

Laboratorio 3 - Frameworks

Computación Distribuida con Apache Spark

Paradigmas 2025 - FAMAF

Introducción

En este laboratorio vamos a trabajar sobre el código del Laboratorio 2 ya implementado por ustedes. El objetivo es extender el lector automático de feeds RSS para que soporte el procesamiento de grandes volúmenes de datos (big data), tanto en la etapa de descarga como en el cálculo de entidades nombradas. Para lograr esto, vamos a utilizar Apache Spark, uno de los frameworks más utilizados actualmente para computación distribuida sobre grandes conjuntos de datos.

Fecha límite de entrega: Viernes 6 de junio de 2025

Problema

Estructura del lab anterior, abstrayendo los detalles técnicos:

1. Parsear *subscriptions.json*. Esto devuelve conceptualmente `List<String>` con urls.
2. Obtener el contenido de cada url en la `List<String>` con un request HTTP.
3. Parsear el contenido de cada feed, obteniendo `List<Article>`.
4. Extraer las entidades nombradas de cada artículo, obteniendo `<List<List<NamedEntity>>>`.
5. Agregar las entidades nombradas contando cuantas veces ocurren, devolviendo un `Map<String, Integer>` (que en realidad lo devuelven por pantalla).

En el archivo *feeds.json* del laboratorio anterior, el usuario configura los feeds que la aplicación debe consumir. Es decir, este archivo define una colección de fuentes *feed_1*, ..., *feed_n*, sobre las cuales luego se computan las entidades nombradas presentes en los títulos y descripciones de los artículos que contienen.

Ahora bien, ¿qué sucede si este conjunto de feeds escala significativamente, es decir, si $n \rightarrow \infty$?

Este crecimiento plantea dos cuellos de botella:

1. Descarga de feeds: Las llamadas HTTP para obtener los feeds pueden demorar mucho, ya que dependen de servidores externos. Sin embargo, son independientes entre sí y, por lo tanto, pueden ejecutarse en paralelo.
2. Procesamiento de texto: A medida que aumenta la cantidad de artículos, también lo hace la cantidad de texto sobre la cual se deben computar entidades nombradas. Procesar esto de forma secuencial sería ineficiente y escalaría mal, particularmente si en lugar de usar una heurística simple utilizáramos un modelo de machine learning complejo.

Posible Solución

La solución más común en entornos reales es aplicar el paradigma de computación distribuida: repartir los datos entre varias computadoras (nodos) para que cada una procese una parte, en paralelo. Esta estrategia permite reducir significativamente los tiempos de cómputo.

Para facilitar este enfoque, existen frameworks como Apache Spark, que abstraen la complejidad del procesamiento distribuido y permiten al programador concentrarse en la lógica del problema, sin preocuparse por la infraestructura subyacente.

Consigna del laboratorio

Crear un programa que lea las URL a partir del archivo de subscriptions.json, descargue cada feed y cuente las entidades nombradas usando Spark. Usaremos paralelización de dos formas:

1. Para cada url, distribuir la descarga y parseo de cada uno de los feeds. Notar que en este caso, hay un worker por cada feed.
2. Para cada artículo, distribuir el conteo de entidades nombradas. Notar que hay un worker por cada artículo.

Finalmente, imprimir por pantalla una lista de **<entidad nombrada>: <conteo>**, donde se cuenta la aparición de la entidad nombrada EN TODOS LOS ARTÍCULOS DE TODOS LOS FEEDS.

Utilizar el siguiente archivo con una lista más extensa de feeds: [subscriptions.json](#)

Notas:

1. Simplificamos el problema **utilizando solo Feeds RSS**, es decir, con formato XML.
2. Idealmente, entre el paso 1 y el paso 2, se guardarán los archivos a disco, ya que si estamos trabajando con “big data”, no entrarían en memoria. No es necesario implementar esta lógica.

Spark

Spark se considera un **framework** porque proporciona una **estructura reutilizable** y **abstracciones de alto nivel** para desarrollar programas distribuidos de procesamiento de datos. En lugar de tener que escribir desde cero todo el código para distribuir datos, coordinar tareas entre máquinas, recuperar errores, etc., Spark ya implementa todo esto. Nosotros solo tenemos que enfocarnos en **qué** queremos computar, no en **cómo** hacerlo en paralelo.

