completo en Java, de código a resultado.

Fase 4: Una Nueva Ruta - El Poder del Streaming

Implementando la misma lógica con la flexibilidad de Python.

Fase 1: Preparando el Terreno



1. Configurar CLASSPATH

Define la variable de entorno para que Hadoop encuentre las librerías de compilación de Java.

```
export  
HADOOP_CLASSPATH=$JAVA_  
HOME/lib/tools.jar
```



2. Crear Workspace y Datos

Organiza el código fuente (.java) y asegura que los datos de entrada (quijote.txt`, `log1.log) residen en HDFS.

Ubicación Sugerida:
`/practicas` en HDFS.



3. Activar Monitorización

Inicia el Job History Server para visualizar el progreso y los detalles de los jobs en la UI web.

Verificación: La interfaz web debe estar disponible en `nodo1:8088`.

Anatomía del Código: `ContarPalabras.java`

El Mapper (`TokenizerMapper`)

- ⚙️ **Función:** Recibe una línea de texto.
- 🧩 **Lógica:** La tokeniza en palabras.
- ➡️ **Salida:** Emite un par clave-valor `(palabra, 1)` por cada palabra encontrada.

```
public void map(Object key, Text value, Context context)
    throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
    }
}
```

El Reducer (`IntSumReducer`)

- ⚙️ **Función:** Recibe una palabra y una lista de '1s'.
- 🧩 **Lógica:** Suma todos los valores para obtener el conteo total.
- ➡️ **Salida:** Emite el par final `(palabra, suma_total)`.

```
public void reduce(Text key, Iterable<IntWritable> values, Context context)
    throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
        sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
}
```

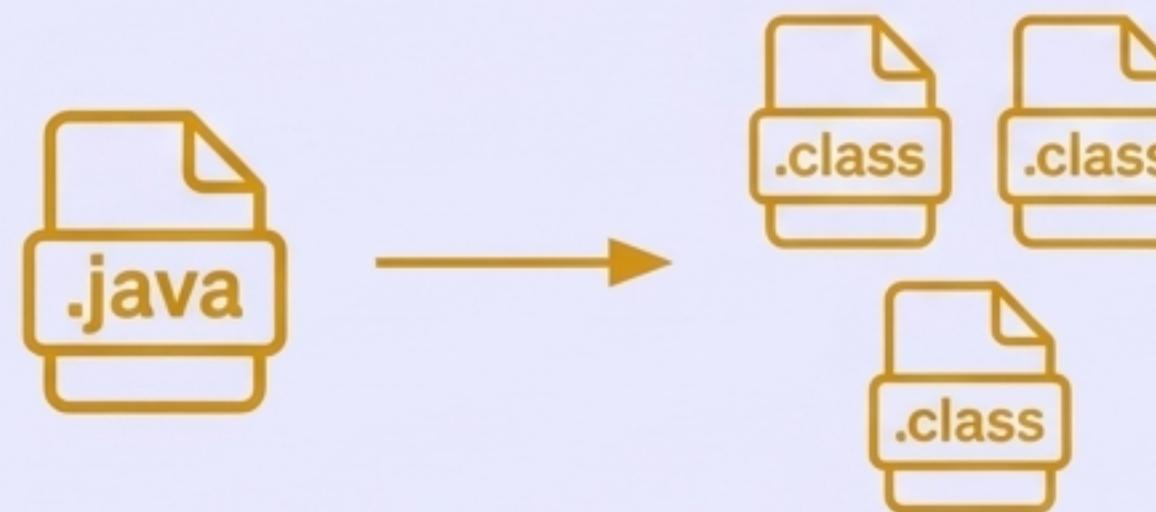
El Driver (`main`)

- ⚙️ **Función:** Orquesta el job completo.
- 🧩 **Lógica:** Configura las clases Mapper y Reducer, y define las rutas de entrada y salida en HDFS.

