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 Python nyelvű interaktív parancsértelmező eszközével. Különböző példákon keresztül demonstráljuk a különböző adatforrásokból történő adat beolvasást és a DataFrame API használatát. Több konkrét transzformációt és akciót is bemutatunk az adatokon. A parancssori kliens mellett betekintést nyújtunk abba, hogyan lehet olyan önálló Python alkalmazásokat fejleszteni, melyek átadhatók a Spark keretrendszernek mind végrehajtható feladat.
A lecke fejezetei:

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

Téma típusa: **gyakorlati**
Olvasási idő: **50 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 Python 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 `pyspark`-ot fogjuk használni, a `spark-shell` és Scala alapú Spark programokról a `9g_a_BigData-spark-scala-cli-java-SPOC` olvasóleckéből tudhat meg többet az olvasó.

A `pyspark` 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/pyspark
Python 2.7.9 (default, Jun 29 2016, 13:08:31)

[GCC 4.9.2] on linux2

Type "help", "copyright", "credits" or "license" for more information.

Setting default log level to "WARN".

To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
20/08/31 09:09:47 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
20/08/31 09:09:51 WARN metastore.ObjectStore: Failed to get database
global_temp, returning NoSuchObjectException
Welcome to

      ____              __

     / __/__  ___ _____/ /__

    _\ \/ _ \/ _ `/ __/  '_/

   /__ / .__/\_,_/_/ /_/\_\   version 2.1.2-SNAPSHOT

      /_/

Using Python version 2.7.9 (default, Jun 29 2016 13:08:31)

SparkSession available as 'spark'.

>>>
```

Az interaktív parancssorunk üzemképes, ahogy látjuk két előre definiált változó is rendelkezésünkre áll a Spark kommunikációhoz: `sc` (SparkContext) és `spark` (SparkSession). Ezeket tetszőleges Python utasításban használhatjuk. A shell tulajdonképpen egy teljes értékű Python értelmező és végrehajtó, bármilyen Python 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 `DataFrame`-et:

```
>>> df = spark.read.text("file:///spark/README.md")
>>> df
DataFrame[value: string]
```

Az első paranccsal létrehoztunk egy `df` változót, ami egy `DataFrame` objektum lett (dinamikus nyelv révén nem lehet a típusos Dataset API-t használni, 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 DataFrame API-t használja, de bármilyen típusú `DataFrame`-ből könnyedén `RDD`-t is készíthetünk. Hajtsunk végre transzformációkat, illetve akciókat a `DataFrame`-en, amihez be kell töltenünk a `pyspark.sql.functions` modult.

```
>>> from pyspark.sql.functions import *
>>> split_col = split(df.value, " ")
>>> exp_col = explode(split_col).alias("exploded")
>>> df.select(exp_col).show()
+-----------+
|   exploded|
+-----------+
|          #|
|     Apache|
|      Spark|
|           |
|      Spark|
|         is|
|          a|
|    unified|
|  analytics|
|     engine|
|        for|
|large-scale|
|       data|
|processing.|
|         It|
|   provides|
| high-level|
|       APIs|
|         in|
|     Scala,|
+-----------+
only showing top 20 rows

>>> df.select(exp_col).count()
568
>>> df.select(exp_col).distinct().count()
289
```

A `df` a szöveg fájl sorait tartalmazza egy oszlopban, ebből a `split` segítségével készítettünk egy `spli_col` nevű oszlopot az eredeti `DataFrame` `value` oszlopából, ami szóközök mentén szétdarabolja a fájl szövegét és szavak tömbjét tárolja egy-egy sorban. Ezután az `explode` művelet "szétrobbantja" ezeket a sorokat, és a tömb által tartalmazott minden egyes szó külön sorba kerül (lásd `show()` eredménye). Az oszlopnak az `alias()` metódussal nevet is adunk (`exploded`). A `DataFrame` így előállított `exp_col` oszlopán aztán meghívjuk a `count` műveletet, ami visszaadja a `DataFrame` megfelelő oszlopában lévő elemek (a szöveg szóinak) számát, ami `568`. A második esetben előbb egy `distinct()` transzformációt is elvégzünk az oszlopon, ami csak a különböző szavakat tartja meg a transzformált `DataFrame`-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 `pyspark` 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:.

