ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
INFORMATIKAI KAR

SZAKDOLGOZAT

OE-NIK Hallgató neve: **Stampel János**

2021 Hallgató törzskönyvi száma: **T/005809/FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Kiberfizikai Rendszerek Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Stampel János**
Törzskönyvi száma: T/005809/FI12904/N
Neptun kódja: [redacted]

A dolgozat címe:

Adatbetöltési módszerek Big Data rendszereken
Data Ingestion methods in Big Data systems

Intézményi konzulens: Légrádi Gábor
Külső konzulens:

Beadási határidő: 2022. május 15.

A záróvizsga tárgyai: Számítógép architektúrák
Big Data és üzleti intelligencia specializáció

A feladat

Tervezzen és valósítson meg olyan Big Data alapú rendszereket, melyekben stream és batch alapú adatbetöltési feladatokat hajt végre! Vizsgálja meg és elemezze az egyes adatbetöltési mechanizmusokat, valamint optimalizálja azokat performancia szempontjából! Valósítsa meg szoftveralkalmazással az adatbetöltés automatizálását az egyes rendszerekbe! Végezzen el összehasonlítást a létrehozott rendszerek között és mutassa be alkalmazási területeit a vázolt adatbetöltési folyamatokon keresztül! Az elkészült munkát dokumentálja!

A dolgozatnak tartalmaznia kell:

- szakirodalom feldolgozását Big Data technológiák területén,
- tesztadatok létrehozását az adatbetöltési folyamatokban való felhasználásra,
- forrásrendszerek létrehozását a tesztadatok tárolásának céljából tetszőleges adatbázisban vagy adatbázisokban,
- Big Data klaszter létrehozását *batch* alapú adatbetöltés céljából,
- *batch* alapú adatbetöltés megvalósítását, a töltési mechanizmusok optimalizációját,
- valósítsa meg a *batch* alapú adatbetöltés automatizálását és klaszter létrehozását szoftveralkalmazással
- Big Data rendszer létrehozását *stream* alapú adatbetöltés céljából,
- *stream* alapú adatbetöltés megvalósítását, a töltési mechanizmusok optimalizációját,
- valósítsa meg a *stream* alapú töltési láncokat szoftveralkalmazással,
- összehasonlítást a két Big Data alapú rendszer, illetve az azokban végzett műveletek között.

Fleiner R

.....
Dr. Fleiner Rita
intézetigazgató

A szakdolgozat elévülésének határideje: **2024. május 15.**
(OE TVSz 55.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

Lőrincz Gábor

....
külső konzulens intézményi konzulens

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat / diplomamunka saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban / diplomamunkámban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2022. 05. 12.

.....

hallgató aláírása

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

Stampel János ..... [redacted] ..... Nappali

Telefon: Levelezési cím (pl: lakcím):

[redacted] .....

Szakdolgozat / Diplomamunka[1] címe magyarul:

Adatbetöltési módszerek Big Data rendszereken .....

Szakdolgozat / Diplomamunka[2] címe angolul:

Data Ingestion methods in Big Data systems .....

Intézményi konzulens: Külső konzulens:

Légrádi Gábor ....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2021. 10. 13	Téma kiválasztása, feladatlap kidolgozása	Légrádi Gábor
2.	2021. 10. 27.	Tartalomjegyzék és felépítés megbeszélése	Légrádi Gábor
3.	2021. 11. 10.	Irodalomkutatás áttekintése	Légrádi Gábor
4.	2021. 12. 01.	Rendszerterv megbeszélése	Légrádi Gábor

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplomamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 2021. 12. 08.

Légrádi Gábor
.....
intézményi konzulens

[1] Megfelelő aláhúzandó!
[2] Megfelelő aláhúzandó!
[3] Megfelelő aláhúzandó!
[4] Megfelelő aláhúzandó!

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

Stampel János ..... [redacted] ..... Nappali

Telefon: Levelezési cím (pl: lakcím):

[redacted] .....

Szakdolgozat / Diplomamunka[1] címe magyarul:

Adatbetöltési módszerek Big Data rendszereken .....

Szakdolgozat / Diplomamunka[2] címe angolul:

Data Ingestion methods in Big Data systems .....

Intézményi konzulens: Külső konzulens:

Légrádi Gábor ....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2021. 03. 11.	Rendszerterv megvalósitása	Légrádi Gábor
2.	2021. 03. 25.	Tesztek kidolgozása	Légrádi Gábor
3.	2021. 04. 15.	Teljesítmény összehasonlítása	Légrádi Gábor
4.	2021. 04. 29.	Szakdolgozat formázása, fogalmazás javítása	Légrádi Gábor

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplomamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 2022. 05. 03.

Légrádi Gábor
.....
intézményi konzulens

[1] Megfelelő aláhúzandó!
[2] Megfelelő aláhúzandó!
[3] Megfelelő aláhúzandó!
[4] Megfelelő aláhúzandó!

TARTALOMJEGYZÉK

1. Bevezetés

1.1. Miért ezt a témát választottam?

Már az egyetemre jelentkezés előtt nagyon szerettem adatbázisokkal foglalkozni és mindig is érdekelt az adatok felhasználása, elemzése. Amikor először tanultunk a Big Data rendszerekről és NoSQL adatbázisokról, egyből felkeltette érdeklődésemet működésük, valamint az új lehetőségek kínálata, amit ezek a technológiák nyújtanak.

Továbbá nagyon megtetszett a témában a kutatás rész, hogy két különböző rendszert kell megtervezzek és felépítsek, majd a teljesítményét le kell mérjem, oly módon, hogy a két rendszer különböző erősségeit és gyengeségeit figyelembe veszem.

1.2. Milyen előnyt jelent és mi a célom?

Reményeim szerint az Apache Hadoop és Apache Sparkot magas szintű elsajátítását fogja biztosítani számomra, s ha valaki ezeket a rendszereket szeretné megismerni akkor hasznos olvasmány lesz számára a szakdolgozatom.

A szakdolgozatom célja, hogy a Hadoop és Sparkot összehasonlítsam és megállapítsam egyes helyzetekben melyiket érdemesebb használni.

2. SZOFTVEREK KIVÁLASZTÁSA

Fájlrendszernek a HDFS-t választom, mivel ezt a Spark és Hadoop is egyaránt támogatja, a teszt adatokat ide töltöm fel és a két program innen fogja beolvasni őket, majd itt tárolja el az eredményeket.

A batch feldolgozáshoz az Apache Hadoopot választottam a MapReducal. Ez a szoftver a saját kategóriája úttörőjeként jelent meg és a mai napig nagy népszerűségnek örvend. Mellesleg az Apache Foundation ökoszisztémájának egyik fő része, hogy sorra adnak ki olyan programokat, amelyekkel jól integrálható és az esetleges gyengeségeit így orvosolni tudják, az újabb követelményekhez igazodni tudnak. A Hadoop csak batch és iteratív feldolgozásra képes, stream feldolgozást nem támogat.

A stream feldolgozáshoz az Apache Spark programját választottam, a szoftvert pedig a Hadoopba integrálva fogom használni. Eleinte amikor kifejlesztették, az volt a cél, hogy a Hadoop iteratív feldolgozását javítsák, és hogy stream feldolgozást is lehetővé tegyenek. Azt viszont fontos megjegyezni, hogy nem a Hadoop leváltására hozták létre, hanem hogy együttműködjön a két szoftver és kiegészítsék egymást.

Ezt a két rendszert az egyetemen is tanultuk a Big Data algoritmusok tárgy keretében, s mivel nagyon felkeltették az érdeklődésemet, szerettem volna jobban megismerkedni velük.

3. Apache Hadoop és Apache Spark működése

Mindkét szoftver az Apache család része és nagy népszerűségnek örvendenek. Egyes kutatók úgy tekintenek rájuk, mint riválisprogramokra, de nem olyan egyszerű összehasonlítani őket, ugyanis vannak dolgok, amelyeket a két program azonosan hajt végre, de más területeken teljesen eltérően működnek. Ettől függetlenül a két program teljesítménye összehasonlítható, csak figyelembe kell vegyük a különbségeiket.

