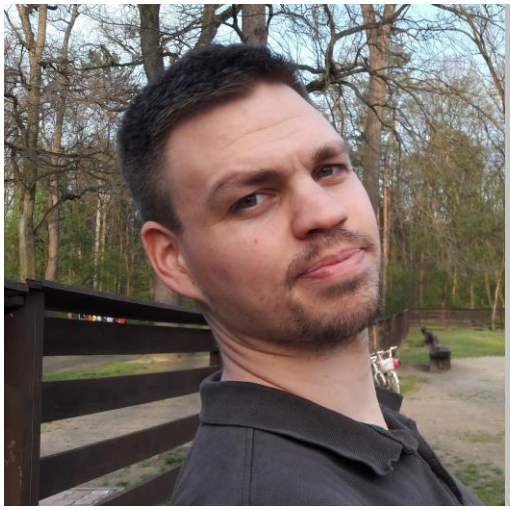



Dr. Hegedűs Péter, Dr. Ferenc Rudolf

Nagyméretű adatbázisok

Jelen tananyag a Szegedi Tudományegyetemen készült az Európai Unió támogatásával.

Projekt azonosító: EFOP-3.4.3-16-2016-00014

Szegedi Tudományegyetem
Cím: 6720 Szeged, Dugonics tér 13.
www.u-szeged.hu
www.szechenyi2020.hu

Apache Spark programozása Scala és Java nyelven

Összefoglalás

Ez az olvasólecke gyakorlati tudást nyújt a Spark keretrendszer használatához. Részletesen foglalkozunk a Spark Scala nyelvű interaktív parancsértelmező eszközével. Különböző példákon keresztül demonstráljuk a különböző adatforrásokból történő adat beolvasást, a Dataset/DataFrame és RDD API-k használatát. Több konkrét transzformációt és akciót is bemutatunk az adatokon. A parancssori kliens mellett betekintést nyújtunk abba, hogyan lehet olyan önálló Java alkalmazásokat fejleszteni, melyek átadhatók a Spark keretrendszernek mind végrehajtható feladat.
A lecke fejezetei:

- 1. fejezet: **Apache Spark indítása docker környezetben, Scala parancssori kliens használata (olvasó)**
- 2. fejezet: **A Word count és árfolyam átlag problmák megoldása Spark Scala parancssorban (olvasó)**
- 3. fejezet: **Önálló Spark alkalmazások programozása Java nyelven (olvasó)**

Téma típusa: **gyakorlati**
Olvasási idő: **55 perc**

📖 1. fejezet

Az Apache Spark telepítése/indítása docker környezetben

Ahogy a kapcsolódó előadás olvasóleckéiben (`8e_BigData-exec-engines-SPOC.md` és `9e_BigData-spark-rdd-df-SPOC.md`) már láttuk, az Apache Spark [1] egy villámgyors klaszter számítási keretrendszer, amit nagyon gyors adatfeldolgozásra terveztek. A Hadoop MapReduce modellen alapul (koncepcionálisan, nem kód szintjén), de olyan módon általánosítja és terjeszti ki azt, ami lehetővé teszi a hatékony felhasználását interaktív lekérdezések készítéséhez vagy stream feldolgozáshoz is.

Telepítés/docker konténerek Spark-hoz

A Spark telepítése a klaszter típusától függően eltérő lépésekből állhat. Az alapvető Spark disztribúció a megfelelő bináris csomag letöltéséből, kicsomagolásából, valamint a környezeti változók és konfigurációs állományok beállításából áll [2]. Ha Hadoop támogatást szeretnénk használni, egy megfelelő Hadoop klasztert is telepítenünk kell, majd a Spark beállításait módosítani eszerint. Ezt követően a klaszter típusának megfelelően (Standalone, Apache Mesos, Hadoop YARN, Kubernetes) az egyes node-okra telepíteni kell a Spark komponenseket (~~master~~ primary, worker, stb.). Részletek a hivatalos dokumentációban [3] olvashatók. Mi a lokális gépre történő telepítés, illetve saját fizikai klaszter összeállítása helyett egy előre előkészített docker container stack-et fogunk használni, ami tartalmazza a Hadoop klaszter elemeit is. A következőkben bemutatott stack az összes docker image-t elindítja, ami szükséges a Spark azonnali használatához. A következő példák tetszőleges gépen futtathatók, ahol telepítve van a Dokcer környezet, valamint a Git verziókövető kliens.