```
>>> split_col = split(df.value, " ")
>>> exp_col = explode(split_col).alias("exploded")
>>> group_data = df.select(exp_col).groupBy("exploded")
>>> group_data
<pyspark.sql.group.GroupedData object at 0x7f6093c788d0>
>>> count_frame = group_data.count()
>>> count_frame
DataFrame[exploded: string, count: bigint]
>>> count_frame.collect()
```

```
[Row(exploded=u'[![PySpark', count=1), Row(exploded=u'online', count=1),
Row(exploded=u'graphs', count=1), Row(exploded=u'["Building', count=1),
Row(exploded=u'documentation', count=3), Row(exploded=u'command,', count=2),
Row(exploded=u'abbreviated', count=1), Row(exploded=u'overview', count=1),
Row(exploded=u'rich', count=1), Row(exploded=u'set', count=2), Row(exploded=u'-
DskipTests', count=1), Row(exploded=u'1,000,000,000:', count=2),
Row(exploded=u'name', count=1), Row(exploded=u'["Specifying', count=1),
Row(exploded=u'stream', count=1), Row(exploded=u'run:', count=1),
Row(exploded=u'not', count=1), Row(exploded=u'programs', count=2),
Row(exploded=u'tests', count=2), Row(exploded=u'./dev/run-tests', count=1),
Row(exploded=u'will', count=1), Row(exploded=u'[run', count=1),
Row(exploded=u'particular', count=2), Row(exploded=u'Alternatively,', count=1),
Row(exploded=u'must', count=1), Row(exploded=u'using', count=3),
Row(exploded=u'./build/mvn', count=1), Row(exploded=u'you', count=4),
Row(exploded=u'MLlib', count=1), Row(exploded=u'DataFrames,', count=1),
Row(exploded=u'variable', count=1), Row(exploded=u'Note', count=1),
Row(exploded=u'core', count=1), Row(exploded=u'protocols', count=1),
Row(exploded=u'Guide](https://spark.apache.org/docs/latest/configuration.html)',
count=1), Row(exploded=u'guidance', count=2), Row(exploded=u'shell:', count=2),
Row(exploded=u'can', count=6), Row(exploded=u'site,', count=1),
Row(exploded=u'*', count=4), Row(exploded=u'systems.', count=1),
Row(exploded=u'[building', count=1), Row(exploded=u'configure', count=1),
Row(exploded=u'for', count=12), Row(exploded=u'README', count=1),
Row(exploded=u'Interactive', count=2), Row(exploded=u'how', count=3),
Row(exploded=u'[Configuration', count=1), Row(exploded=u'Hive', count=2),
Row(exploded=u'provides', count=1), Row(exploded=u'Hadoop-supported', count=1),
Row(exploded=u'pre-built', count=1), Row(exploded=u'["Useful', count=1),
Row(exploded=u'directory.', count=1), Row(exploded=u'Example', count=1),
Row(exploded=u'example', count=3), Row(exploded=u'Kubernetes', count=1),
Row(exploded=u'one', count=2), Row(exploded=u'MASTER', count=1),
Row(exploded=u'guide](https://spark.apache.org/contributing.html)', count=1),
Row(exploded=u'in', count=5), Row(exploded=u'library', count=1),
Row(exploded=u'Spark.', count=1), Row(exploded=u'contains', count=1),
Row(exploded=u'Configuration', count=1), Row(exploded=u'programming', count=1),
Row(exploded=u'with', count=3), Row(exploded=u'contributing', count=1),
Row(exploded=u'downloaded', count=1), Row(exploded=u'1000).count()', count=2),
Row(exploded=u'comes', count=1), Row(exploded=u'machine', count=1),
Row(exploded=u'building', count=2), Row(exploded=u'params', count=1),
Row(exploded=u'given.', count=1), Row(exploded=u'be', count=2),
Row(exploded=u'same', count=1), Row(exploded=u'integration', count=1),
Row(exploded=u'Programs', count=1), Row(exploded=u'locally', count=2),
Row(exploded=u'using:', count=1), Row(exploded=u'[Apache', count=1),
Row(exploded=u'your', count=1), Row(exploded=u'optimized', count=1),
Row(exploded=u'Developer', count=1), Row(exploded=u'R,', count=1),
Row(exploded=u'[![AppVeyor', count=1), Row(exploded=u'should', count=2),
Row(exploded=u'graph', count=1), Row(exploded=u'package', count=1),
Row(exploded=u'[project', count=1), Row(exploded=u'project', count=1),
Row(exploded=u'`examples`', count=2), Row(exploded=u'resource-
managers/kubernetes/integration-tests/README.md', count=1),
Row(exploded=u'versions', count=1), Row(exploded=u'Spark"]
(https://spark.apache.org/docs/latest/building-spark.html).', count=1),
Row(exploded=u'Spark](#building-spark).', count=1), Row(exploded=u'general',
count=2), Row(exploded=u'other', count=1), Row(exploded=u'1000', count=2),
Row(exploded=u'learning,', count=1), Row(exploded=u'when', count=1),
Row(exploded=u'submit', count=1), Row(exploded=u'Apache', count=1),
Row(exploded=u'detailed', count=2), Row(exploded=u'About', count=1),
Row(exploded=u'is', count=7), Row(exploded=u'on', count=7),
Row(exploded=u'scala>', count=1), Row(exploded=u'print', count=1),
```

