Laboratorio 3
“Computación Distribuida con Apache Spark”

Paradigmas 2024 - FAMAF

Abstract

En este laboratorio vamos a trabajar sobre el laboratorio 2 ya implementado por ustedes. La idea es extender su lector de feed automático para que soporte “big data” (una gran cantidad de datos, o sea, varios gigas de datos) en el cálculo de entidades nombradas. Para ello, utilizaremos el framework “Apache Spark” *por ser uno de los framework de big data y computación distribuida más utilizado del momento. Por último, cabe resaltar que este laboratorio es más un trabajo de investigación que de desarrollo, es decir, solo tendrán que codear muy muy muy pocas líneas en comparación con los laboratorios anteriores.*

Introducción

En el archivo “feeds.json” del laboratorio 2, el usuario configura los feeds a consumir. Es decir, podemos ver a este archivo como el lugar de nuestra aplicación donde se define un conjunto de feeds `feed_1, … , feed_n` a obtener, sobre el cual uno luego computa las entidades nombras que ocurren en el título y descripción de cada artículo de cada feed. Pensemos ahora qué sucede si este conjunto escala notoriamente, es decir, llevado al extremo, si $n \to \infty$.

Esto nos generaría una gran cantidad de texto (gigas y gigas de texto) sobre la cual después hay que computar las entidades nombradas que ocurren en el. Claramente, si computamos secuencialmente las entidades sobre gigas y gigas de texto, tardaríamos mucho. Surge la pregunta de cómo podemos devolver las entidades nombradas de una gran cantidad texto (big data) en un tiempo de respuesta razonable para el usuario? *La solución más utilizada es distribuir (repartir) los datos en diferentes computadoras y cada una de ellas realice un parte del cómputo sobre la partición de los datos que le fue*

asignada, es decir, computar en paralelo con k-computadoras la solución del problema. Actualmente, existen diferentes frameworks que facilitan al programador realizar cómputos en forma distribuida sobre una dataset de gran tamaño. Uno de ellos es “Apache Spark” y nuestra idea es que ustedes hagan una pequeña experiencia con el mismo. Lo que le pedimos es muy simple y concreto conceptualmente: que computen las entidades nombradas de un gran archivo de texto en forma distribuida mediante Spark. Para ello, deberán primero leer documentación sobre spark (su arquitectura) y su API para java para poder ser llamado desde su aplicación.

Sobre la arquitectura de Apache Spark

Este framework trabaja con una arquitectura Master-Slaves: tiene una única computadora “MASTER” y k-computadoras “SLAVES”. Esto configura lo que se denomina un cluster de Spark y funciona de la siguiente manera: el usuario envía un petición de cómputo al master y este se encarga de dividir el big data entre los slaves y estos últimos computan sobre su partición asignada. Finalmente, cuando todos los slaves finalizan, el master se encarga de recolectar las soluciones parciales de cada slave, para devolverle al usuario la solución completa (claramente vemos una estrategia divide y vencerás). Todo esto es transparente para el usuario, que puede pensar que del otro lado hay solamente una super-máquina de cómputo haciendo todo el trabajo. Cabe señalar, que por razones obvias, no contamos con k+1 computadoras distintas para realizar el laboratorio. Por lo tanto, el master y los slaves serán implementados en nuestra propia máquina en forma local. En particular, vamos a montar un cluster de Spark con 2 slaves (también llamados workers).

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

Montar un Cluster de Spark

Descargar Apache Spark (spark-3.5.1-bin-hadoop3.tgz) del sitio https://spark.apache.org/downloads.html y descomprimir el archivo descargado en algún directorio de su disco (${SPARK_FOLDER}).

Dentro del directorio ${SPARK_FOLDER}/sbin, lanzar una instancia de master:

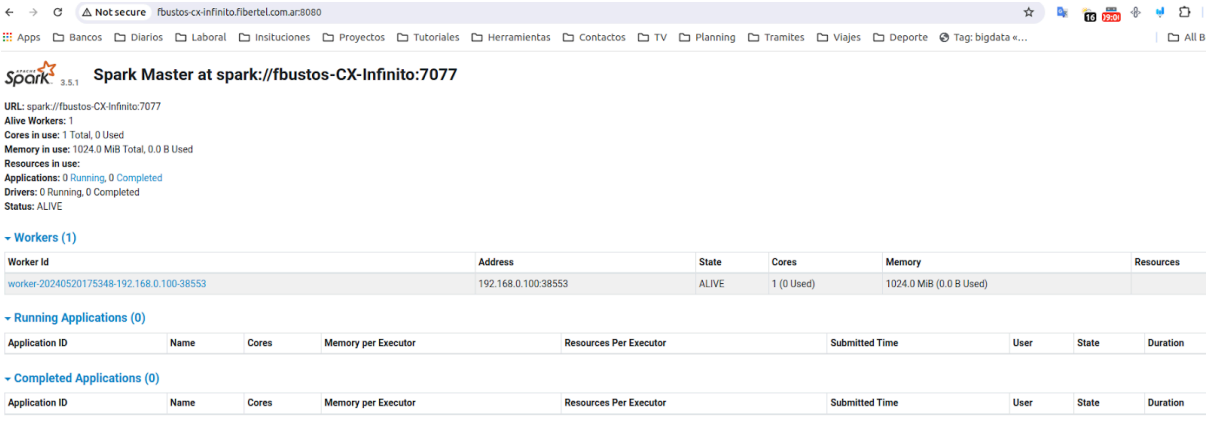
```
./start-master.sh
```

Dentro del directorio ${SPARK_FOLDER}/sbin, lanzar una instancia de worker:

```
./start-worker.sh spark://localhot: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:



Ahora estamos en condiciones de mandarle un trabajo a dicho cluster, interactuando con Spark mediante su shell. 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("/home/fbustos/Documents/cursos/famaf/paradigmas24/lab3/b
ook.txt")
scala>dataset.count()
res0: Long = 22836
```

También, podemos computar la cantidad de líneas en donde ocurre la cadena “Moby”:

```
scala> val dataset2 = dataset.filter(line => line.contains("Moby"))
scala>dataset2.count()
dataset2: Long = 83
```

Ahora en lugar de interactuar con spark desde su shell, queremos hacerlo desde nuestra aplicación mediante la api que spark ofrece para ello.

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 (notar que esto es muy parecido a los que ustedes deben computar, salvo que en lugar de contar palabras que ocurren el texto, deben contar entidades nombradas):

```
package org.apache.spark.examples;
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 JavaWordCount {
  private static final Pattern SPACE = Pattern.compile(" ");
  public static void main(String[] args) throws Exception {
    if (args.length < 1) {
      System.err.println("Usage: JavaWordCount <file>");
      System.exit(1);
    }
    SparkSession spark = SparkSession
      .builder()
```

```
            .appName("JavaWordCount")
            .getOrCreate();
          JavaRDD<String> lines =
spark.read().textFile(args[0]).javaRDD();
          JavaRDD<String> words = lines.flatMap(s ->
Arrays.asList(SPACE.split(s)).iterator());
          JavaPairRDD<String, Integer> ones = words.mapToPair(s ->
new Tuple2<>(s, 1));
          JavaPairRDD<String, Integer> counts =
ones.reduceByKey((i1, i2) -> i1 + i2);
          List<Tuple2<String, Integer>> output = counts.collect();
          for (Tuple2<?,?> tuple : output) {
            System.out.println(tuple._1() + ": " + tuple._2());
          }
          spark.stop();
        }
      }
```

En el directorio ”${SPARK_FOLDER}/examples/src/main/java/org/apache/spark/examples/” se encuentran varios ejemplos de spark en java que pueden ser útiles para sacar ideas!!!

Pipeline de Trabajo Sugerido

Crear la big data escribiendo en un archivo los artículos (title + description) de todos los feeds.

Montar un Cluster de Spark con un master y dos workers en forma local.

Computar Entidades Nombradas utilizando la api de Spark para java.

Imprimir por Pantalla las entidades nombradas que ocurren en el big data.

Recomendaciones

Leer un tutorial o libro de Java antes de comenzar (si no lo hicieron para el lab2).

Leer sobre la arquitectura de Spark.

Leer sobre la API de java de Spark (https://spark.apache.org/docs/latest/quick-start.html)

Puntos Extras

Hacer una comparativa entre la velocidad de respuesta de la versión no distribuida (lab2) vs la versión distribuida (lab3) de su aplicación.
Evaluar que otra parte de las tareas que realiza su aplicación se puede realizar en forma distribuida y luego implementarlo.

Entrega y Defensa

Fecha límite de entrega: Viernes 21 de Junio de 2024. Entrega del código fuente + informe mediante push en bitbucket en el repositorio del grupo. Día y horario de la defensa a coordinar con el docente asignado a su grupo.