Del Código al Paquete: Creando el Archivo JAR

Paso 1: Compilar el Código Fuente

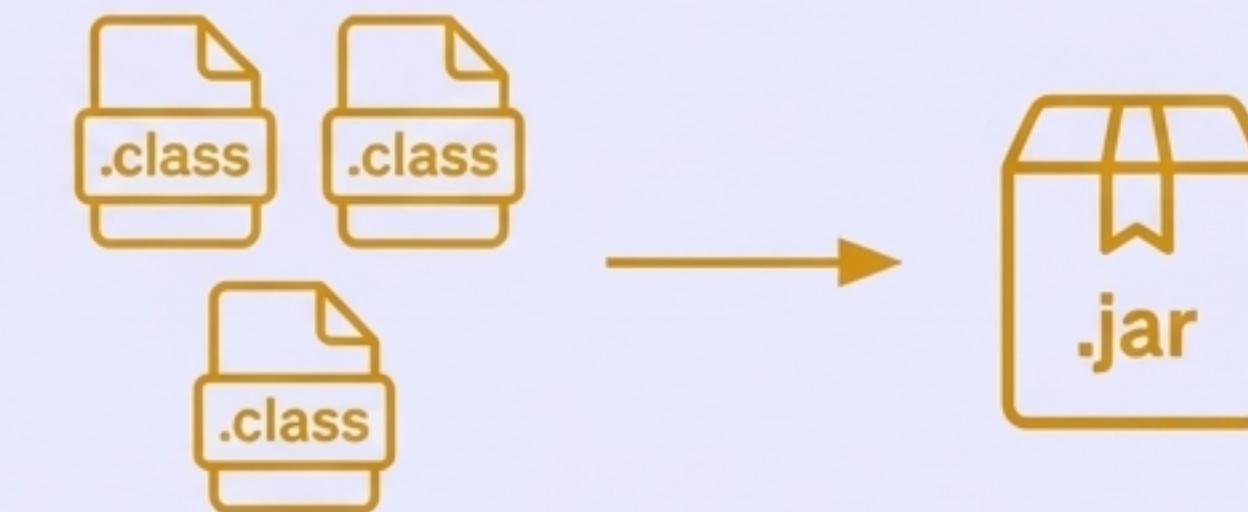
```
hadoop com.sun.tools.javac.Main  
ContarPalabras.java
```



Se generan los ficheros ` `.class` (uno para cada clase: `ContarPalabras`, `TokenizerMapper`, `IntSumReducer`).

Paso 2: Empaquetar las Clases

```
jar cf mi_libreria.jar Contar*.class
```



Se crea un único archivo ` `mi_libreria.jar` que contiene todas las clases compiladas, listo para ser distribuido en el clúster.

Lanzamiento, Monitorización y Verificación



1. Ejecutar el Job

```
hadoop jar mi_libreria.jar  
ContarPalabras  
/practicas/quijote.txt  
/resultado3
```



2. Monitorizar el Progreso

- **Consola**: Observar el avance de las fases map.
- **Interfaz Web (nodo1:8088)**: Navegar a la vista de "History" para ver detalles de los Mappers y Reducers lanzados y los nodos donde se ejecutaron.



3. Verificar la Salida en HDFS

```
hdfs dfs -cat  
/resultado3/part-r-00000
```

| Resultado | Esperado: |
|-----------|-----------|
| amigo | 150 |
| batalla | 87 |
| caballero | 250 |
| ... | |

Fase 3: Analizando Logs con `AnalizarLog.java`

Objetivo: Calcular el tamaño **máximo, mínimo y la media** de los **ficheros solicitados** en un log de acceso web.

Mapper (`AMapper`)

Clave: Utiliza una expresión regular (`httplogPattern`) para parsear cada línea del log.