Mindkét keretrendszer a párhuzamos feldolgozás elvén működik. Ilyen keretrendszerek esetén a legfontosabb kérdés, hogy milyen eszközökkel dolgozzák fel a beérkező adatokat. [13]

3.1. Apache Hadoop keretrendszer áttekintése

Az Apache Hadoopra a Big Data elemzés sztenderdjeként tekinthetünk. Az elosztott és párhuzamos számítások által okozott kihívásokra tökéletes megoldásokat nyújt. Az Apache Hadoop keretrendszert úgy tervezték, hogy az adatokat batchekben dolgozza fel. Két fő komponensből áll: a Hadoop Distributed File Systemből és a MapReduceból. [14] A MapReduce a kernelje az Apache Hadoopnak, amit Hadoop MapReduce Frameworknek hívnak. A Hadoop klaszter gépek csoportja, ami HDFS-t és MapReducet futtatja, az egyes gépeket pedig a klaszterben csomópontoknak (node-nak) nevezzük. A csomópontok száma, azaz a klaszterben levő gépek száma, egyenesen arányos a teljesítménnyel, ami azt jelenti, hogy minél több gépünk van egy klaszterben, annál gyorsabban tud a klaszterünk adatokat feldolgozni. [15] A Hadoop az elosztott és párhuzamos számítás elvén működik, az adatokat szétosztja a klaszterben lévő csomópontok között, amelyek egyszerre dolgozzák fel az adatokat.

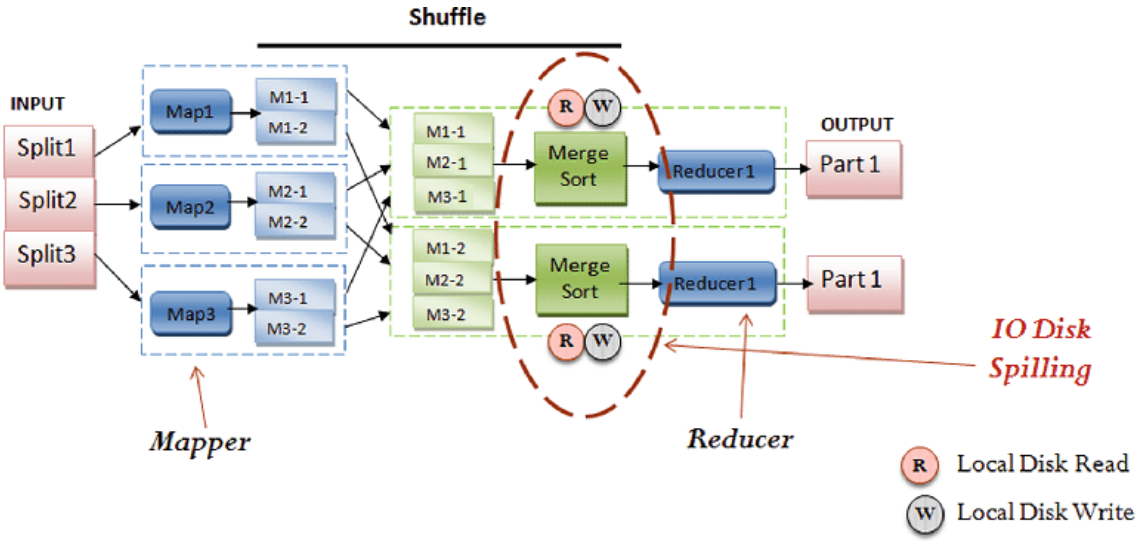
3.2. Apache Hadoop MapReduce

2004-ben a Google fejlesztette ki a MapReduce modellt, ami képes párhuzamosan feldolgozni nagy mennyiségű adatokat. Ezután létrejött az Apache Hadoop része, ami a HDFS-ből és a MapReduceból áll. [16] A MapReduce két fázisból áll: az első a Map fázis, a második a Reduce fázis.

A Map folyamat felelős a bemeneti adatok feldolgozásáért, amik a HDFS-ben vannak tárolva, fájl formájában. A Map folyamat átalakítja a bemeneti rekordokat egy közbenső állapotba. [17] A Hadoop keretrendszer meghív egy beállító metódust, ami csak egyszer hívódik meg, ezután pedig a Map metódus következik, amely minden egyes kulcs-érték pár után meghívódik egyszer és utána mindig meghívódik egy takarító metódus. [13]

A MapReduce támogatja a TextInputFormatot. A formátum sajátossága, hogy tartalmaz kulcsot Byte Offset formátumban és értéket Text formátumban. Egy word count feladatban a kulcs típusa Long Writeable és az érték Text formátumú, majd a Context engedni fogja a mappernek és a reducernek, hogy kommunikáljon a Hadoop rendszer többi részével. [13]

A Reduce fázisban az adatok feldolgozása történik. A rendszer fogja a közbenső kulcs érték párokat, amiket a mapper állított elő és előállítja belőlük a kimenetet, amit a HDFS-ben fog eltárolni. Ez a fázis összegzi az egész adathalmazt. [13]

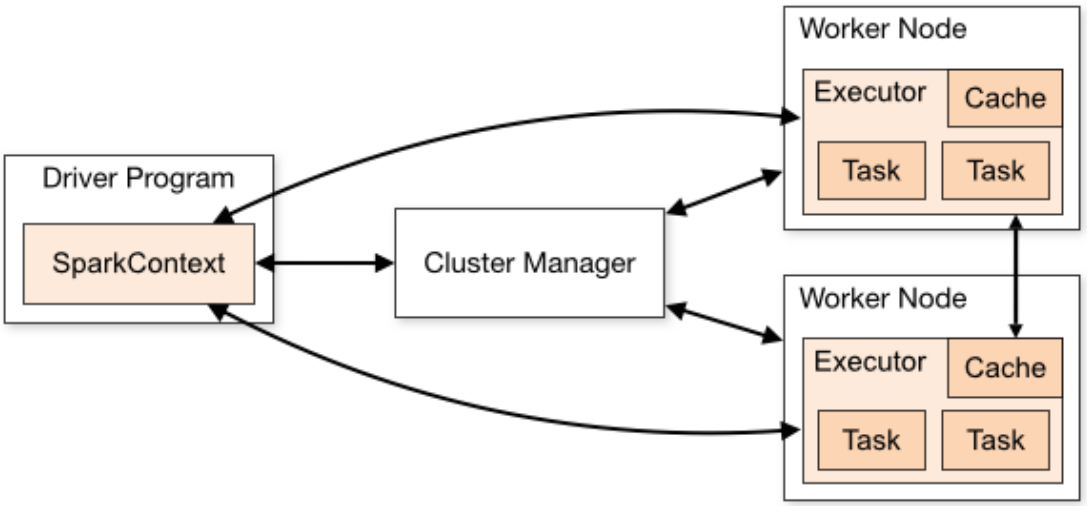


1. ábra: Map Reduce [18]

3.3. Apache Spark

Az Apache Spark azzal a céllal lett kifejlesztve, hogy az Apache Hadoop gyengeségeit kijavítsa. Az Apache Sparkot a kaliforniai Berkeley egyetem AMPLabjában fejlesztették ki és az első verzióját 2014-ben adták ki. [19] Az Apache Hadoop nagyon jó a batch alapú adatfeldolgozásban, de a teljesítménye romlik amikor iteratív módon kell feldolgozni az adatokat. Ennek az oka, hogy a Hadoop a részeredményeit a HDFS-en tárolja és a merevlemezre való írás és olvasás elég időigényes művelet a CPU-ra nézve. Ezzel ellentétben az Apache Spark a memórián belül tárolja a részeredményeket és ezért gyorsabb, mint a Hadoop. [20] Az Apache Spark a stream és batch adatfeldolgozást is támogatja. Az Apache Spark egy sajátos adatszerkezetet használ, amit RDD-nek nevezünk (Resilient Distributed Dataset). Az RDD-k pedig olyan rekordok gyűjteményei, amik partícionálva vannak és nem lehet őket szerkeszteni. [21] RDD-k a következő módokon jöhetnek létre:

- adatokat olvasunk a HDFS-ről
- a meglevő RDD-ken elvégzünk transzformációkat
- párhuzamosító metódust hívunk meg a már meglévő kollekciókon



2. ábra Apache Spark Cluster [22]

A Spark programok önálló feladatok csoportjaként futnak a clusteren, s ezeket a SparkContext irányítja a fő programban. A SparkContext többféle Cluster Managert összetud kötni, legyen akár a sajátja vagy Mesos, YARN, esetleg Kubernetes. Amint ezek között létrejött a kapcsolat, a Spark összegyűjti az executorokat a cluster egyes nodejain és ezek fogják a számításokat és az adatok eltárolását végrehajtani a programunkban, továbbá ez juttatja el a végrehajtandó programkódot az executorokhoz. Végül pedig a SparkContext taskokat küld az executorokhoz, amiket futtatniuk kell. [22]