A Spark stack indításához először töltsük le a docker leírókat és a teljes stack konfigurációt tartalmazó git repository-t a következő parancs segítségével:

```
$ git clone https://github.com/big-data-europe/docker-hadoop-spark-workbench.git
```

A letöltött `docker-hadoop-spark-workbench` mappa két stack konfigurációt tartalmaz: `docker-compose.yml` és `docker-compose-hive.yml`. Mi az elsőt fogjuk használni, amely egy Hadoop klasztert hoz létre egy Spark ~~master~~ primary és egy worker node-dal, valamint néhány támogató eszközt tartalmazó container-rel. A stack egészen pontosan a következő container-eket indítja el:

- `namenode` - a Hadoop klaszter NameNode szervere
- `datanode` - egy darab Hadoop DataNode
- `spark-master` - a Spark primary szerver node-ja
- `spark-worker` - egy Spark worker gép (a számítások elvégzéséhez)
- `spark-notebook` - egy interaktív, web alapú kódszerkesztő Spark adatelemzések készítéséhez (vesd össze pl. Jupyter Notebook [4])
- `hue` - egy HDFS fájl böngésző alkalmazás fejlettebb funkciókkal, mint a beépített Hadoop browse utility

A `docker-compose-hive.yml` a fenti stack-hez még Apache Hive támogatást is ad, hiszen a Spark Hive táblákból is tud dolgozni. Ha ilyet szeretnénk használni, ezt a konfigurációt kell elindítanunk.

A stack indításához lépjünk be a `docker-hadoop-spark-workbench` mappába, és adjuk ki a következő docker parancsot:

```
$ docker-compose up -d
```

A stack sikeres indítása után a következő paranccsal ellenőrizhetjük, hogy minden container sikeresen elindult:

```
$ docker ps
```

Windows felhasználók figyelem!
Amennyiben Windows host-on indítjuk a docker stack-et, és a következő hibaüzenetet kapjuk "**ERROR: for namenode Cannot start service namenode: Ports are not available: listen tcp 0.0.0.0:50070: bind: An attempt was made to access a socket in a way forbidden by its access permissions.**", az azért van, mert a dokcer daemon bizonyos portokat lefoglal magának, amivel ütközik a docker konfigurációnk. A legegyszerűbb megoldás, ha módosítjuk a *docker-compose.yml* fájlt, és minden 50000-en felüli portszám esetén a mapping-et átírjuk, pl. - "50070:50070" helyett legyen - "30070:50070".

Az Apache Spark Scala parancssori kliense

Az Apache Spark beépített parancssori klienssel [5] rendelkezik, amivel könnyen és gyorsan tudunk hozzáférni a Spark motorhoz, és interaktív módon tudunk adatelemzéseket végezni. A Spark kliensnek két változata van, az egyik a `spark-shell`, ami Scala utasításokat tud végrehajtani. A Scala [6] JVM (vagy akár JavaScript motor) fölött futó erősen típusos OO és funkcionális programnyelv, fejlett többszálúság kezeléssel (így ideális választás Spark-hoz). A másik interaktív parancssori kliens a `pyspark`, amely ugyanazt a funkcionalitást nyújtja, mint a

`spark-shell`, de Python [7] nyelvű utasításokat tud végrehajtani. Ezáltal egy dinamikus típussal rendelkező, szkript nyelv segítségével tudunk hozzáférni a Spark API-khoz. Jelen olvasóleckék során a `spark-shell`-t fogjuk használni, a `pyspark` és Python alapú Spark programokról a `9g_b_BigData-spark-python-cli-python-SPOC` olvasóleckéből tudhat meg többet az olvasó.