Arquitectura de Apache Spark

Apache Spark funciona sobre una arquitectura de tipo Master-Worker, en la cual el trabajo se distribuye entre varias computadoras. Esta arquitectura se puede describir así:

- Existe una única computadora llamada Master, que coordina el trabajo.
- Hay múltiples computadoras llamadas Workers (trabajadores), que realizan el procesamiento real de los datos.

Este conjunto de computadoras forma lo que se llama un **cluster** de Spark. El flujo de ejecución es el siguiente:

1. El usuario envía una tarea o trabajo (job) al Master.
2. El Master divide el conjunto de datos en partes más pequeñas (particiones) y asigna cada parte a un Worker.
3. Cada Worker procesa su parte de los datos de forma paralela.
4. Una vez que todos los Workers terminan, el Master recolecta los resultados parciales y los combina para producir el resultado final.

Desde el punto de vista del usuario, todo esto ocurre de forma transparente: parece como si se estuviera utilizando una sola supercomputadora.

(Ver video: <https://www.youtube.com/watch?v=B038xGcnaG4&t=20s>)



Java API de Spark

Para enviar una tarea al cluster desde nuestra aplicación, debemos utilizar la API diseñada para Java que ofrece Spark. Ustedes deberán investigar sobre esta API para

luego saber como llamarla desde su aplicación (<https://spark.apache.org/docs/latest/api/java/index.html>).

A modo de ejemplo, el siguiente código computa con Spark la cantidad de palabras que tiene un archivo de texto como [El Manifiesto Comunista](#):

```
Java
import scala.Tuple2;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.sql.SparkSession;

import java.util.Arrays;
import java.util.List;
import java.util.regex.Pattern;

public final class WordCounter {

    // Expresión regular para separar por espacios
    private static final Pattern SPACE = Pattern.compile(" ");

    public static void main(String[] args) throws Exception {

        // Verifica que se haya pasado al menos un argumento (el archivo de entrada)
        if (args.length < 1) {
            System.err.println("Usage: WordCounter <file>");
            System.exit(1);
        }

        // Crea una sesión de Spark en modo local (usa todos los núcleos de la
        máquina)
        SparkSession spark = SparkSession
            .builder()
            .appName("WordCounter")
            .master("local[*]") // Ejecuta en modo local usando todos los núcleos
            .getOrCreate();

        // Lee el archivo de texto especificado como RDD de líneas
        JavaRDD<String> lines = spark.read().textFile(args[0]).javaRDD();
```

```

// Divide cada línea en palabras usando espacios como separador
JavaRDD<String> words = lines.flatMap(s ->
Arrays.asList(SPACE.split(s)).iterator());

// Asocia a cada palabra el número 1 (pares <palabra, 1>)
JavaPairRDD<String, Integer> ones = words.mapToPair(s -> new Tuple2<>(s, 1));

// Suma los valores (conteos) asociados a la misma palabra (reduceByKey)
JavaPairRDD<String, Integer> counts = ones.reduceByKey((i1, i2) -> i1 + i2);

// Trae el resultado final a la máquina local como una lista de pares
List<Tuple2<String, Integer>> output = counts.collect();

// Imprime los resultados (palabra: cantidad de apariciones)
for (Tuple2<?, ?> tuple : output) {
    System.out.println(tuple._1() + ": " + tuple._2());
}

// Cierra la sesión de Spark
spark.stop();
}
}

```

Cómo correr el laboratorio

Primero descargar Apache Spark ([spark-3.5.1-bin-hadoop 3.tgz](https://spark.apache.org/downloads.html) - ~300MB) del sitio <https://spark.apache.org/downloads.html> y descomprimir el archivo descargado en algún directorio de su disco (\${SPARK_FOLDER}).

Hay dos formas de ejecutar el programa: modo local (como un programa Java común) o modo distribuido con Spark en un cluster.