Ennek az architektúrának számos előnye van:

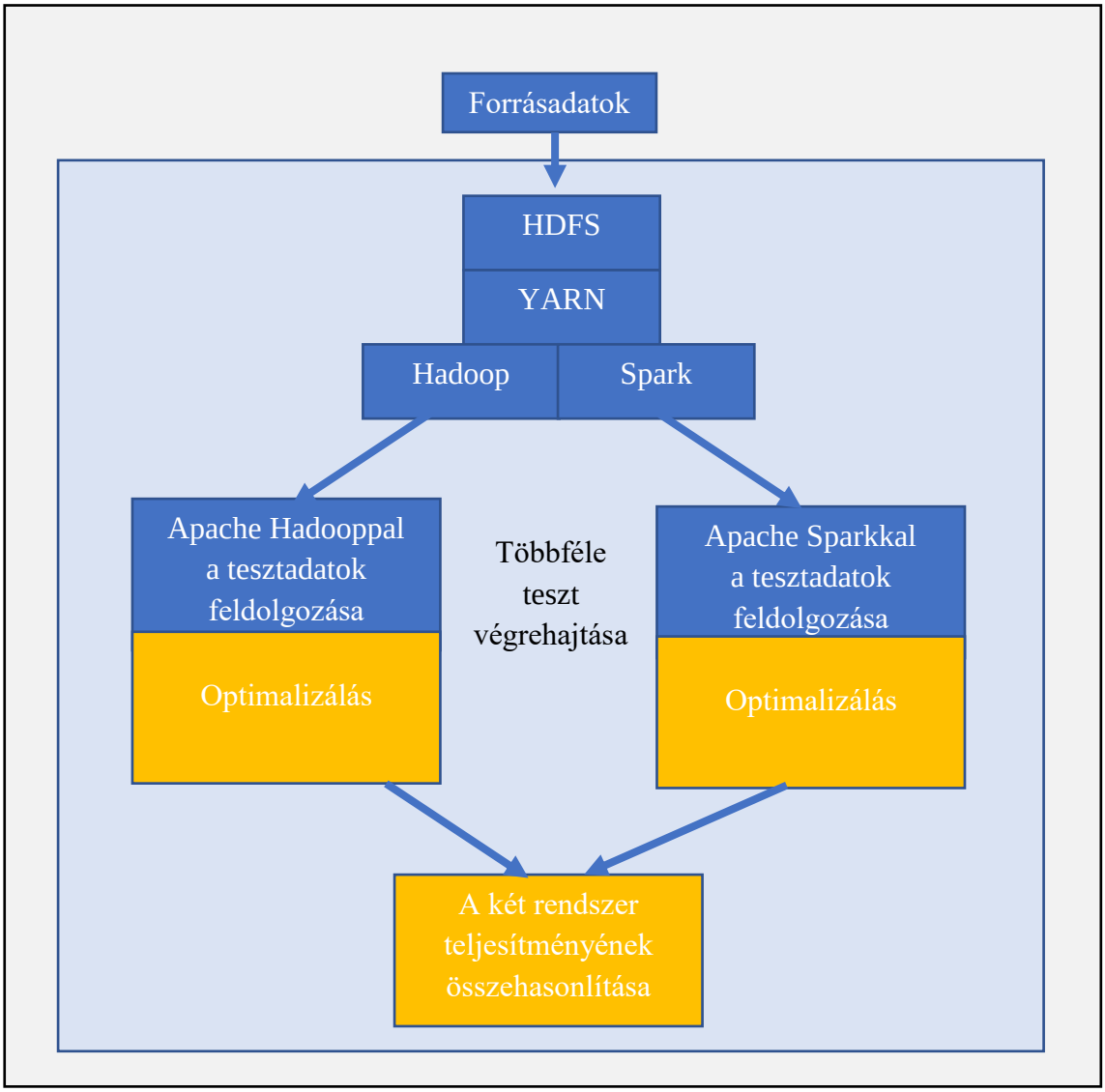
- Minden alkalmazás kap saját executor folyamatokat, amik az alkalmazás életciklusa végéig megmaradnak és több szálon futtatják a feladatot. Ennek az előnye, hogy az alkalmazásokat elkülönítjük egymástól időbeosztás terén (minden driver a saját feladatait ütemezi) és a végrehajtás terén is. Viszont ennek a hátránya, hogy az adatokat nem tudjuk megosztani különböző Spark alkalmazások között (SparkContext egyedek között) anélkül, hogy lemezre írnánk. [22]
- A Spark számos cluster menedzserrel kompatibilis. Ha megkapja a cluster menedzserektől az executor folyamatokat és ezek tudnak kommunikálni egymással, akkor nagyon könnyű a Sparkot futtatni akár olyan cluster menedzsereken is, amely más programokat is támogat. [22]
- A driver programnak mindig figyelnie és fogadnia kell a bejövő kapcsolatokat az executoroktól. A driver programnak a hálózaton mindig elérhetőnek kell lennie a worker nodeok számára. [22]
- Mivel a driver ütemezi a feladatokat a clusteren belül ezért fontos, hogy közel legyen a worker nodeokhoz. A legjobb, ha egy helyi hálózaton helyezkednek el. Ha távolról akarunk utasításokat küldeni a clusternek, akkor a legjobb, ha rácsatlakozunk távolról a gépre, ahol a driver fut és ott adjuk le az utasításokat. [22]

4. Apache Hadoop és Apache Spark összehasonlítása

Tulajdonságok	Apache Hadoop	Apache Spark
Adat feldolgozó motor	Mindkét rendszer alapja egy Batch feldolgozó motor.	
Támogatott nyelvek	Főleg Java	Java, Python, Scala, R
Milyen nyelven fejlesztették	A Hadoopot Java nyelven fejlesztették	A Sparkot Scala nyelven fejlesztették
Feldolgozási sebesség	A Hadoop általánosságban lassabb, mint a Spark	Egyes helyzetekben a Spark akár százszor gyorsabb is lehet a Hadoopnál
Iteratív feldolgozást támogat-e	Nem támogatja	Támogatja
Stream feldolgozást támogat-e	Nem támogatja	Támogatja
Hibatűrőség	Mindkét keretrendszer nagyon hibatűrő	
Adatok tárolása	A merevlemezen tárolja a feldolgozandó adatokat	A RAM-ban tárolja a feldolgozandó adatokat
Biztonság	A Sparknál fejlettebb biztonsági opciókat nyújt	A Sparknak a biztonsága még korai stádiumban van. Csak jelszavas hitelesítést nyújt

1. táblázat Hadoop és Spark összehasonlítása [13]

5. Rendszerterv



Az irodalomkutatás alapján ezt tartom a legjobb módszernek a stream és a batch adatfeldolgozás összehasonlítására.

5.1. Forrásadatok

A forrásadatokat a Lorem Ipsum generátorral hoztam létre a Word Count alapú feladatokhoz, a Page Rank feladathoz pedig a http://snap.stanford.edu/data/-ról töltöttem le. [30]

5.2. HDFS

A HDFS-ben tárolom majd el a forrásadatokat, amely a Hadoop és a Spark fájlrendszereként funkcionál. Az előbb említett szoftverek ezen keresztül fognak hozzáférni az adatokhoz és dolgozzák fel őket.

5.3. YARN

A YARN a Hadoop és a Spark erőforrás kezelője és a feladatok végrehajtásának az ütemezéséről is ez a szoftver gondoskodik.

5.4. Adatbetöltés optimalizálása

Hadoop optimalizálási lehetőségei: adat tömörítés, memóriakezelés és YARN konfiguráció, adat particionálás beállításainak megváltoztatása.

Spark optimalizálási lehetőségei: memóriakezelés megváltoztatása, adat particionálás megváltoztatása.

6. A KÉT RENDSZER TELJESÍTMÉNYÉNEK ÖSSZEHASONLÍTÁSA

A két rendszer teljesítményének összehasonlítása érdekében négy egyforma virtuális gépet hozok létre, amik egy clustert fognak alkotni. Mind a négy gépre telepítem a Hadoop és Spark programokat, s az előbb említett clusteren fogok három esettanulmányt elvégezni, majd dokumentálni ezek eredményeit.

