Informe Individual - Emanuel Nicolás Herrador

En el presente informe se detallará cómo fue el desarrollo de la implementación del *lector de feeds* implementado en el Laboratorio 2 utilizando el Framework de Spark para cálculo distribuido.

La forma de realizarlo fue **sin** uso de ayuda por inteligencia artificial (sea ChatGPT, WriteSonic, Copilot, entre otros), solo con consultas a *StackOverflow* y a la *documentación oficial de Spark*.

Para ello, veamos todos los puntos a considerar en el desarrollo planteados por el enunciado del proyecto:

Índice de puntos considerados

¿Qué herramientas se utilizaron?

En el presente laboratorio se utilizaron como herramientas *fundamentales* Spark (porque es el objetivo del proyecto), Maven (para tener un buen manejo de las dependencias a utilizar, tanto de Spark como de JSON) e IntelliJ IDEA (en base a recomendación de mi compañero Guille y de los pros que tiene respecto a VSCode, dado que es una herramienta creada para desarrollo puro de Java).

¿Cómo instalar las herramientas usadas?

Spark

Se ha instalado el framework de Spark siguiendo las instrucciones dadas por:

- Documentación de descarga de Spark
- Ejemplo de instalación de Spark y su contexto

Por ello, los pasos que se siguieron fueron los siguientes:

1. Ir a https://spark.apache.org/downloads.html y descargar la versión `3.4.0 (Apr 13 2023)` con el tipo de paquete `Pre-built for Apache Hadoop 3.3 and later`
    1. Se descarga en https://dlcdn.apache.org/spark/spark-3.4.0/spark-3.4.0-bin-hadoop3.tgz
2. Moverlo de la carpeta `~/Descargas` al directorio en donde se está trabajando (en este caso `~/`)
3. Extraer el `.tar` y eliminarlo
4. Cambiar el nombre de la carpeta por `spark`
5. Setear las variables de entorno para Spark en el archivo `.bashrc`

    ```
    # SPARK
    export SPARK_HOME='/home/emanuel-nicolas-herrador/spark'
    export PATH=$PATH:$SPARK_HOME/bin
    ```

6. Recargar el archivo de configuración

    ```
    $ source ~/.bashrc
    ```

Para verificar la instalación y que todo ande de forma correcta, se hicieron pruebas de ejecución de `.jar` como de la shell de spark:

```
$ spark-submit --class org.apache.spark.examples.SparkPi
spark/examples/jars/spark-examples_2.12-3.4.0.jar 10

$ run-example SparkPi 10

$ spark-shell --master local[2]
# `--master` especifica la master URL para un cluster distribuido. Sin
embargo, si la flag es `local[N]`, se usa para correr de forma local N
hilos (threads).
# Más información se puede ver en
https://spark.apache.org/docs/latest/submitting-applications.html#master-
urls
```

Maven

En este punto, en mi computadora de forma local estuve probando utilizar maven instalándolo con

```
$ sudo apt-get install maven
```

pero no funcionaba y se generaban muchísimos errores al querer usar maven para crear un proyecto nuevo y/o compilar un programa de java para crear el archivo `.jar` correspondiente.

Por ello mismo, estuve investigando y los pasos que seguí fueron de la documentación oficial de Maven:

1. Descargar `apache-maven-3.9.2-bin.tar.gz` desde la página de descargas
2. Extraerlo al archivo y mover la carpeta resultante al directorio `~`
3. Cambio el nombre de la carpeta a, simplemente, `maven`
4. Se agrega la variable de entorno al archivo `.bashrc` del siguiente modo:

```
# APACHE MAVEN
export MAVEN_HOME='/home/emanuel-nicolas-herrador/maven'
export PATH=$PATH:$MAVEN_HOME/bin
```

5. Recargar el archivo de configuración

```
$ source ~/.bashrc
```

Se comprueba la instalación mediante:

```
$ mvn -v
Apache Maven 3.9.2 (c9616018c7a021c1c39be70fb2843d6f5f9b8a1c)
Maven home: /home/emanuel-nicolas-herrador/maven
Java version: 17.0.7, vendor: Private Build, runtime: /usr/lib/jvm/java-
17-openjdk-amd64
Default locale: es_AR, platform encoding: UTF-8
OS name: "linux", version: "5.15.0-72-generic", arch: "amd64", family:
"unix"
```

IntelliJ IDEA

La instalación de este IDE fue **extremadamente sencilla**. Simplemente se tuvo que instalar desde *snap* con el comando:

```
sudo snap install intellij-idea-community --classic
```

¿Cómo debe ser un proyecto de Spark con Java y cómo se corre?