Ejecutar Spark en modo local

Es la forma más simple de correr experimentos. Correr en **modo local** en Apache Spark significa ejecutar tu programa Spark en una sola máquina, sin necesidad de un clúster distribuido. Tu código sigue usando la API de Spark (RDDs, DataFrames, etc.), pero:

- **Todos los procesos (driver y workers)** corren en **tu computadora**, como si fuera un mini-clúster de un solo nodo.
- **No necesitas levantar manualmente un master o worker.**
- Spark internamente simula la ejecución distribuida usando múltiples hilos o procesos.

Se compila y ejecuta como un proyecto de Java común, incluyendo los binarios de la librería de Spark. Este es un posible Makefile que pueden usar, adaptando los paths.

```
Shell
# Comment this line to run with default java
JAVA_PATH=/opt/homebrew/opt/openjdk@11/bin/

JAVAC=javac
JAVA=java
LIB_DIR=lib
# Directory where spark is installed
SPARK_FOLDER=/usr/local/spark-3.5.1-bin-hadoop3
# Extra libraries to include, replace with your own .jars
CLASSPATH=$(OUT_DIR):$(LIB_DIR)/json-20231013.jar:$(SPARK_FOLDER)/jars/*

SOURCES=$(shell find src -name "*.java")

all: build run

build:
    $(JAVA_PATH)$(JAVAC) -cp $(CLASSPATH) -d out $(SOURCES)

run:
    $(JAVA_PATH)$(JAVA) -cp $(CLASSPATH) SparkFeedFetcher

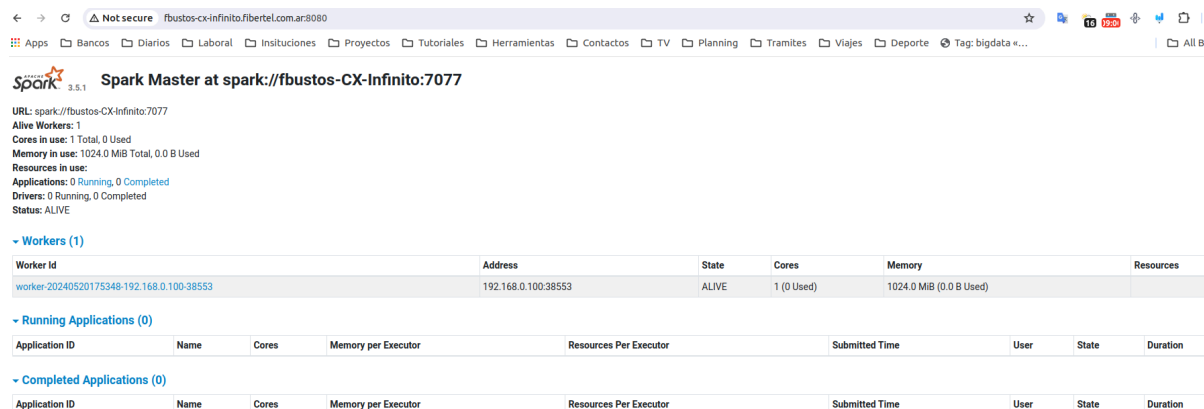
clean:
    rm -rf $(OUT_DIR)
```

Montar un cluster de Spark

Correr un programa Spark en un clúster significa ejecutarlo distribuido en múltiples máquinas. Spark se conecta al gestor del clúster (como YARN, Kubernetes, o Spark Standalone). El procesamiento se distribuye automáticamente entre los nodos del clúster. Se puede simular un cluster aun dentro de una única computadora:

1. Dentro del directorio `${SPARK_FOLDER}/sbin`, lanzar una instancia de master:
`./start-master.sh`
2. Dentro del directorio `${SPARK_FOLDER}/sbin`, lanzar una instancia de worker:
`./start-worker.sh spark://localhost:7077 -m 1G -c 1`

Con los dos scripts de shell anteriores, hemos creado un cluster de Spark con un master y un worker. Si está todo ok en la url <http://localhost:8080> podemos observar el estado actual del cluster:



Spark Master at spark://fbustos-CX-Infinito:7077

URL: spark://fbustos-CX-Infinito:7077

Alive Workers: 1

Cores in use: 1 Total, 0 Used

Memory in use: 1024.0 MiB Total, 0.0 B Used

Resources in use:

Applications: 0 Running, 0 Completed

Drivers: 0 Running, 0 Completed

Status: ALIVE

Workers (1)

| Worker Id | Address | State | Cores | Memory | Resources |
|---|---------------------|-------|------------|-------------------------|-----------|
| worker-20240520175348-192.168.0.100-38553 | 192.168.0.100:38553 | ALIVE | 1 (0 Used) | 1024.0 MiB (0.0 B Used) | |

Running Applications (0)

| Application ID | Name | Cores | Memory per Executor | Resources Per Executor | Submitted Time | User | State | Duration |
|----------------|------|-------|---------------------|------------------------|----------------|------|-------|----------|
|----------------|------|-------|---------------------|------------------------|----------------|------|-------|----------|

Completed Applications (0)

| Application ID | Name | Cores | Memory per Executor | Resources Per Executor | Submitted Time | User | State | Duration |
|----------------|------|-------|---------------------|------------------------|----------------|------|-------|----------|
|----------------|------|-------|---------------------|------------------------|----------------|------|-------|----------|

Ahora estamos en condiciones de mandarle un trabajo a dicho cluster. Les queda como ejercicio investigar cómo realizar esta tarea.

Entregables

Deben entregar el código del laboratorio completo a través de su repositorio de GitHub. Pueden incluir modificaciones al código que corresponde al lab2, si lo consideran necesario.

Deben entregar un archivo Makefile que permita ejecutar el programa con un único comando.

Agregar un archivo [README.md](#) con la siguiente estructura, donde tienen que responder las preguntas listadas, en base a la solución final que implementaron:

None

Título

Configuración del entorno y ejecución

Instrucciones para el usuario sobre cómo correr las dos partes del laboratorio con spark. Explicación del resultado que se espera luego de ejecutar cada parte.

Decisiones de diseño

Opcional. Cualquier cosa que quieran aclarar sobre la implementación del laboratorio

Conceptos importantes

1. ****Describa el flujo de la aplicación**** ¿Qué pasos sigue la aplicación desde la lectura del archivo feeds.json hasta la obtención de las entidades nombradas? ¿Cómo se reparten las tareas entre los distintos componentes del programa?

2. ****¿Por qué se decide usar Apache Spark para este proyecto?**** ¿Qué necesidad concreta del problema resuelve?

3. ****Liste las principales ventajas y desventajas que encontró al utilizar Spark.****

4. ****¿Cómo se aplica el concepto de inversión de control en este laboratorio?**** Explique cómo y dónde se delega el control del flujo de ejecución. ¿Qué componentes deja de controlar el desarrollador directamente?

5. ****¿Considera que Spark requiere que el código original tenga una integración tight vs loose coupling?****

6. ****¿El uso de Spark afectó la estructura de su código original?**** ¿Tuvieron que modificar significativamente clases, métodos o lógica de ejecución del laboratorio 2?

Recomendaciones

- Leer/ver un tutorial de Java antes de comenzar (si no lo hicieron para el lab2).
- Investigar sobre la arquitectura de Spark.
- Investigar la API de java de Spark (<https://spark.apache.org/docs/latest/quick-start.html>)

- Evitar errores de serialización con Spark al tomar como argumento o retornar objetos complejos. Por ejemplo, creen las instancias de los parsers dentro de la función paralelizada.

Puntos Extras

1. Agregar un parámetro para solo mostrar las primeras N entidades nombradas más frecuentes.
2. Hacer una comparativa entre la velocidad de respuesta de la versión no distribuida (lab2) vs la versión distribuida (lab3) de su aplicación.
3. Evaluar que otra parte de las tareas que realiza su aplicación se puede realizar en forma distribuida y luego implementarlo.
4. Proponer e implementar un nuevo diseño para el proyecto simplificando y limpiando los componentes. Un ejemplo es pasar todos los valores hardcoded como parámetros al ejecutable.