6.1. Tesztkörnyezet

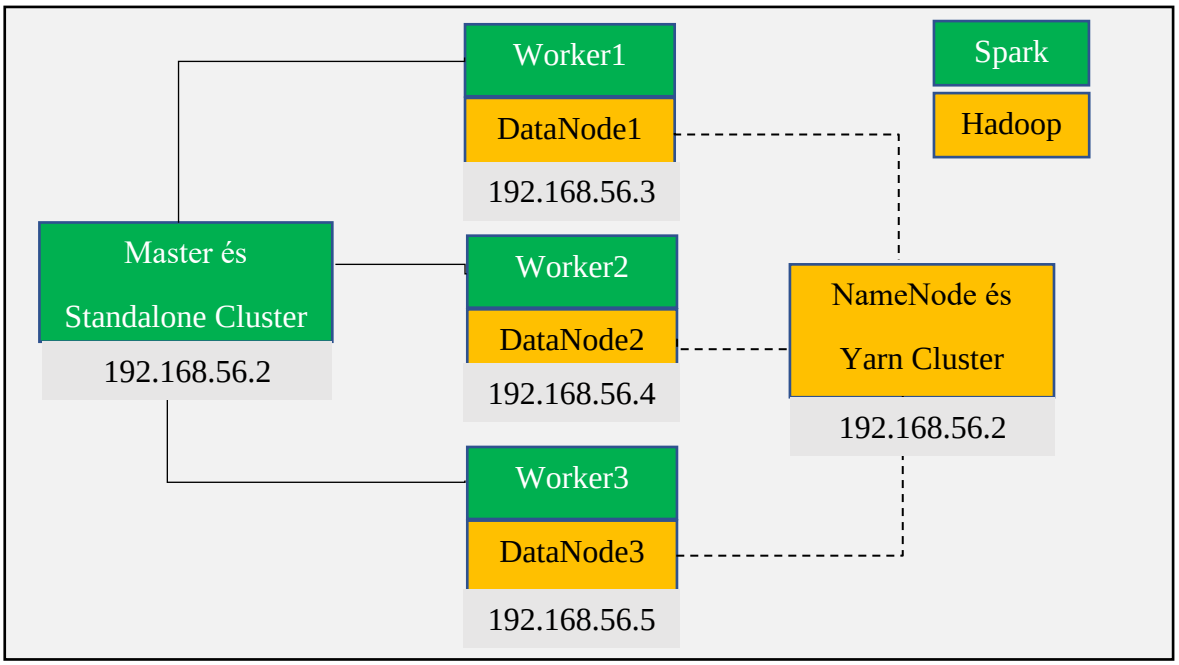
A következő fejezetben a tesztkörnyezet konfigurálását fogom lépésről lépésre bemutatni.

6.1.1. Hardver konfiguráció

Számítógép neve	IP cím	Processzor	Merevlemez
hadoop-master	192.168.56.2	Intel i7-9750H - 8 mag, 2,7 GHz	lokális SSD
hadoop-slave1	192.168.56.3		
hadoop-slave2	192.168.56.4		
hadoop-slave3	192.168.56.5		
Merevlemez mérete a gépeken: 50 GB Processzor magok száma a gépekben: 2 mag Memória mérete a gépeken: 2 GB DDR4 2666 Mhz			

6.1.2. Hálózati beállítások

A négy virtuális gép egy hálózatban található. A Hadoop rendszerben a hadoop-masteren fog a NameNode és a YARN Cluster Manager futni. A hadoop-slave[1-3]-on fognak futni a DataNode-ok. A Spark rendszerben a hadoop-masteren fog a Master Process és a Standalone Cluster futni. A hadoop-slave[1-3]-on fognak futni a Worker-ek.



6.1.3. Szoftver konfiguráció

	Szoftver neve
Operációs rendszer	Ubuntu 18.04 64 bit
Apache Hadoop	Hadoop 3.3.2
Apache Spark	Spark 3.2.1
Java Runtime Enviroment	OpenJDK Runtime Environment (build 1.8.0_312-8u312-b07-0ubuntu1~18.04-b07)
Virtualizációs szoftver	VirtualBox 6.1.32

6.1.4. Virtuális gépek előkészítés

A VirtualBox virtualizációs szoftvert a hivatalos weboldaláról (https://www.virtualbox.org/wiki/Downloads) letöltöm a 6.1.32-es változatot Windows 64 bites rendszerre, majd a C:\Program Files\Oracle\VirtualBox\ mappába installálom. A telepítés befejezése után elindítom a programot és létre fogom hozni a 4 darab virtuális gépet.

Létrehoztam egy új virtuális gépet, amire Ubuntu 18.04.06 LTS-t telepítettem, amelyet az adott operációs rendszer megbízhatósága és alacsony erőforrás igénye miatt választottam. A telepítés után pedig egyből frissítettem is a legújabb verziójára.

A gépeken két felhasználói fiókot hoztam létre: a „stampel” elnevezésű fiókban végeztem el a telepítést és a „hadoop” nevű fiókban futtattam a teszteket.

Felhasználónév	Jelszó
stampel	Jelszo123
hadoop	Jelszo123

Az IP címeket a 6.2.1-es pontban olvasható statikus IP címekre állítottam be, így biztosítva, hogy a clusteren belül a gépek zavartalanul kommunikálhassanak.

6.1.5. Hadoop telepítése

A telepítés megkezdése előtt frissítem az Ubuntu-t.

```
sudo apt install update
```

Fel kell installálni az SSH-t, hogy a gépek tudjanak egymással kommunikálni.

```
sudo apt install ssh
```

A gépek között beállítok jelszónélküli SSH kapcsolatot, ehhez le kell generálni rsa kulcs párokat. Fontos, hogy a -P után ne adjunk meg jelszót.

```
ssh-keygen -t rsa -P ""
```

```
stampel@master:~$ ssh-keygen -t rsa -P ""
Generating public/private rsa key pair.
Enter file in which to save the key (/home/stampel/.ssh/id_rsa):
Created directory '/home/stampel/.ssh'.
Your identification has been saved in /home/stampel/.ssh/id_rsa.
Your public key has been saved in /home/stampel/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:o3HRl8S8PzpUmi1hDPV9wy0QsXuZSMnnEalh+edNhjs stampel@master
The key's randomart image is:
+---[RSA 2048]----+
|           +*+.. |
|         ..o**oo.|
|        . .+B==++|
|         . o*=+=*|
|      . S  .oB=*.|
|       + .  =.E o|
|      .    . o o |
|            o    |
|             .   |
+----[SHA256]-----+
stampel@master:~$
```

Ezután fontos, hogy leellenőrizzük, hogy sikeresen létrejöttek-e a fájlok az SSH mappában. A következő képen láthatjuk a létrejött fájlokat.

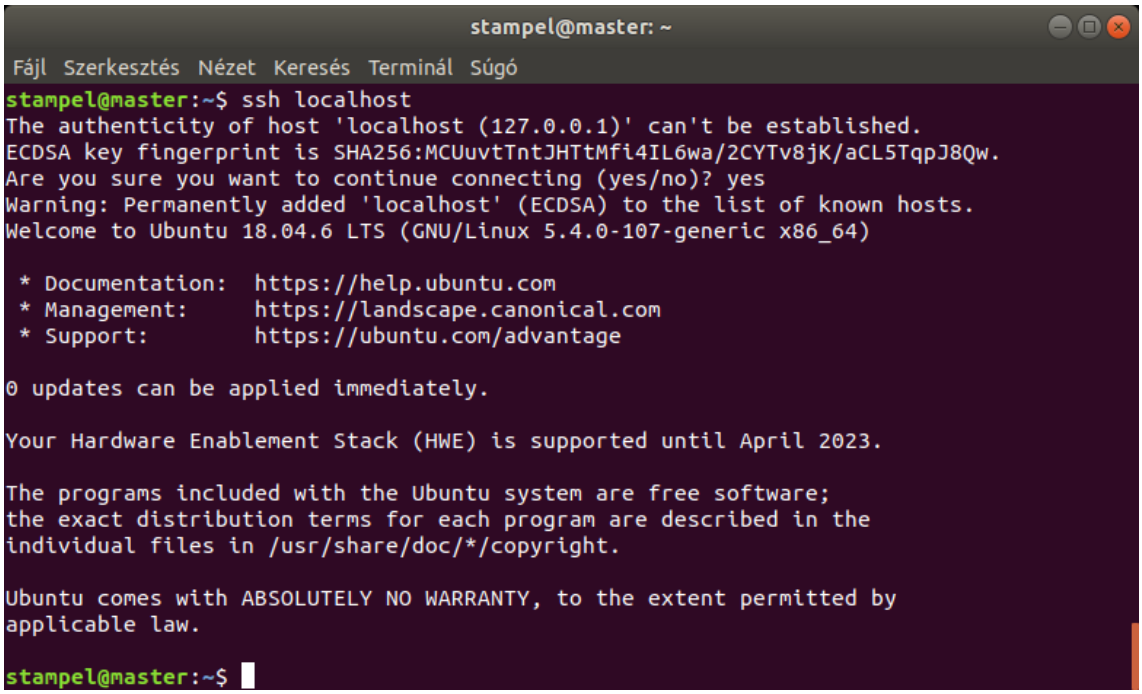
```
stampel@master: ~
Fájl  Szerkesztés  Nézet  Keresés  Terminál  Súgó
stampel@master:~$ ls -lrt .ssh/
összesen 8
-rw-r--r-- 1 stampel stampel  396 ápr   11 18:34 id_rsa.pub
-rw------- 1 stampel stampel 1679 ápr   11 18:34 id_rsa
stampel@master:~$
```

A publikus kulcsot átmásolom az authorized_keys mappába.

```
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

Megpróbálok rácsatlakozni a localhost-ra, ezzel letesztelve, hogy az SSH működik-e.