A `spark-shell` indításához be kell lépnünk a `spark-master` docker container-be, ott érhető el a bekonfigurált kliens. Adjuk ki a következő utasítást a parancssorból:

```
$ docker exec -it spark-master bash
root@2c8f5e82d5b5:/# cd spark/
root@2c8f5e82d5b5:/spark# bin/spark-shell
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
20/08/28 11:50:37 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
20/08/28 11:50:42 WARN metastore.ObjectStore: Failed to get database
global_temp, returning NoSuchObjectException
Spark context Web UI available at http://172.22.0.4:4040
Spark context available as 'sc' (master = local[*], app id = local-
1598615438575).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.1.2-SNAPSHOT
      /_/

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_121)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Az interaktív parancssorunk üzemképes, ahogy látjuk két előre definiált változó is rendelkezésünkre áll a Spark kommunikációhoz: `sc` (Spark context) és `spark` (Spark session). Ezeket tetszőleges Scala utasításban használhatjuk. A shell tulajdonképpen egy teljes értékű Scala értelmező és végrehajtó, bármilyen Scala programot írhatunk benne, de természetesen mi a Spark fölötti programok készítéséhez használjuk. Írjuk is meg az első Spark programunkat. Töltsük be a lokális fájlrendszer `/spark/README.md` állományát, azaz készítsünk ebből egy `Dataset`-et (illetve `RDD`-t):

```
scala> val df = spark.read.textFile("file:///spark/README.md")
df: org.apache.spark.sql.Dataset[String] = [value: string]

scala> df.rdd
res1: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at rdd at
<console>:26
```

Az első paranccsal létrehoztunk egy `df` változót, ami egy `Dataset<String>` objektum lett, a tárolt elemek string-ek, amik a lokális fájlrendszeren található szövegfájl egyes sorai. Fontos megjegyezni, hogy ha nem használjuk a `file://` protokollt, akkor a Spark alapból a HDFS-ről próbálja betölteni a fájlt, mert a docker image-ek előre be vannak konfigurálva így. Látszik, hogy a

shell alapból az újabb Dataset API-t használja, de bármilyen típusú `Dataset`-ből vagy `DataFrame`-ből könnyedén `RDD`-t készíthetünk, ahogy azt a második utasítás mutatja. Hajtsunk végre transzformációkat, illetve akciókat a `Dataset`-en (lásd `9e_BigData-spark-rdd-df-SPOC.md`).

```
scala> val mappedDf = df.flatMap(line => line.split(" "))
mappedDf: org.apache.spark.sql.Dataset[String] = [value: string]

scala> mappedDf.show()
+----------+
|     value|
+----------+
|         #|
|    Apache|
|     Spark|
|          |
|     Spark|
|        is|
|         a|
|      fast|
|       and|
|   general|
|   cluster|
| computing|
|    system|
|       for|
|       Big|
|     Data.|
|        It|
|  provides|
|high-level|
|      APIs|
+----------+
only showing top 20 rows

scala> mappedDf.count()
res10: Long = 568

scala> mappedDf.distinct().count()
res11: Long = 289
```

A `df` a szöveg fájl sorait tartalmazza, ebből a `flatMap` transzformációval készítettünk egy olyan `Dataset`-et, ami szóközök mentén szétdarabolja a fájl szövegét, és minden szót egy külön elemként tárol el (lásd a `show()` által kiírt elemeket). Amennyiben a `flatMap` helyett simán csak `map`-et használnánk (próbáljuk ki!), akkor továbbra is minden szöveg sorhoz egy elem tartozna, az értéke viszont az abban a sorban szereplő szavak tömbje lenne. A `flatMap` ezt "kilapítja", és a tömb elemeiből külön elemeket készít. Az így előállított `mappedDf` `Dataset`-en aztán meghívjuk a `count` akciót, ami visszaadja a `Dataset` elemeinek (a szöveg szóinak) számát, ami `568`. A második esetben előbb egy `distinct()` transzformációt is elvégzünk a `Dataset`-en, ami csak a különböző szavakat tartja meg a transzformált `Dataset`-ben, amire ismét meghívva a `count()`-ot már csak `289` lesz az eredmény (ennyi különböző szó van a szövegfájlban).

📖 2. fejezet

Word count és árfolyam átlag feladatok megoldása

Word count probléma lokális input fájllal

Most, hogy megismerkedtünk a `spark-shell` alapvető használatával, oldjuk meg a klasszikus MapReduce példa problémát, a word count, azaz szó összeszámoló problémát. Számoljuk össze a `README.md` fájlban szereplő szavak előfordulásainka számát. Ez klasszikusan egy map/reduce programmal könnyen megoldható feladat, ami a Spark-ban is nagyon könnyedén megfogalmazható, hiszen mind a map mind a reduce műveleteket támogatja (sőt azoknál sokkal többet). Lássuk a feladat megoldását:.