Para presentar la estructura que debe tener un programa y un proyecto realizado en Spark, se va a utilizar el ejemplo de java presente en la documentación.

Para ello, notemos que un programa en Spark se ejecuta desde un `.jar` con el comando `spark-submit --class "nombreClase" --master local[N] target/"nombreDelJar"`, por lo que se debe usar compilar la aplicación JAR con, por ejemplo, *Maven*.

Por ello, entonces, hay solo dos cuestiones *fundamentales* a tener en cuenta para el proyecto:

- Uso de Maven para manejo de dependencias, compilación y ejecución (es recomendado por la documentación de Spark y dado que estamos usando un IDE que tiene lo tiene integrado)
  - Requiere el uso de un archivo `pom.xml` en donde se listen las configuraciones de compilación y dependencias a utilizar. En este caso, como es un proyecto de Spark, se considera:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>sparkTest</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <!-- SPARK dependency -->
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.12</artifactId>
            <version>3.4.0</version>
        </dependency>
    </dependencies>
```

```
</project>
```

- Estructura usual de proyectos Java

```
./pom.xml
./src
./src/main
./src/main/java
./src/main/java/SimpleApp.java
```

A diferencia del ejemplo de la documentación, el programa `SimpleApp.java` del que se hace uso contiene una leve modificación dado que, dada la integración con las librerías SQL de Spark, se consideraba error la ambigüedad del tipo de la función `filter(s -> s.contains("a"))`. Por ello, se procedió a cambiarla por `filter((String s) -> s.contains("a"))`, quedando:

```
/* SimpleApp.java */
import org.apache.spark.sql.SparkSession;
import org.apache.spark.sql.Dataset;

public class SimpleApp {
    public static void main(String[] args) {
        String logFile = "/home/emanuel-nicolas-herrador/spark/README.md";
        SparkSession spark = SparkSession.builder().appName("Simple
Application").getOrCreate();
        Dataset<String> logData = spark.read().textFile(logFile).cache();

        long numAs = logData.filter((String s) ->
s.contains("a")).count();
        long numBs = logData.filter((String s) ->
s.contains("b")).count();

        System.out.println("Lines with a: " + numAs + ", lines with b: " +
numBs);

        spark.stop();
    }
}
```

Finalmente, entonces, para poder compilar y correr este proyecto, se debe hacer lo siguiente:

- Empaquetar la aplicación usando Maven