```
ssh localhost
```

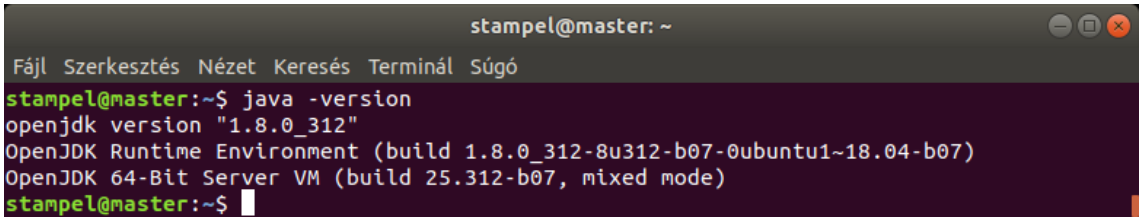


A Hadoopnak és a Sparknak is szüksége van a Javára, ezért ennek telepítése elengedhetetlen a programok megfelelő futtatása érdekében.

```
sudo apt install openjdk-8-jdk
```

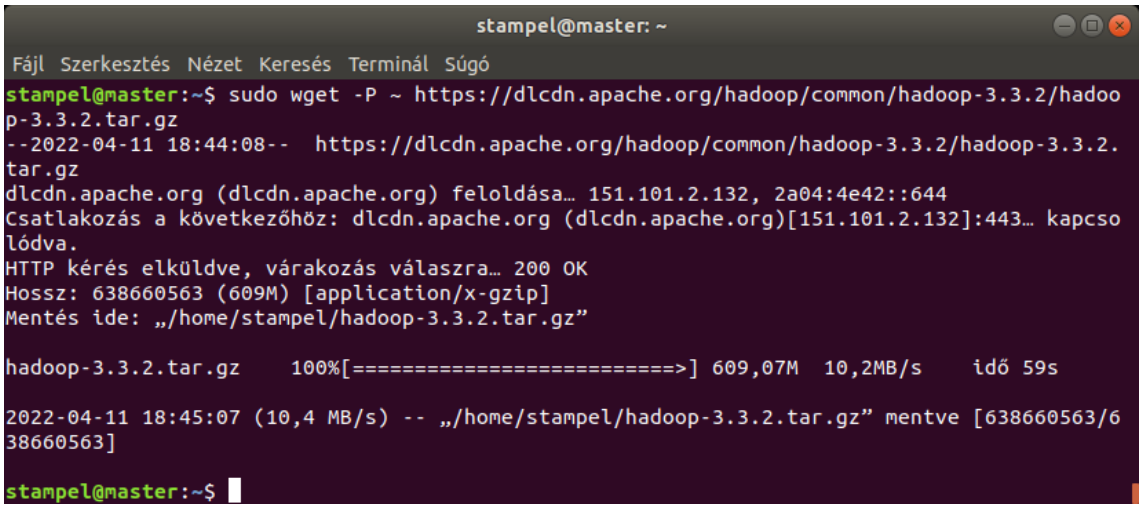
Az installálást követően le ellenőrzöm a Java verzióját, hogy sikeresen települt-e.

```
java -version
```



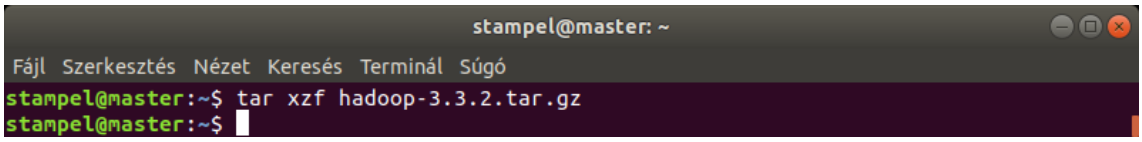
Letöltöm a Hadoop 3.3.2-t a hivatalos oldalukról.

```
sudo wget -P ~ https://dlcdn.apache.org/hadoop/common/hadoop-3.3.2/hadoop-3.3.2.tar.gz
```



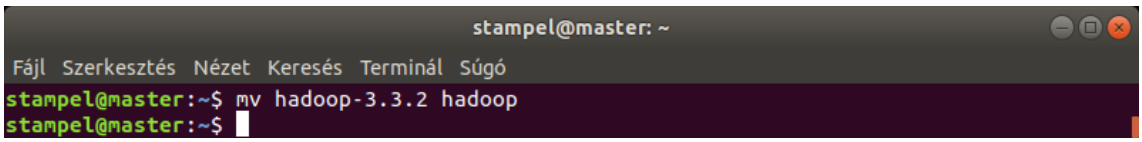
A letöltött fájlt ki kell csomagolni használat előtt.

```
tar xzf hadoop-3.3.2.tar.gz
```



Átnevezem a hadoop-3.3.2 mappát hadoop-ra a könnyebb elérés érdekében.

```
mv hadoop-3.3.2 hadoop
```

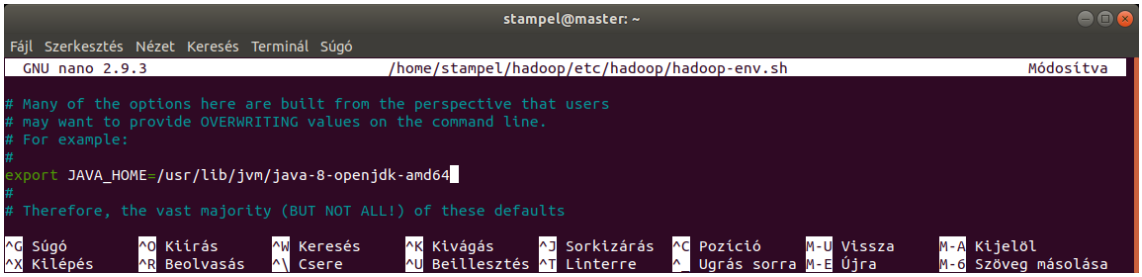


Megnyitom a hadoop-env.sh fájlt, hogy beállítsam a JAVA_HOME-t.

```
nano ~/hadoop/etc/hadoop/hadoop-env.sh
```

A fájl tartalmát a következő sorral bővítem.

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/
```



A hadoop mappát áthelyezem a /usr/local/hadoop mappába.

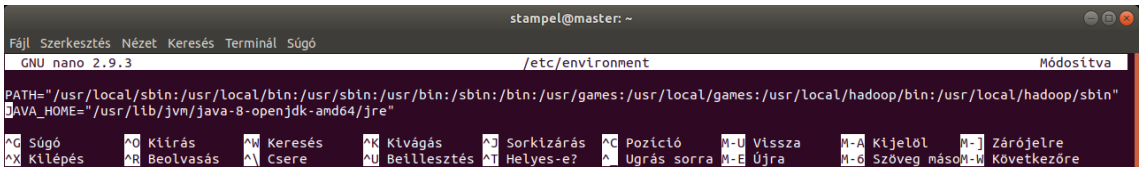
```
sudo mv hadoop /usr/local/hadoop
```

Ez után a Hadoop environment beállításait kell elvégeznem.

```
sudo nano /etc/environment
```