```
scala> val df = spark.read.textFile("file:///spark/README.md")
df: org.apache.spark.sql.Dataset[String] = [value: string]

scala> val mappedDf = df.flatMap(line => line.split(" "))
mappedDf: org.apache.spark.sql.Dataset[String] = [value: string]

scala> val mappedWords = mappedDf.map(w => (w, 1))
mappedWords: org.apache.spark.sql.Dataset[(String, Int)] = [_1: string, _2: int]

scala> val summedWords = mappedWords.rdd.reduceByKey((x, y) => x + y)
summedWords: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[89] at
reduceByKey at <console>:29

scala> summedWords.collect()
res23: Array[(String, Int)] = Array((package,1), (For,3), (Programs,1),
(processing.,1), (Because,1), (The,1), (page]
(http://spark.apache.org/documentation.html).,1), (cluster.,1), (its,1),
([run,1), (than,1), (APIs,1), (have,1), (Try,1), (computation,1), (through,1),
(several,1), (This,2), (graph,1), (Hive,2), (storage,1), (["Specifying,1),
(To,2), ("yarn",1), (Once,1), (prefer,1), (SparkPi,2), (engine,1), (version,1),
(file,1), (documentation,,1), (processing,,1), (the,24), (are,1), (systems.,1),
(params,1), (not,1), (different,1), (refer,2), (Interactive,2), (R,,1),
(given.,1), (if,4), (build,4), (when,1), (be,2), (Tests,1), (Apache,1),
(thread,1), (programs,,1), (including,4), (./bin/run-example,2), (Spark.,1),
(package.,1), (1000).count(),1), (Versions,1), (HDFS,1), (Data.,1), (>>>...

scala> summedWords.filter(w => w._1=="the").collect()
res24: Array[(String, Int)] = Array((the,24))
```

A fájl betöltést és a `flatMap` szerepét már láttuk. A szavak `Dataset`-jén elvégzünk egy `map` transzformációt, ami minden szóhoz egy (K, V) kulcs-érték párt rendel, ahol a kulcs maga a szó, az érték pedig `1`. Ezután jön a reduce lépés, amit a leghatékonyabban a Spark `reduceByKey` transzformációjával végezhető el. Mivel ezt a transzformációt csak az `RDD` API támogatja, ezért előbb a `Dataset`-ből `RDD`-t készítünk, és meghívjuk rá a transzformációt. A paraméterben egy olyan függvényt adunk át, ami két értéket összead (ha adott egy `(K, V1)` és `(K, V2)` pár, akkor a transzformáció eredménye `(K, V1+V2)` lesz, hiszen a függvényt a `V1` és `V2` értékekkel fogja meghívni a Spark). Ezután nincs más dolgunk, mint összegyűjteni egy tömbbe a kiszámított (szó, előfordulás szám) párokat, ami a feladat végeredménye. Ha például egy konkrét szó előfordulásának számára vagyunk kíváncsiak, használhatjuk a `filter` transzformációt a fenti módon.

Természetesen nem kell minden lépés eredményét egy változóban eltárolni, csak a példa szemléletesebbé tétele érdekében csináltuk. A műveleteket össze lehet kötni egy hívási láncba így:

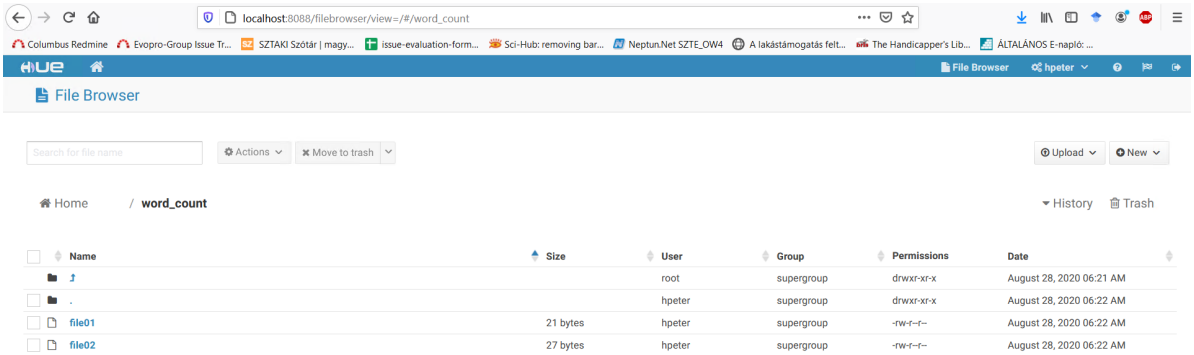
```
scala> df.flatMap(line => line.split(" ")).map(w => (w, 1)).rdd.reduceByKey((x,
y) => x + y).collect()
```

Természetesen más megoldás is létezik a problémára (a Spark gazdag operátor készletének köszönhetően), íme néhány:

```
scala> df.flatMap(line => line.split("
")).groupByKey(identity).count().collect()

scala> df.flatMap(line => line.split(" ")).map(w => (w,
1)).groupBy("_1").sum().collect()
```

Word count probléma HDFS input fájllal

A fenti megoldást alkalmazzuk HDFS-en tárolt szövegállományokra is. Először másoljuk fel HDFS-re a `code/5g_BigData-mapred-SPOC/input` mappában található `file01` és `file02` állományokat, majd a fenti word count megoldást hajtsuk végre azokat használva bemenetként. Átmásolhatjuk a fájlokat először a `namenode` container-be, majd onnan a `hadoop` klienssel feltölthetjük a HDFS-re, ám a Hue használatával egyszerűbben is felmásolhatjuk a fájlokat. Nyissuk meg a böngészőben a http://localhost:8088/filebrowser/#/ címet, és hozzunk létre egy új könyvtárat a `New` gomb segítségével ("word_count"), majd az `Upload`-ot használva tallózzuk ki a két fájlt és töltsük fel őket HDFS-re közvetlenül a lokális gépünkről (lásd ábra).



Ha ez megvan, akkor ezekből a fájlokból töltsük be az input `Dataset`-et, és erre hajtsuk végre a fenti megoldások egyikét:

```
scala> val hdf = spark.read.textFile("/word_count/*")
hdf: org.apache.spark.sql.Dataset[String] = [value: string]

scala> hdf.show()
+--------------------+
|               value|
+--------------------+
|Hello Hadoop Good...|
|Hello World Bye W...|
+--------------------+

scala> hdf.flatMap(line => line.split(" ")).map(w => (w,
1)).groupBy("_1").sum().collect()
res6: Array[org.apache.spark.sql.Row] = Array([World,2], [Goodbye,1], [Hello,2],
[Bye,1], [Hadoop,2])
```

A betöltésnél használhatunk wildcard-okat több fájl együttes betöltésére. Mivel nem adtunk meg protokollt, így alapértelmezetten a HDFS-en keresi a mappát (megegyezik a `hdfs://` használatával).

Árfolyam átlagok számítása országonként

Oldjuk meg a `5g_BigData-mapred-SPOC` gyakorlati olvasólecke 3. fejezetében kitűzött feladatot. Ehhez másoljuk fel a `code/5g_BigData-mapred-SPOC/data/daily_csv.csv` fájlt HDFS-re (a fenti módon), és írjunk olyan Spark programot, amely országonként kiszámítja az átlagos USD árfolyam értéket. Lássuk a megoldást:

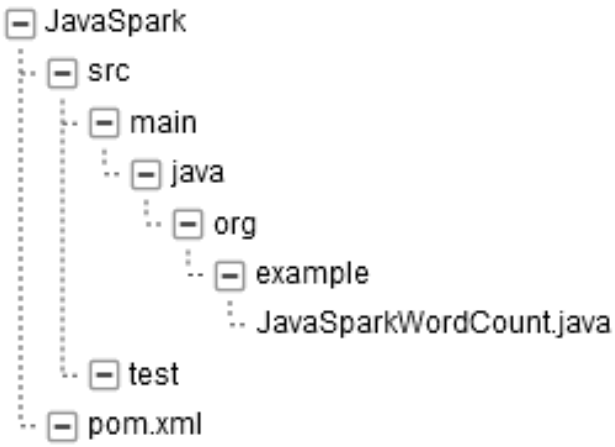
```
scala> val df = spark.read.option("header", true).csv("/data/daily_csv.csv")

df: org.apache.spark.sql.DataFrame = [Date: string, Country: string ... 1 more
field]


scala> df.show()