```
Row(exploded=u'Tools"](https://spark.apache.org/developer-tools.html).',
count=1), Row(exploded=u'use', count=3), Row(exploded=u'different', count=1),
Row(exploded=u'following', count=2), Row(exploded=u'YARN"]
(https://spark.apache.org/docs/latest/building-spark.html#specifying-the-hadoop-
version-and-enabling-yarn)', count=1),
Row(exploded=u'<https://spark.apache.org/>', count=1), Row(exploded=u'SparkPi',
count=2), Row(exploded=u'refer', count=2), Row(exploded=u'./bin/run-example',
count=2), Row(exploded=u'data', count=2), Row(exploded=u'Tests', count=1),
Row(exploded=u'Versions', count=1), Row(exploded=u'processing.', count=2),
Row(exploded=u'its', count=1), Row(exploded=u'tests]
(https://spark.apache.org/developer-tools.html#individual-tests).', count=1),
Row(exploded=u'basic', count=1), Row(exploded=u'latest', count=1),
Row(exploded=u'only', count=1), Row(exploded=u'<class>', count=1),
Row(exploded=u'have', count=1), Row(exploded=u'runs.', count=1),
Row(exploded=u'You', count=3), Row(exploded=u'tips,', count=1),
Row(exploded=u'project.', count=1), Row(exploded=u'developing', count=1),
Row(exploded=u'YARN,', count=1), Row(exploded=u'It', count=2),
Row(exploded=u'"local"', count=1), Row(exploded=u'processing,', count=1),
Row(exploded=u'built', count=1), Row(exploded=u'Pi', count=1),
Row(exploded=u'thread,', count=1), Row(exploded=u'A', count=1),
Row(exploded=u'APIs', count=1), Row(exploded=u'Scala,', count=1),
Row(exploded=u'file', count=1), Row(exploded=u'computation', count=1),
Row(exploded=u'Once', count=1), Row(exploded=u'find', count=1),
Row(exploded=u'the', count=23), Row(exploded=u'To', count=2),
Row(exploded=u'uses', count=1), Row(exploded=u'Version', count=1),
Row(exploded=u'N', count=1), Row(exploded=u'programs,', count=1),
Row(exploded=u'"yarn"', count=1), Row(exploded=u'see', count=3),
Row(exploded=u'./bin/pyspark', count=1), Row(exploded=u'Structured', count=1),
Row(exploded=u'[![Jenkins', count=1), Row(exploded=u'return', count=2),
Row(exploded=u'Java,', count=1), Row(exploded=u'from', count=1),
Row(exploded=u'Because', count=1), Row(exploded=u'Streaming', count=1),
Row(exploded=u'More', count=1), Row(exploded=u'cluster', count=1),
Row(exploded=u'analytics', count=1), Row(exploded=u'analysis.', count=1),
Row(exploded=u'cluster.', count=1), Row(exploded=u'Running', count=1),
Row(exploded=u'Please', count=4), Row(exploded=u'talk', count=1),
Row(exploded=u'distributions.', count=1), Row(exploded=u'guide,', count=1),
Row(exploded=u'There', count=1), Row(exploded=u'"local[N]"', count=1),
Row(exploded=u'Try', count=1), Row(exploded=u'and', count=9),
Row(exploded=u'do', count=2), Row(exploded=u'Scala', count=2),
Row(exploded=u'class', count=2), Row(exploded=u'build', count=3),
Row(exploded=u'setup', count=1), Row(exploded=u'need', count=1),
Row(exploded=u'spark://', count=1), Row(exploded=u'Hadoop,', count=2),
Row(exploded=u'Thriftserver', count=1), Row(exploded=u'are', count=1),
Row(exploded=u'requires', count=1), Row(exploded=u'package.', count=1),
Row(exploded=u'Enabling', count=1), Row(exploded=u'clean', count=1),
Row(exploded=u'large-scale', count=1), Row(exploded=u'high-level', count=1),
Row(exploded=u'SQL', count=2), Row(exploded=u'page]
(https://spark.apache.org/documentation.html).', count=1),
Row(exploded=u'against', count=1), Row(exploded=u'of', count=5),
Row(exploded=u'through', count=1), Row(exploded=u'review', count=1),
Row(exploded=u'package.)', count=1), Row(exploded=u'Python,', count=2),
Row(exploded=u'easiest', count=1), Row(exploded=u'no', count=1),
Row(exploded=u'Testing', count=1), Row(exploded=u'several', count=1),
Row(exploded=u'help', count=1), Row(exploded=u'The', count=1),
Row(exploded=u'sample', count=1), Row(exploded=u'MASTER=spark://host:7077',
count=1), Row(exploded=u'examples', count=2), Row(exploded=u'an', count=4),
Row(exploded=u'#', count=1), Row(exploded=u'Online', count=1),
Row(exploded=u'test,', count=1), Row(exploded=u'including', count=4),
```