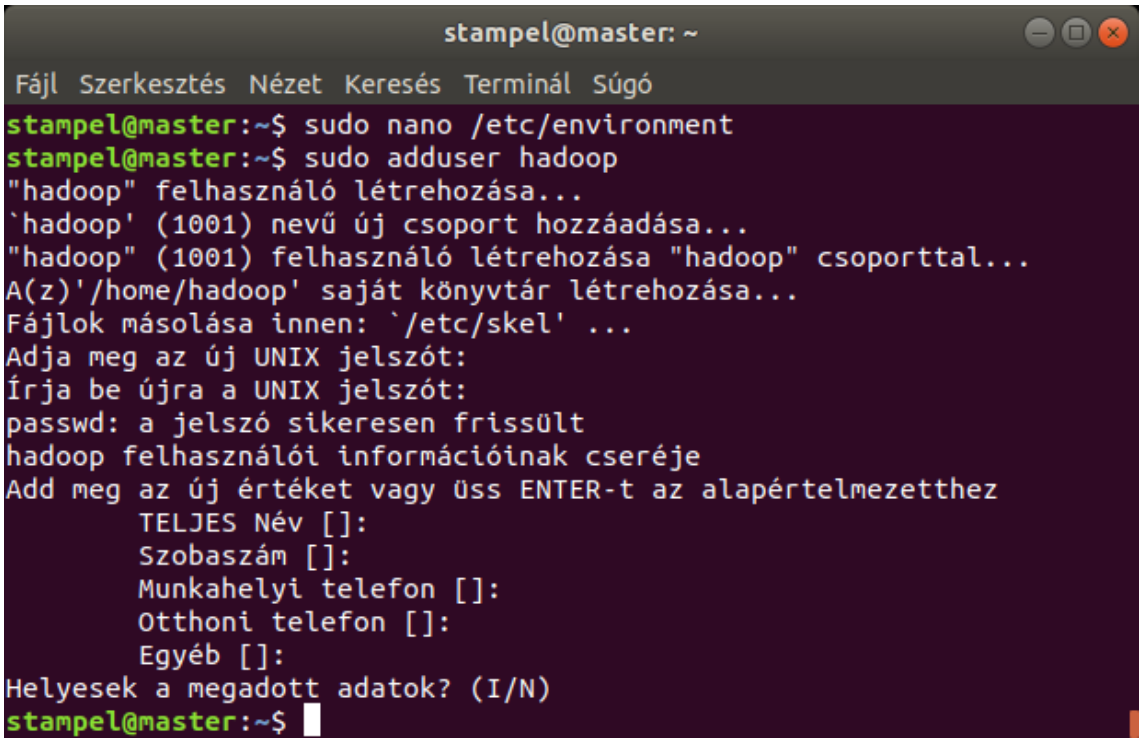
A tartalmát a következőre módosítom:

```
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/usr/lo-
cal/hadoop/bin:/usr/local/hadoop/sbin"

JAVA_HOME="/usr/lib/jvm/java-8-openjdk-amd64/jre"
```



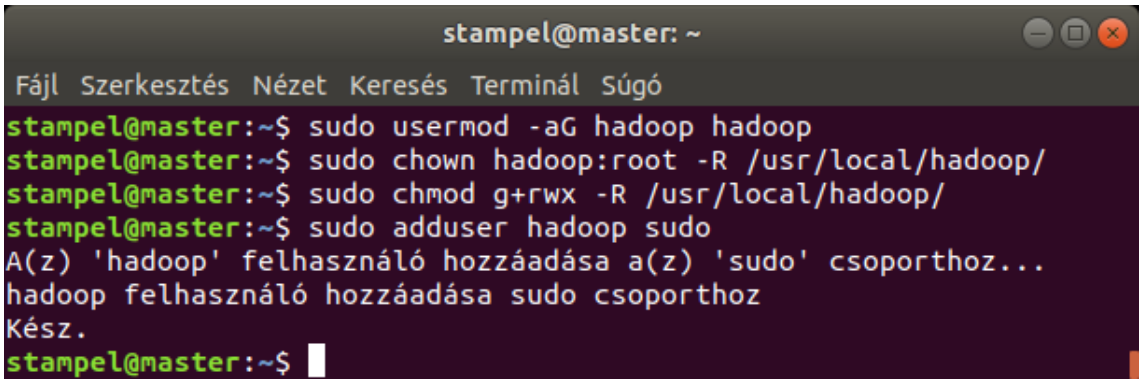
Létrehozok biztonsági okokból egy hadoop nevű felhasználót. Jelszó: Jelszo123

```
sudo adduser hadoop
```



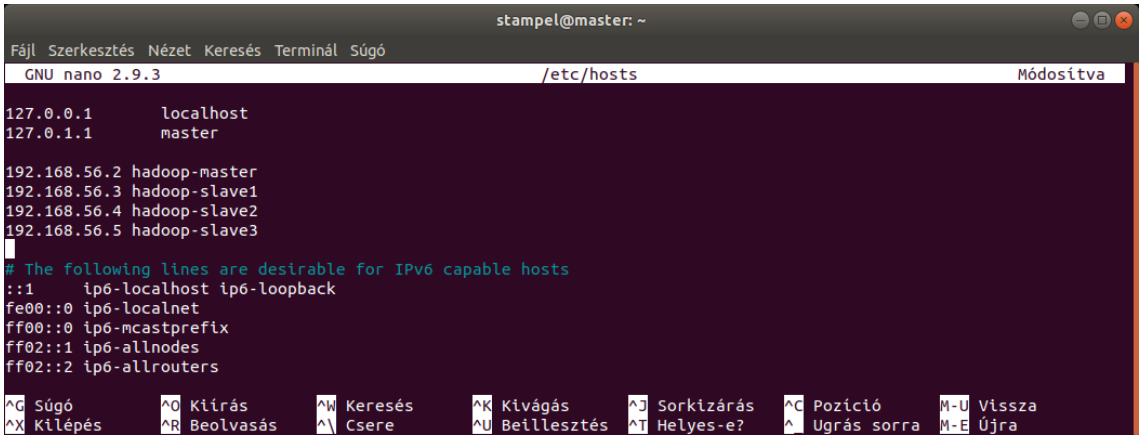
Az új felhasználónak elérést adok a hadoop mappához és sudo engedélyeket adok neki.

```
sudo usermod -aG hadoop hadoop

sudo chown hadoop:root -R /usr/local/hadoop/

sudo chmod g+rwx -R /usr/local/hadoop/

sudo adduser hadoop sudo
```

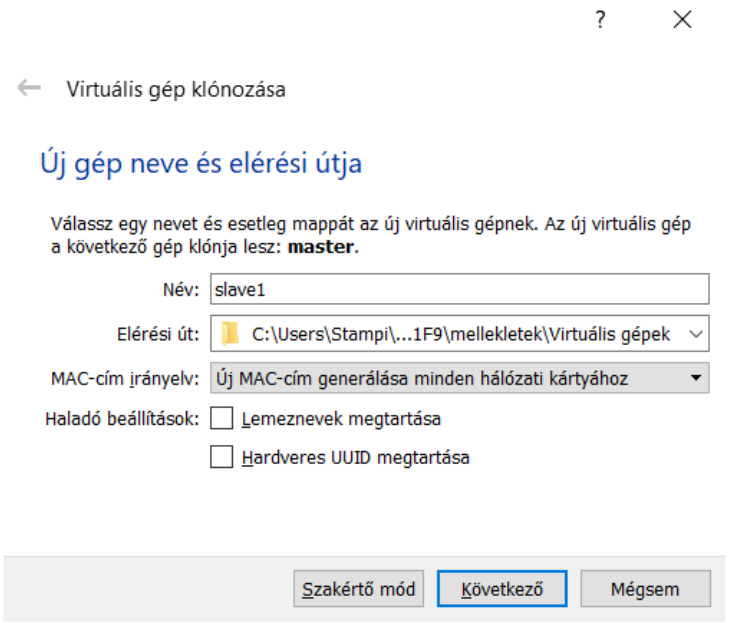


A hosts fájlba beleírom a gépek IP címeit, a 6.2.1-es pontból:

```
sudo nano /etc/hosts
```

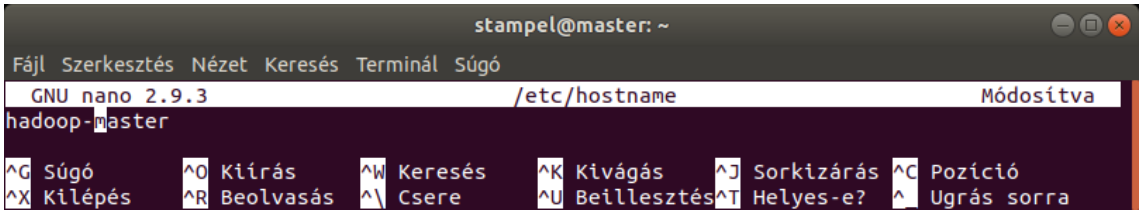


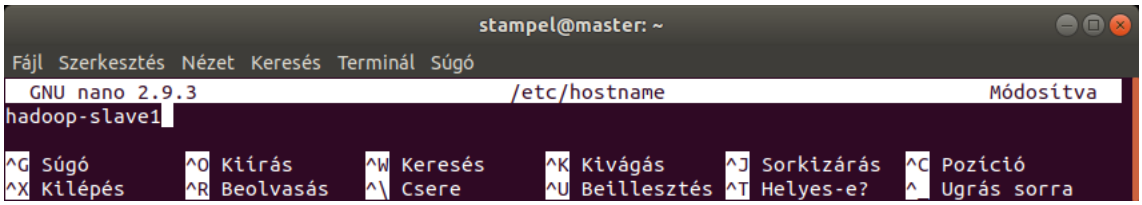
Most létrehozom a slaveket. Ehhez háromszor klónozom a mestert. Figyelek rá, hogy új MAC címet generáljon hozzájuk a VirtualBox.