```
$ mvn package
```

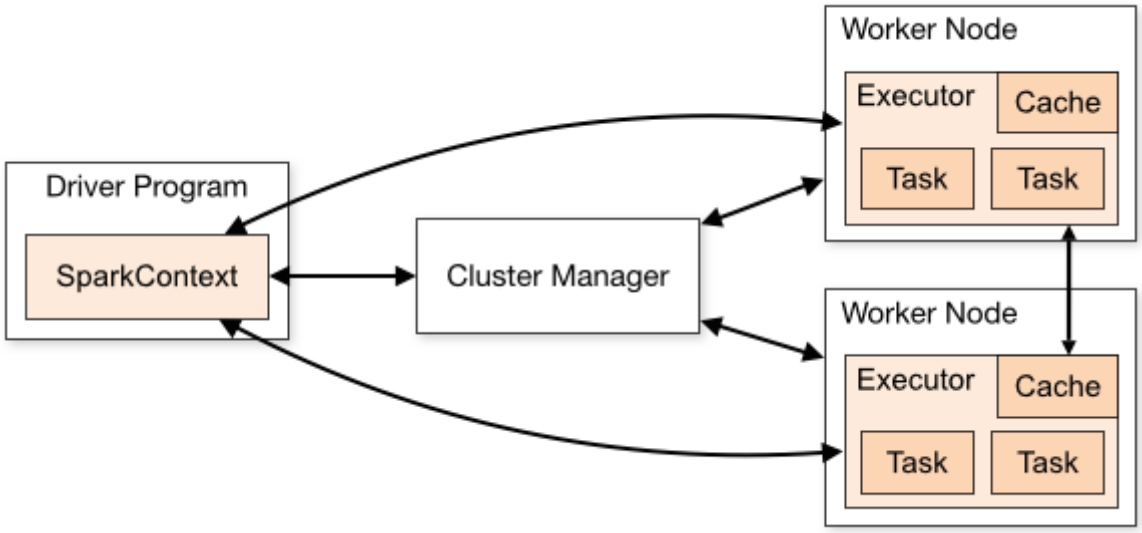
- Correr la aplicación con `spark-submit`

```
$ spark-submit --class "SimpleApp" --master local[4] target/sparkTest-
1.0-SNAPSHOT.jar
```

Respecto a mi caso en particular, al estar usando Intellij IDEA, se siguieron las recomendaciones dadas por la documentación de developer-tools.

¿Qué estructura tiene un programa en Spark?

Dado que la idea del framework Spark es cálculo distribuido, la estructura del sistema distribuido es:



La idea de los programas es crear aplicaciones o tareas para que puedan ser ejecutadas por varios nodos en paralelo de forma "independiente" entre sí. La idea es poder paralelizar el proceso de ejecución del programa principal dividiendo *tareas* entre distintos nodos de procesamiento y luego mergeando toda la información obtenida.

Por ello mismo, en este caso que se usa Java, lo que se requiere como estructura del programa (y que se puede ver en el ejemplo de la sección anterior) es lo siguiente:

1. **Creación de una sesión de Spark**: dentro del programa en particular, se debe usar la clase `SparkSession`, la cual es la interfaz principal para interactuar con Spark y proporciona acceso a las funcionalidades principales (Como DataFrame y SQL). En particular, en el ejemplo anterior, lo que se hace es crear desde la sesión una aplicación que ejecute haga el procesamiento del archivo `README.md`. Aquí, se debe configurar la sesión con las propiedades y opciones adecuadas (cantidad de nodos, de recursos asignados a cada no, etc...)

2. **Lectura de datos**: se especifica la fuente de datos que se va a usar en el programa (en este caso, el README). Pueden ser muchos tipos de datos, sean archivos locales, bases de datos, etc, etc...
3. **Transformaciones y operaciones**: una vez que los datos se cargaron en un DataFrame o un RDD, podemos aplicar distintas transformaciones y operaciones para manipular y procesar los datos, sea de filtrado, mapeo, ordenamiento, etc (documentación de transformaciones de RDD (resilient distributed dataset)). En Spark estas transformaciones, tal y como en muchos lenguajes declarativos, son operaciones lazy que se van almacenando como un plan de ejecución (en algún momento se ejecutarán pero no de inmediato)
4. **Acciones con los datos**: después de las transformaciones, se pueden aplicar acciones para activar la ejecución del plan definido. Las acciones son operaciones que desencadenan la computación de los datos y generan resultados o realizan acciones secundarias como puede ser imprimir en la consola o guardar los resultados en un archivo en otro directorio (documentación de acciones de RDD).
5. **Cierre de la sesión de Spark**: una vez que se han hecho todas las operaciones necesarias, se cierra la sesión para liberar los recursos utilizados y finalizar el programa.

Este es un esquema general de cómo es la estructura de los programas de Spark, más que nada, relacionados a la tarea que debemos realizar en este proyecto.

¿Qué estructura tiene un programa de conteo de palabras en diferentes documentos en Spark?

Un programa de conteo de palabras de **diferentes** documentos es, básicamente, una generalización de este ejemplo brindado por Spark.
Por ello mismo, entonces, el programa sería el siguiente:

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;
import org.jetbrains.annotations.NotNull;
import scala.Tuple2;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public final class SimpleApp {
    public static void main(String @NotNull [] args) {
        if(args.length == 0) {
```

```
            System.out.println("Usage: JavaWordCount <file>");
            System.exit(1);
        }

        // Configuración de Spark
        SparkConf sparkConf = new SparkConf()
                .setAppName("JavaWordCount")
                .setMaster("local[*]");

        JavaSparkContext spark = new JavaSparkContext(sparkConf);

        // Paralelizar los paths a los archivos como una lista de strings
        JavaRDD<String> fileRDD = spark.parallelize(Arrays.asList(args));

        // Leer archivos, separar el contenido en palabras y contar las
ocurrencias
        JavaPairRDD<String, Integer> output = fileRDD
                .flatMap(file -> {
                    // Leer el contenido de cada archivo y separarlo en
líneas
                    try {
                        BufferedReader reader = new BufferedReader(new
FileReader(file));

                        List<String> linesRead = new ArrayList<>();
                        String nwLine;
                        while((nwLine = reader.readLine()) != null)
linesRead.add(nwLine);
                        reader.close();

                        return linesRead.iterator();
                    } catch (IOException e) {
                        e.printStackTrace();
                        return Collections.emptyIterator();
                    }
                })
                // Separar las líneas en palabras
                .flatMap(line -> Arrays.asList(line.split("
")).iterator())
                // Contar las ocurrencias de las palabras
                .mapToPair(word -> new Tuple2<>(word, 1))
                .reduceByKey(Integer::sum);

        // Mostrar por pantalla los resultados obtenidos
        output.foreach(wordCount -> System.out.println(wordCount._1() + ":
" + wordCount._2()));

        spark.close();
```

```
    }
}
```