```
Row(exploded=u'usage', count=1), Row(exploded=u'Python', count=2),
Row(exploded=u'at', count=2), Row(exploded=u'development', count=1),
Row(exploded=u'IDE,', count=1), Row(exploded=u'way', count=1),
Row(exploded=u'Contributing', count=1), Row(exploded=u'get', count=1),
Row(exploded=u'that', count=2), Row(exploded=u'##', count=9),
Row(exploded=u'For', count=3), Row(exploded=u'prefer', count=1),
Row(exploded=u'This', count=2), Row(exploded=u'running', count=1),
Row(exploded=u'Build]
(https://img.shields.io/appveyor/ci/ApacheSoftwareFoundation/spark/master.svg?
style=plastic&logo=appveyor)]
(https://ci.appveyor.com/project/ApacheSoftwareFoundation/spark)', count=1),
Row(exploded=u'web', count=1), Row(exploded=u'run', count=7),
Row(exploded=u'locally.', count=1), Row(exploded=u'Spark', count=14),
Row(exploded=u'URL,', count=1), Row(exploded=u'a', count=9),
Row(exploded=u'higher-level', count=1), Row(exploded=u'tools', count=1),
Row(exploded=u'if', count=4), Row(exploded=u'available', count=1),
Row(exploded=u'', count=73), Row(exploded=u'Documentation', count=1),
Row(exploded=u'this', count=1), Row(exploded=u'Maven]
(https://maven.apache.org/).', count=1), Row(exploded=u'(You', count=1),
Row(exploded=u'Build](https://amplab.cs.berkeley.edu/jenkins/job/spark-master-
test-sbt-hadoop-2.7-hive-2.3/badge/icon)]
(https://amplab.cs.berkeley.edu/jenkins/job/spark-master-test-sbt-hadoop-2.7-
hive-2.3)', count=1), Row(exploded=u'>>>', count=1),
Row(exploded=u'information', count=1), Row(exploded=u'info', count=1),
Row(exploded=u'unified', count=1), Row(exploded=u'Shell', count=2),
Row(exploded=u'environment', count=1), Row(exploded=u'built,', count=1),
Row(exploded=u'module,', count=1), Row(exploded=u'them,', count=1),
Row(exploded=u'`./bin/run-example', count=1), Row(exploded=u'instance:',
count=1), Row(exploded=u'first', count=1), Row(exploded=u'[Contribution',
count=1), Row(exploded=u'documentation,', count=1), Row(exploded=u'[params]`.',
count=1), Row(exploded=u'mesos://', count=1), Row(exploded=u'engine', count=2),
Row(exploded=u'GraphX', count=1), Row(exploded=u'example:', count=1),
Row(exploded=u'HDFS', count=1), Row(exploded=u'or', count=3),
Row(exploded=u'to', count=16), Row(exploded=u'Hadoop', count=3),
Row(exploded=u'individual', count=1), Row(exploded=u'also', count=5),
Row(exploded=u'changed', count=1), Row(exploded=u'started', count=1),
Row(exploded=u'./bin/spark-shell', count=1), Row(exploded=u'threads.', count=1),
Row(exploded=u'supports', count=2), Row(exploded=u'storage', count=1),
Row(exploded=u'version', count=1), Row(exploded=u'instructions.', count=1),
Row(exploded=u'Building', count=1), Row(exploded=u'start', count=1),
Row(exploded=u'Many', count=1), Row(exploded=u'which', count=2),
Row(exploded=u'Coverage](https://img.shields.io/badge/dynamic/xml.svg?
label=pyspark%20coverage&url=https%3A%2F%2Fspark-test.github.io%2Fpyspark-
coverage-
site&query=%2Fhtml%2Fbody%2Fdiv%5B1%5D%2Fdiv%2Fh1%2Fspan&colorB=brightgreen&styl
e=plastic)](https://spark-test.github.io/pyspark-coverage-site)', count=1),
Row(exploded=u'spark.range(1000', count=2), Row(exploded=u'And', count=1),
Row(exploded=u'distribution', count=1)]
>>>
>>> count_frame.filter("exploded=='the'").collect()
[Row(exploded=u'the', count=23)]
```

A fájl betöltést és a `split` valamint `explode` szerepét már láttuk. A szavak oszlopán elvégzünk egy `groupBy` műveletet, ami egy `GroupedData` objektumot állít elő, ami az azonos szavakat csoportosítva tartalmazza. Ezen már csak egy `count()` műveletet kell meghívnunk, ami a csoportosított sorok számát összegzi, és egy új oszlopként (`count`) hozzáilleszti az eredeti `DataFrame`-hez. Ezen a `DataFrame`-en a `collect()`-et meghívva, megkapjuk az eredmény

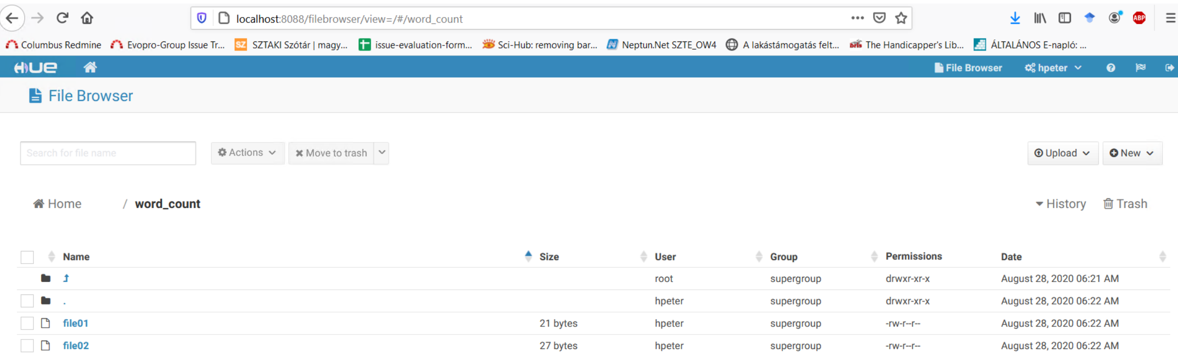
táblázat sorait egy tömbben (`Row` típusú elemek tömbjét két oszloppal). Ha például egy konkrét szó előfordulásának számára vagyunk kíváncsiak, használhatjuk a `filter` műveletet a fenti módon.

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:

```
>>> df.select(explode(split(df.value, "
")).alias("exploded")).groupBy("exploded").count().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 `DataFrame`-et, és erre hajtsuk végre a fenti megoldások egyikét:

```
>>> hdf = spark.read.text("hdfs://namenode:8020/word_count/*")
>>> hdf.show()
+--------------------+
|               value|
+--------------------+
|Hello Hadoop Good...|
|Hello World Bye W...|
+--------------------+

>>> hdf.select(explode(split(hdf.value, "
")).alias("exploded")).groupBy("exploded").count().collect()
[Row(exploded=u'World', count=2), Row(exploded=u'Goodbye', count=1),
Row(exploded=u'Hello', count=2), Row(exploded=u'Bye', count=1),
Row(exploded=u'Hadoop', count=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:

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

>>> df.groupBy("Country").agg(mean("Value")).collect()
```

```
[Row(Country=u'Sweden', avg(Value)=6.7553831747918816), Row(Country=u'Malaysia',
avg(Value)=2.9909150174422585), Row(Country=u'Singapore',
avg(Value)=1.6717471317662342), Row(Country=u'Taiwan',
avg(Value)=31.208181032818533), Row(Country=u'China',
avg(Value)=6.158194064026062), Row(Country=u'India',
avg(Value)=31.294657237951395), Row(Country=u'Norway',
avg(Value)=6.655837098692055), Row(Country=u'Denmark',
avg(Value)=6.621088551044671), Row(Country=u'Thailand',
avg(Value)=31.299020480748407), Row(Country=u'Hong Kong',
avg(Value)=7.653892366576895), Row(Country=u'Venezuela',
avg(Value)=3.0033560500695846), Row(Country=u'South Korea',
avg(Value)=981.608407379105), Row(Country=u'Mexico',
avg(Value)=11.163365298692703), Row(Country=u'Euro',
avg(Value)=0.8462333403449731), Row(Country=u'Switzerland',
avg(Value)=1.6787115329086926), Row(Country=u'Canada',
avg(Value)=1.2178624140565348), Row(Country=u'Brazil',
avg(Value)=2.1426414032650305), Row(Country=u'Japan',
avg(Value)=161.39410060327953), Row(Country=u'New Zealand',
avg(Value)=1.4602258015137293), Row(Country=u'Australia',
avg(Value)=1.2137489717879033), Row(Country=u'South Africa',
avg(Value)=4.791916712034976), Row(Country=u'United Kingdom',
avg(Value)=0.5910698343949052)]
```