+----------+---------+------+

|      Date|  Country| Value|

+----------+---------+------+

|1971-01-04|Australia|0.8987|

|1971-01-05|Australia|0.8983|

|1971-01-06|Australia|0.8977|

|1971-01-07|Australia|0.8978|

|1971-01-08|Australia| 0.899|

|1971-01-11|Australia|0.8967|

|1971-01-12|Australia|0.8964|

|1971-01-13|Australia|0.8957|

|1971-01-14|Australia|0.8937|

|1971-01-15|Australia|0.8943|

|1971-01-18|Australia|0.8945|

|1971-01-19|Australia|0.8934|

|1971-01-20|Australia|0.8934|

|1971-01-21|Australia| 0.893|

|1971-01-22|Australia|0.8925|

|1971-01-25|Australia|0.8909|
```

```
|1971-01-26|Australia|0.8905|

|1971-01-27|Australia|0.8905|

|1971-01-28|Australia|0.8902|

|1971-01-29|Australia|  0.89|

+----------+---------+------+

only showing top 20 rows


scala> df.groupBy("Country").agg(mean("Value")).collect()
res21: Array[org.apache.spark.sql.Row] = Array([Sweden,6.7553831747918816],
[Malaysia,2.9909150174422585], [Singapore,1.6717471317662342],
[Taiwan,31.208181032818533], [China,6.158194064026062],
[India,31.294657237951395], [Norway,6.655837098692055],
[Denmark,6.621088551044671], [Thailand,31.299020480748407], [Hong
Kong,7.653892366576895], [Venezuela,3.0033560500695846], [South
Korea,981.608407379105], [Mexico,11.163365298692703], [Euro,0.8462333403449731],
[Switzerland,1.6787115329086926], [Canada,1.2178624140565348],
[Brazil,2.1426414032650305], [Japan,161.39410060327953], [New
Zealand,1.4602258015137293], [Australia,1.2137489717879033], [South
Africa,4.791916712034976], [United Kingdom,0.5910698343949052])
```

📖 3. fejezet
Standalone Spark alkalmazás Java nyelven

Ebben a fejezetben bemutatjuk, hogyan lehet olyan önálló alkalmazást írni Java nyelven, amely végrehajtható feladatként átadható a Spark-nak. Azaz ebben az esetben nem az interaktív klienst használjuk a Spark API eléréséhez, hanem önálló Java programot készítünk, ami természetesen használja a Spark könyvtárakat. Java esetén a preferált fejlesztés a Maven [8] build rendszer segítségével történik. A Maven projektekhez könnyen hozzá tudjuk adni a megfelelő könyvtár függőségeket és fordítani tudunk egy végrehajtható jar fájlt, ami a Spark motornak átadható végrehajtásra. Oldjuk meg a fenti word count problémát egy önálló Java programmal.

Ehhez hozzunk létre egy Maven projektet az alábbi könyvtárszerkezettel (`code/9g_a_BigData-spark-scala-cli-java-SPOC/JavaSpark`).



A `src/main/java/org/example/JavaSparkWordCount.java` fájl tartalma a következő:

```java
package org.example;

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

import java.util.Arrays;
import java.util.List;

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

        SparkSession spark = SparkSession
                .builder()
                .appName("JavaWordCount")
                .getOrCreate();

        JavaRDD<String> lines =
spark.read().textFile("/word_count/*").javaRDD();