Ejemplo: Consola de Spark

Spark también puede ejecutarse mediante un shell interactivo (como ghci). Primero debemos lanzar dicho shell desde la carpeta `${SPARK_FOLDER}/bin`:

```
./spark-shell.sh spark://localhost:7077
```

Luego, leemos un archivo de texto "book.txt" y computamos por ejemplo su cantidad líneas

```
scala> val dataset = spark.read.textFile("/Users/ada/famaf/paradigmas25/el-manifiesto-comunista.txt")
```

```
dataset: org.apache.spark.sql.Dataset[String] = [value: string]
```

```
scala> dataset.count()
```

```
res0: Long = 439
```

```
scala> val dataset2 = dataset.filter(line => line.contains("Marx"))
```

```
dataset2: org.apache.spark.sql.Dataset[String] = [value: string]
```

```
scala> dataset2.count()
```

```
res1: Long = 1
```

```
scala> val dataset2 = dataset.filter(line => line.contains("proletario"))
```

```
dataset2: org.apache.spark.sql.Dataset[String] = [value: string]
```

```
scala> dataset2.count()
```

```
res2: Long = 18
```

También, podemos computar la cantidad de líneas en donde ocurre la cadena "Marx" o "proletario":

```
scala> val dataset2 = dataset.filter(line => line.contains("Marx"))
```

```
dataset2: org.apache.spark.sql.Dataset[String] = [value: string]
```

```
scala> dataset2.count()
```

```
res1: Long = 1
```

```
scala> val dataset2 = dataset.filter(line => line.contains("proletario"))
```

```
dataset2: org.apache.spark.sql.Dataset[String] = [value: string]
```

```
scala> dataset2.count()
```

```
res2: Long = 18
```

Posibles problemas

Al instalar spark usando el link de la cátedra, funciona solo con java 11, pero tengo java 24

Una opción es instalar java 11 y sobrescribir el binario de java solo para spark, de esta manera, la versión por defecto sigue siendo la más reciente. Luego se sobrescribe el JAVA_PATH para spark únicamente (versión para mac):

```
JAVA_HOME=/opt/homebrew/opt/openjdk@11 ./sbin/start-master.sh
```

En el enunciado hay un Makefile de ejemplo que pueden usar para compilar el laboratorio 3 con java 11

Corro alguno de los comandos como [start-master.sh](#) o `start_worker.sh` pero no funciona

En la consola se imprime o el error, o el archivo donde se guardan los logs. Leer el archivo de logs para saber cuál es el problema.

El master y el worker se inicializan correctamente en los puertos 8080 y 8081, pero el master no encuentra el worker



Spark Master at spark://Milagros-MacBook-Pro.local:7077

URL: spark://Milagros-MacBook-Pro.local:7077

Alive Workers: 0

Cores in use: 0 Total, 0 Used

Memory in use: 0.0 B Total, 0.0 B Used

Resources in use:

Applications: 0 [Running](#), 0 [Completed](#)

Drivers: 0 Running, 0 Completed

Status: ALIVE

▼ Workers (0)

| Worker Id | Address | State | Cores | Me |
|-----------|---------|-------|-------|----|
|-----------|---------|-------|-------|----|

▼ Running Applications (0)

| Application ID | Name | Cores | Memory per Executor | Resources Per Executor |
|----------------|------|-------|---------------------|------------------------|
|----------------|------|-------|---------------------|------------------------|

▼ Completed Applications (0)

| Application ID | Name | Cores | Memory per Executor | Resources Per Executor |
|----------------|------|-------|---------------------|------------------------|
|----------------|------|-------|---------------------|------------------------|

Spark, por defecto, toma el hostname del sistema (por ejemplo:

Milagros-MacBook-Pro.local). Entonces la URL real del máster podría ser

[spark://Milagros-MacBook-Pro.local:7077](#) en lugar de `localhost:7077`

Para asegurarte, revisa el archivo de logs del proceso master y buscá una línea como

`Starting Spark master at <url del master>:7077`

Usá esa URL cuando inicies el worker. Recordá terminar el proceso primero con

`stop-worker.sh`