3. fejezet

Standalone Spark alkalmazás Python nyelven

Ebben a fejezetben bemutatjuk, hogyan lehet olyan önálló alkalmazást írni Python nyelven, amely végrehajtható feladatként átadható a Spark-nak. Azaz ebben az esetben nem az interaktív klienst használjuk a Spark API eléréséhez, hanem önálló Python programot készítünk, ami természetesen használja a Spark könyvtárakat. Oldjuk meg a fenti word count problémát egy önálló Python programmal. Ehhez szükség van a `pyspark` Python modul-ra, hogy elérhessük a Spark API funkcionalitását.

Készítsük el az önálló alkalmazás Python kódját (`code/9g_b_BigData-spark-python-cli-python-SPOC/PySparkWordCount.py`):

```
"""PySparkWordCount.py"""
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

textFile = "/word_count/*"  # text files
spark = SparkSession.builder.appName("PySparkWordCount").getOrCreate()
textData = spark.read.text(textFile).cache()

split_col = split(textData.value, " ")
exp_col = explode(split_col).alias("exploded")
group_data = textData.select(exp_col).groupBy("exploded")
count_frame = group_data.count()
for row in count_frame.collect():
    print("%s, %s" % (row['exploded'], row['count']))

spark.stop()
```

Az interaktív shell megoldása a Python API-ra átültetve. Először szükségünk van egy `SparkSession` objektumra, majd az oszlop transzformációk sorozatával előállítjuk az eredményt. Végül a `collect()` eredményén végigiterálva a végeredményt a konzolra írjuk.

A `PySparkWordCount.py` fájlt másoljuk fel a `spark-master` container-be:

```
$ docker cp PySparkWordCount.py spark-master:/spark/PySparkWordCount.py
```

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

```
$ docker exec -it spark-master bash
root@2c8f5e82d5b5:/# cd spark/
root@2c8f5e82d5b5:/spark# bin/spark-submit PySparkWordCount.py
20/08/31 13:29:08 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
20/08/31 13:29:08 INFO SparkContext: Running Spark version 3.0.0
20/08/31 13:29:08 INFO ResourceUtils:
==============================================================
20/08/31 13:29:08 INFO ResourceUtils: Resources for spark.driver:

20/08/31 13:29:08 INFO ResourceUtils:
==============================================================
20/08/31 13:29:08 INFO SparkContext: Submitted application: PySparkWordCount
...
20/08/31 13:29:17 INFO DAGScheduler: Job 0 finished: collect at
/spark/PySparkWordCount.py:13, took 3.269971 s
World, 2
Goodbye, 1
Hello, 2
Bye, 1
Hadoop, 2
20/08/31 13:29:17 INFO SparkUI: Stopped Spark web UI at http://43642cd7fb93:4041
...
```

A `spark-submit` programmal adhatunk át egy önálló alkalmazást futtatásra, a futtatandó Python szkriptet 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 Python alkalmazásként, és hajtsuk végre azt a Spark klaszteren! ⭐
2. Írjuk meg a sorted word count programot Spark-ban (`pyspark`-ot használjuk(, ami az eredeti word count problémától abban tér el, hogy az eredményt az előfordulási gyakoriságok számában csökkenő sorrendbe rendezve adja meg! ⭐
3. 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/

[9] https://www.py4j.org/