        JavaRDD<String> words = lines.flatMap(s -> Arrays.asList(s.split("
")).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();
    }
}
```

Az interaktív shell megoldása a Java API-ra átültetve. Először szükségünk van egy `SparkSession` objektumra, majd `JavaRDD`-k sorozatát állítjuk elő a fent már ismertetett transzformációkkal. Majd végül a `collect()` eredményén végigiterálva a végeredményt a konzolra írjuk. A transzformációk esetén a szükséges függvényeket lambda kifejezésekkel adjuk meg.

A `pom.xml`-hez hozzá kell adnunk a Spark könyvtárat mint függőség, illetve az assembly plug-int, hogy függőségekkel egybecsomagolt jar-t tudjunk gyártani:

```xml
<?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>JavaSpark</artifactId>
  <version>1.0-SNAPSHOT</version>
```

```
  <name>JavaSpark</name>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.11</artifactId>
      <version>2.1.0</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <archive>
            <manifest>
              <mainClass>org.example.JavaSparkWordCount</mainClass>
            </manifest>
          </archive>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Fordítsunk egy futtatható jar-t minden függőséggel egybe csomagolva a Maven segítségével. Adjuk ki a `JavaSpark` könyvtárban a következőt:

```
$ mvn assembly:single
```

A `target` mappában létrejövő `JavaSpark-1.0-SNAPSHOT-jar-with-dependencies.jar` fájlt másoljuk fel a `spark-master` container-be:

```
$ docker cp target/JavaSpark-1.0-SNAPSHOT-jar-with-dependencies.jar spark-
master:/spark/JavaSpark-1.0-SNAPSHOT-jar-with-dependencies.jar
```

Ezután lépjünk be a `spark-master` container-be és indítsunk egy új Spark job-ot a feltöltött jar segítségével:

```
$ docker exec -it spark-master bash
root@2c8f5e82d5b5:/# cd spark/
root@2c8f5e82d5b5:/spark# bin/spark-submit --class org.example.JavaSparkWordCount
JavaSpark-1.0-SNAPSHOT-jar-with-dependencies.jar
20/08/28 21:51:41 INFO spark.SparkContext: Running Spark version 2.1.2-SNAPSHOT
...
20/08/28 21:51:46 INFO scheduler.DAGScheduler: Job 0 finished: collect at
JavaSparkWordCount.java:27, took 1.125716 s
Bye: 1
Hello: 2
World: 2
Goodbye: 1
Hadoop: 2
20/08/28 21:51:47 INFO server.ServerConnector: Stopped Spark@6d8a6e04{HTTP/1.1}
{0.0.0.0:4040}
...
```

A `spark-submit` programmal adhatunk át egy önálló alkalmazást futtatásra, a futtatandó osztályt kell csak paraméterben megadnunk. A program beégetett módon a HDFS `/word_count` mappában lévő összes fájlra számolja ki a szó előfordulási gyakoriságokat.

További feladatok

1. Írjuk meg a valutaárfolyamok átlagát kiszámító Spark programot önálló Java alkalmazásként, és hajtsuk végre azt a Spark klaszteren! ⭐
2. Írjunk Spark programot, ami Hive adattáblából veszi a bemenetét (a `docker-compose-hive.yml` docker stack-et kell indítanunk, és amegfelelő Spark API-kat használnunk)! ⭐⭐
3. Adjunk a fent bemutatott word count problémára egy olyan Spark megoldást, ami nem szerepel az olvasóleckében! ⭐
4. Írjuk meg a sorted word count programot Spark-ban, ami az eredeti word count problémától abban tér el, hogy az eredményt az előfordulási gyakoriságok számában csökkenő sorrendbe rendezve adja meg! ⭐
5. Töltsük be a korábbi leckék során bemutatott `personal_entries.json` és `billing_entries.json`, valamint `sales_entries.csv` állományokat Spark-ba, egyesítsük (`union`) a három adathalmazt és csoportosítsuk az adatokat `PID` alapján! ⭐⭐

Referenciák

[1] https://spark.apache.org/

[2] https://spark.apache.org/downloads.html

[3] https://spark.apache.org/docs/latest/cluster-overview.html

[4] https://jupyter.org/

[5] https://spark.apache.org/docs/latest/quick-start.html

[6] https://www.scala-lang.org/

[7] https://www.python.org/

[8] https://maven.apache.org/