En este caso, a diferencia del ejemplo provisto por Spark para un solo archivo, se decidió usar el `BufferedReader` para paralelizar la lectura de los diferentes archivos.

¿Cómo adaptar el código del Laboratorio 2 a la estructura del programa objetivo en Spark?

Primera aproximación

En la adaptación del código del Laboratorio 2, se consideró (obviamente) que el driver se crea y reside en el `Main.java` mientras va enviando tareas a los worker nodes. Si bien en el código está bien explicada cada parte con comentarios, aquí va a mencionarse nuevamente.

El esquema que se consideró fue (**haciendo el PUNTO EXTRA de integrar en estructura map-reduce la obtención de feeds**):

- Validar los argumentos (en caso que no sean válidos, imprimo una ayuda y termino)
- Hacer las configuraciones para usar Spark (SparkConf y JavaSparkContext)
- Creo una Subscription que parsea el archivo JSON dividiendo cada JSONObject para las SimpleSubscription (todavía no se parsearon)
- Paralelizo las listas de subscripciones para hacer las siguientes acciones de forma concurrente cada una
    - Separar las subscripciones por sus parámetros
    - Para cada combinación de subscripción - parámetro, se parsea el JSONObject, se hace la HTTP Request y se obtiene el Feed
        - Se manejan los casos de error devolviendo tuplas en el `flatMap`. Se considera `(feed, error)` donde uno es, sí o sí, `null`. Esto es para dividir los casos
- Se filtra de la lista de tuplas por un lado los feeds y por otro los errores
- Ahora hay dos casos dependiendo del parámetro que se haya pasado para ejecutar el programa
    - Si tengo que imprimir normal, para cada feed hago `prettyPrint`
    - Si tengo que hacer las heurísticas, lo que se realiza es:
        - Crear el objeto de la heurística
        - Se obtiene la lista de artículos para cada feed
        - Se computan las namedEntities para cada feed y se obtiene la lista de estas para todos los artículos de todos los feeds
        - Se muestran por pantalla las entidades nombradas
- Se imprimen los errores en caso que haya habido

Esto se realizó de forma sencilla teniendo en cuenta que:

- Para realizar concurrencia, los elementos que se usen deben implementar la clase `Serializable`, lo cual implica que se puede convertir en una secuencia de bytes que pueden ser leídos posteriormente para restaurar el objeto original.
  - Esto se usa para guardar el estado del objeto y poder transmitirlo fácilmente (cuando se comparte a los nodos como es el caso de las clases de las heurísticas, simpleSubcriptions, Feed, por ejemplo)
- La concurrencia se hace en la parte de las transformaciones de los objetos RDD (o iterables) que se utilizan en el programa
  - Esto implica que se hace fuerte uso de `flatMap` y de `filter` (este solo para separar entre feed y errores)
  - Hay que tener en cuenta que como esto se aplica en cada nodo de forma aparte y Spark funciona para que, si uno tuvo un error o se cayó, otro pueda hacer la tarea; entonces jamás se puede tener una excepción sin catchear dentro de la función
    - Por ello, siempre devuelven algo. En el caso particular en el que se usan, es para crear la lista de errores a mostrar al final del programa
    - Esto permite que las transformaciones se hagan de forma lazy hasta tanto se invoque a una acción
- Dada la naturaleza del feedReader, las únicas acciones que se hacen con los objetos de Spark son: mostrar por pantalla (con un `foreach`) y contar la cantidad (para saber si imprimo o no errores, con un `count`).

Implementación final

Para poder seguir paralelizando más en otros objetos y clases, requerimos tener alguna forma de "pasarles" a estos el "manejador" de Spark. Por ello mismo, para poder realizar esto, usé la herramienta **SparkSession** que habilita esta opción y está disponible en versiones más nuevas de Spark.

Por ello mismo, teniendo en cuenta esto y pasando la sesión a los diferentes objetos, se ha realizado también lo siguiente (se suma a lo hecho en la 1era aproximación):

- **Subscriptions.java**: se agrega la paralelización del parseo de los JSONObjects, creación de las SimpleSubscriptions y su instanciación
- **Compute Named Entities**: se probó agregar la paralelización para el conteo de las entidades nombradas pero dado que tardaba entre 2 o 3 veces más que la implementación anterior, decidí no considerar esta opción

¿Cómo se integra una estructura ordenada a objetos con la estructura funcional de map-reduce?

Para realizar la combinación de estos *dos mundos*, tenemos que estructurar la idea de cómo se va a plantear el código para maximizar lo que nos brinda cada parte:

- Procesamiento en paralelo (map-reduce) dividiendo datos en fragmentos más pequeños
- Encapsulamiento y secuenciación para preservar relación y orden entre objetos durante el procesamiento

Por ello mismo, entonces, queremos combinar estos dos conceptos: el de programación funcional de Map-Reduce con el diseño orientado a objetos, para mantener la estructura ordenada mientras se aprovecha el procesamiento en paralelo y la escalabilidad de Map-Reduce.

Como consecuencia, lo que se tiene que tener en cuenta es:

- ¿Cómo debe ser el orden y qué se va a usar para ello?
  - Determina qué atributo o conjunto de estos se va a usar para establecer el orden de los objetos
- Fase de mapeo
  - Cada objeto se asigna a una clave de ordenación utilizando la función de map. La clave de ordenación y el objeto se emiten como pares clave-valor
  - En este caso, la clave sería el valor del atributo de ordenación y el valor sería el objeto en sí
- Fase de ordenación
  - Los pares clave-valor generados en la fase de mapeo se ordenan según la clave de ordenación (garantiza que los objetos se agrupen de acuerdo con su orden definido)
- Fase de reducción
  - Los objetos se procesan en función de su orden

En esta primera parte se pueden observar todos estos conceptos más que nada en la sección del *wordCounter* de diferentes archivos. Respecto a dónde se vería reflejado fielmente en el proyecto actual, sería en la parte grupal cuando se ordenen los artículos en base al criterio que se establece desde un principio (las palabras que se seleccionaron para buscar).

- En los dos casos, la función de ordenación es la suma de las ocurrencias. En uno de las palabras y en el otro de las entidades seleccionadas