Lógica Destacada: Extrae el tamaño del fichero (grupo 5 del regex) y lo emite con una clave fija: `("msgSize", tamaño)`.

```
public static class AMapper extends Mapper<Object, Text, Text, IntWritable> {
    private static final Pattern httplogPattern = Pattern.compile(
        "^(\\S+) - - [((\\d{2})/[A-Za-z]{3}/\\d{4}):\\d{2}:\\d{2}:[+\\/-]\\d{4})\\] \\[(GET|POST) .* HTTP/.*/\\] (\\d{3}) (\\d{+})\\d{+}\\");
    private final IntWritable tamaño = new IntWritable();
    private final Text clave = new Text("msgSize");

    public void map(Object key, Text value, Context context) throws
        IOException, InterruptedException {
        Matcher matcher = httplogPattern.matcher(value.toString());
        if (matcher.matches()) {
            tamaño.set(Integer.parseInt(matcher.group(5)));
            context.write(clave, tamaño);
        }
    }
}
```

Reducer (`AReducer`)

Clave: Recibe todos los tamaños bajo la clave "msgSize".

Lógica Destacada: Itera sobre los valores para calcular `min`, `max`, `sum`, y `count`, y finalmente emite la media.

```
public static class AReducer extends Reducer<Text, IntWritable, Text, DoubleWritable> {
    private final DoubleWritable media = new DoubleWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        int min = Integer.MAX_VALUE;
        int max = Integer.MIN_VALUE;
        long sum = 0;
        int count = 0;
        for (IntWritable val : values) {
            int v = val.get();
            if (v < min) min = v;
            if (v > max) max = v;
            sum += v;
            count++;
        }
        media.set((double) sum / count);
        context.write(key, media);
    }
}
```

Actualización del Paquete y Ejecución del Análisis

1. Compilar el Nuevo Código

```
hadoop com.sun.tools.javac.Main AnalizarLog.java
```

2. Actualizar el JAR (Paso Clave)

Se utiliza el flag `uf` (update file) para añadir las nuevas clases al JAR existente sin reconstruirlo.

```
jar uf mi_libreria.jar Analizar*.class ← update file
```

3. Ejecutar el Nuevo Job

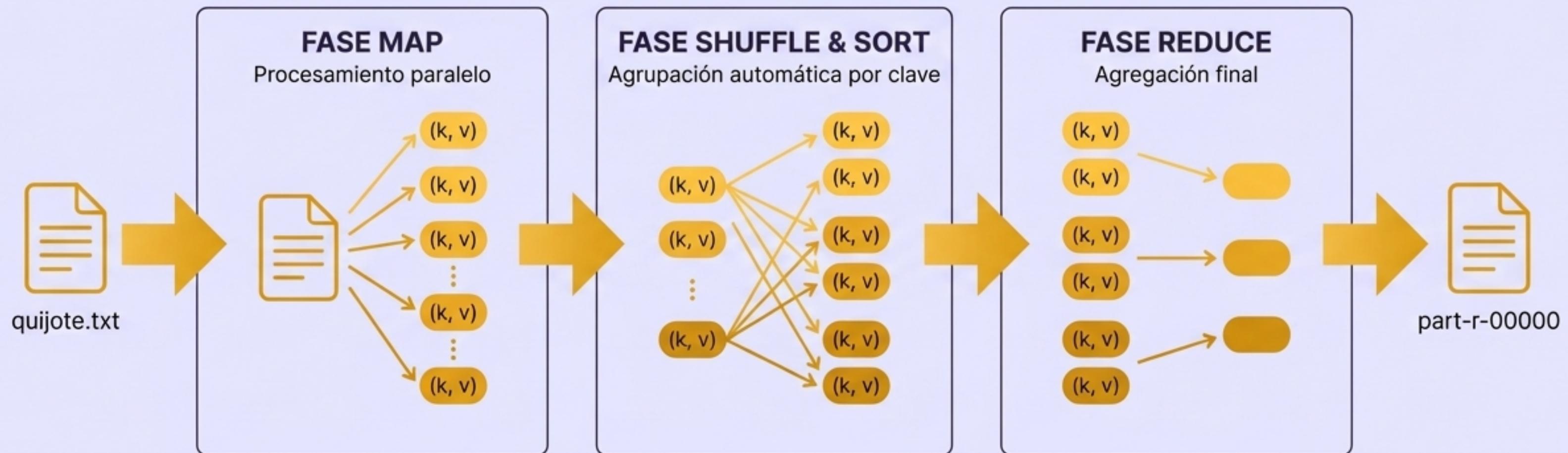
```
hadoop jar mi_libreria.jar AnalizarLog /practicas/log1.log /resultado_log
```

4. Verificar el Resultado

Se muestra la salida esperada del fichero `part-r-00000`.

| | |
|------|---------|
| Mean | 1150 |
| Max | 6823936 |
| Min | 0 |

Resumen Conceptual: El Flujo de Datos MapReduce



La fase Map extrae información relevante (palabras, tamaños de fichero). La fase **Reduce** realiza la agregación final (suma, media, mínimo, máximo).

Fase 4: Python y Hadoop Streaming, una Ruta Alternativa

Concepto: Hadoop Streaming permite usar cualquier ejecutable (como un script de Python) para las fases de Map y Reduce. La comunicación se realiza a través de la entrada/salida estándar.

El Mapper: `pymap.py`

Lee líneas de `sys.stdin`, las divide en palabras y escribe `palabra\t1` en `sys.stdout`.

```
#!/usr/bin/env python
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print '%s\t1' % word
```

El Reducer: `pyreduce.py`

Lee la salida ordenada del mapper, agrega los conteos para la misma palabra y escribe `palabra\tconteo_total` en `sys.stdout`.

```
#!/usr/bin/env python
from operator import itemgetter
import sys
last_word = None
last_count = 0
cur_word = None
for line in sys.stdin:
    line = line.strip()
    cur_word, count = line.split('\t', 1)
    count = int(count)
    if last_word == cur_word:
        last_count += count
    else:
        if last_word:
            print '%s\t%s' % (last_word, last_count)
        last_count = count
        last_word = cur_word
if last_word == cur_word:
    print '%s\t%s' % (last_word, last_count)
```

Descomponiendo el Comando de Streaming

```
hadoop jar /opt/hadoop/share/hadoop/tools/lib/hadoop-streaming-2.9.0.jar \
    -file pymap.py -mapper pymap.py \
    -file pyreduce.py -reducer pyreduce.py \
    -input /practicas/quijote.txt -output /resultado4
```



La **herramienta** de Streaming que orquesta el proceso.



Distribuye el script del mapper a los nodos y lo asigna a la fase Map.



Distribuye el script del reducer y lo asigna a la fase Reduce.



Define las rutas de datos, igual que en un job de Java.

Nota: Al verificar la UI web, se confirma que Hadoop lo ejecuta como un job MapReduce nativo.

Java Nativo vs. Python Streaming: Cuándo Usar Cada Uno

| Característica | Java Nativo | Python con Streaming |
|----------------|---|---|
| Rendimiento | Generalmente mayor (compilado, JVM) | Menor (overhead de streaming, interpretado) |
| Complejidad | Más verbosidad (“boilerplate”), tipos de datos Hadoop | Código más simple y rápido de escribir |
| Flexibilidad | Estrictamente tipado, orientado a JVM | Usa cualquier lenguaje, ideal para prototipado rápido |
| Ecosistema | Integración profunda y nativa con herramientas Hadoop | Integración a través de una capa de abstracción |

Habilidades Adquiridas en Este Viaje

- ✓ **Configurar** un entorno de desarrollo Hadoop funcional.
- ✓ **Implementar** el patrón MapReduce en Java desde cero.
- ✓ **Compilar y empaquetar** aplicaciones Java, incluyendo la creación y actualización de archivos JAR.
- ✓ **Lanzar y monitorizar** jobs, interpretando la salida de la consola y la UI web.
- ✓ **Utilizar Hadoop Streaming** para ejecutar scripts de Python como jobs MapReduce.
- ✓ **Verificar** los resultados de cualquier job directamente en HDFS.

El Patrón MapReduce: Una Herramienta Fundamental

Has completado el ciclo de vida de desarrollo, desde el código fuente hasta el resultado verificado, usando tanto la robustez de Java como la agilidad de Python. Este dominio de MapReduce es la base para entender arquitecturas de datos distribuidos más complejas.

Explora otros entornos de Streaming y frameworks de procesamiento de nivel superior construidos sobre estos principios.