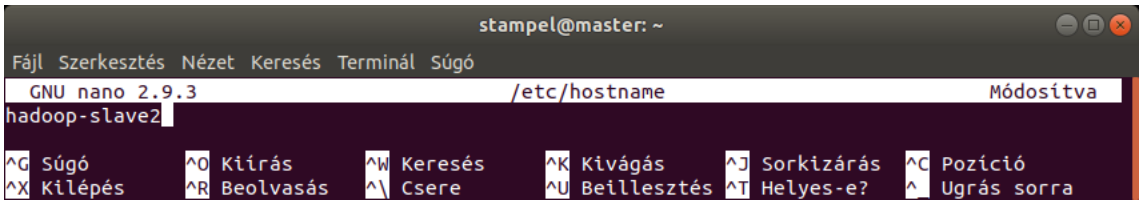


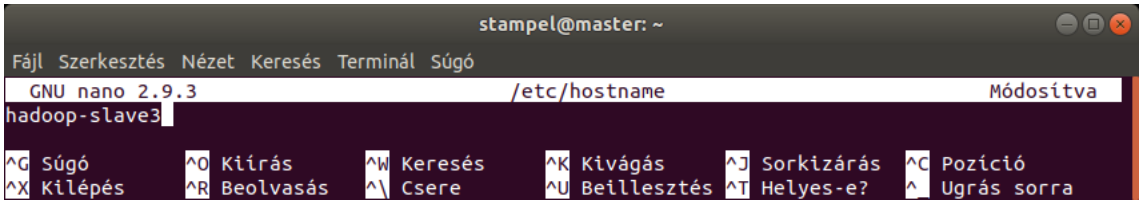
Minden gépen átírom a hostname fájlban a nevet.

```
sudo nano /etc/hostname
```









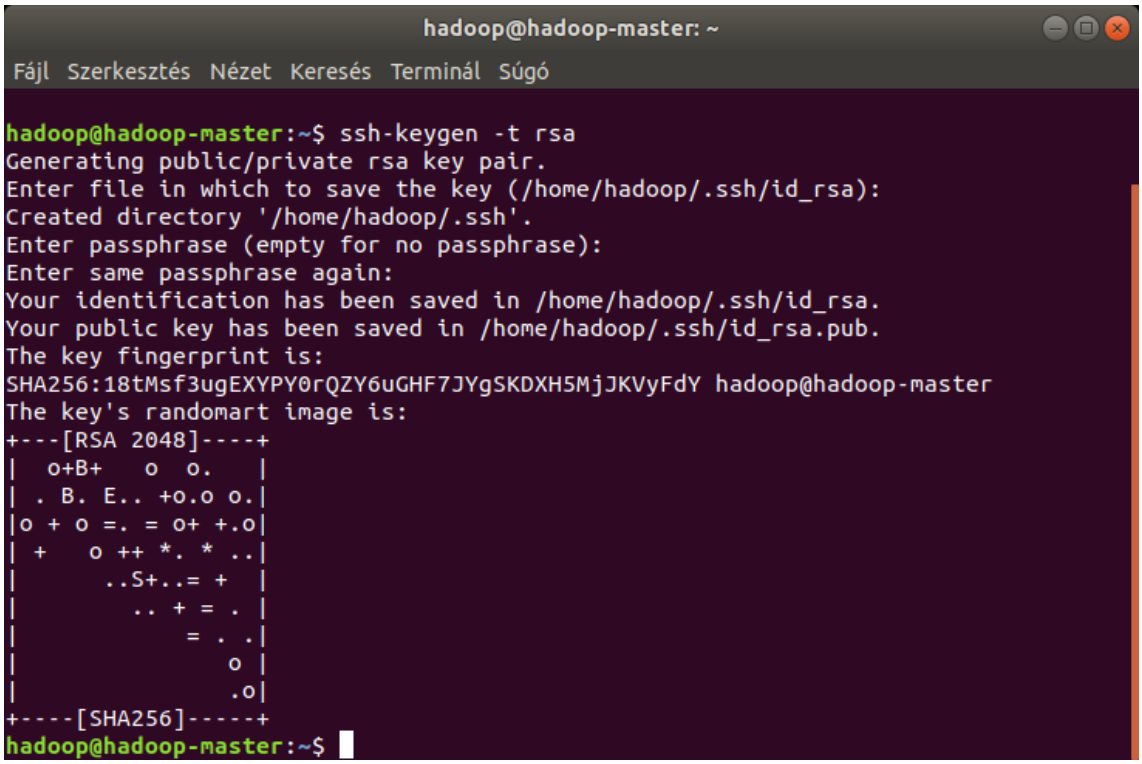
Ez után a gépek újraindítása szükséges.

Beállítom az SSH-t masteren, amelyhez a hadoop nevű felhasználót használom.

```
su – hadoop
```

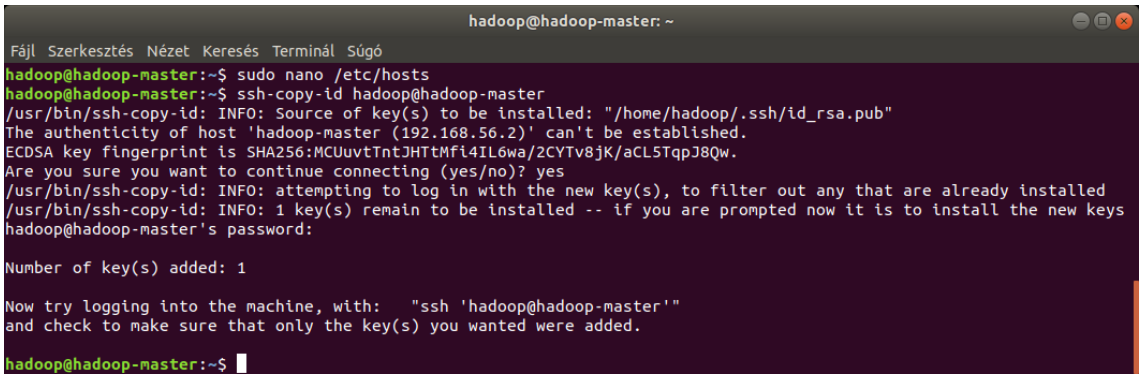
Létrehozom az SSH kulcsot:

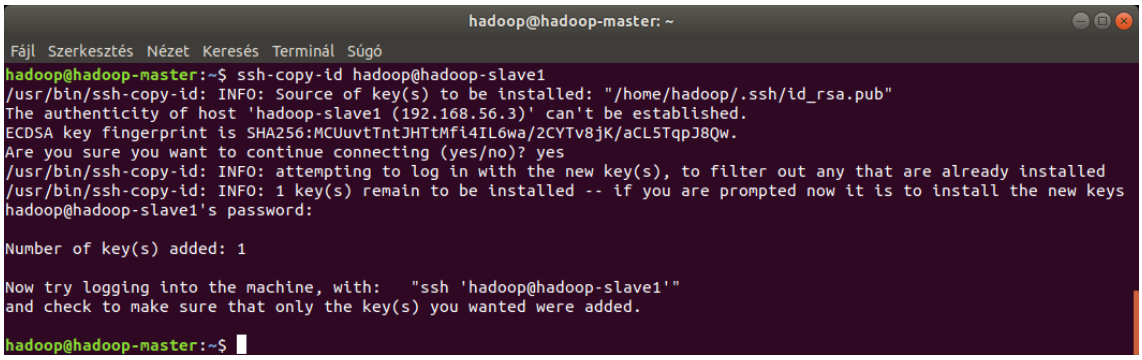
```
ssh-keygen -t rsa
```

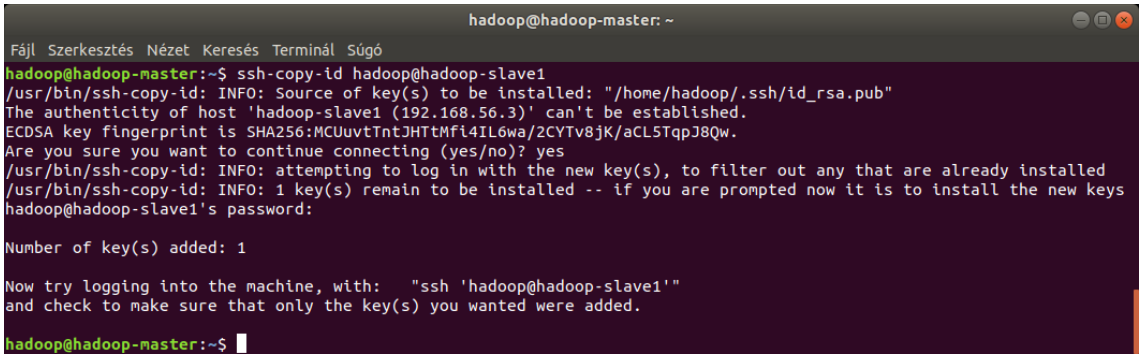


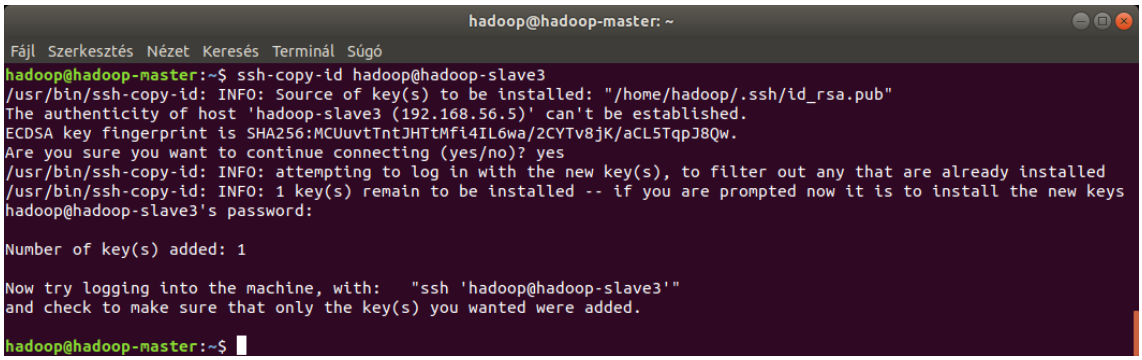
Ez után átmásolom az SSH kulcsot az összes másik gépekre.

```
ssh-copy-id hadoop@hadoop-master

ssh-copy-id hadoop@hadoop-slave1

ssh-copy-id hadoop@hadoop-slave2

ssh-copy-id hadoop@hadoop-slave3
```



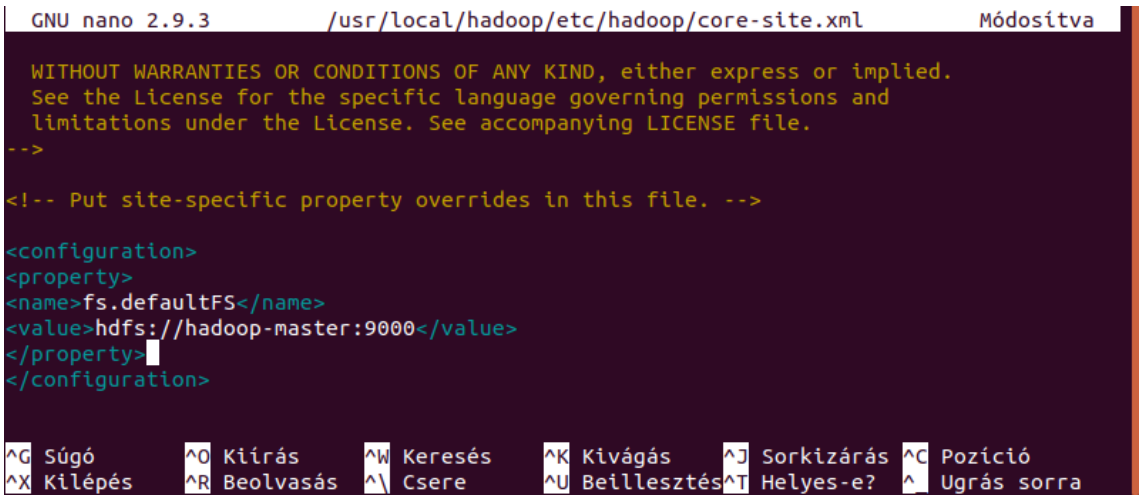






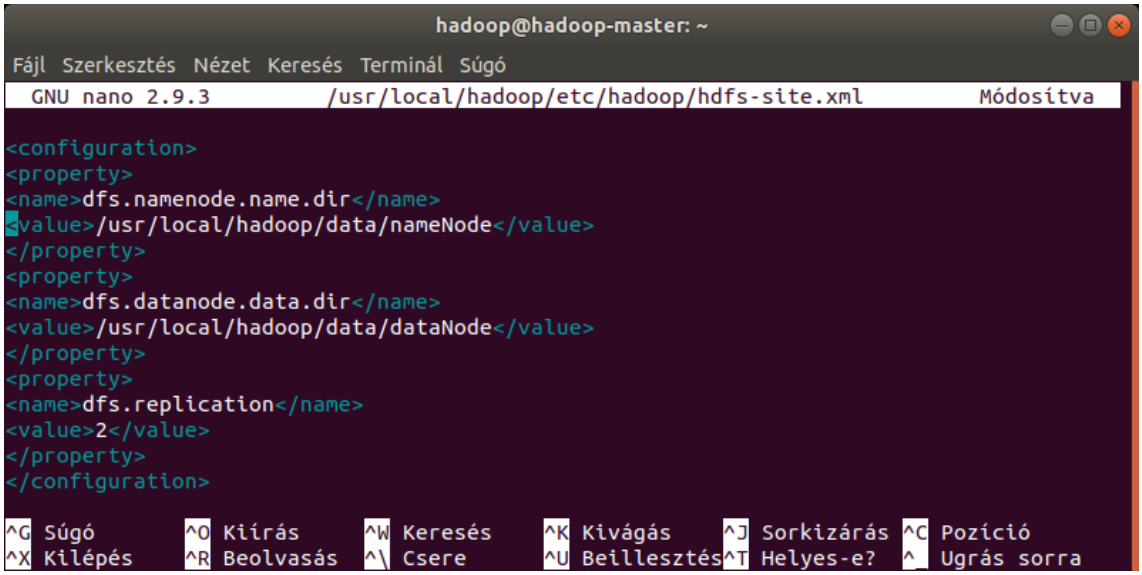
A mesteren megnyitom a core-site.xml-t és a következőket írom bele:

```
sudo nano /usr/local/hadoop/etc/hadoop/core-site.xml
```



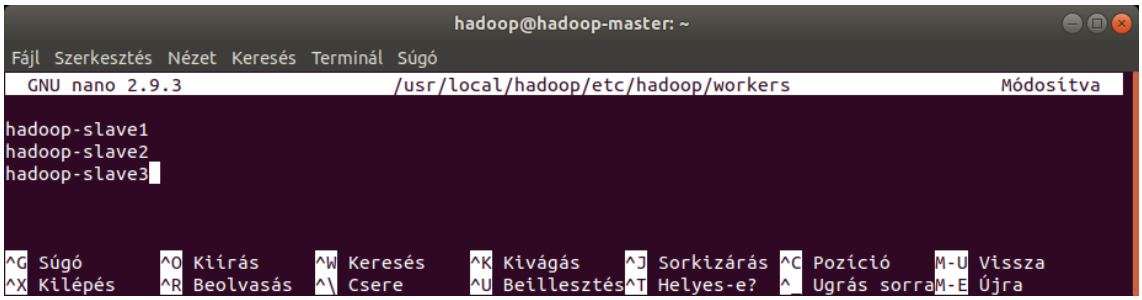
A mesteren még a hdfs-site.xml-ben el kell végezzek pár beállítást.

```
sudo nano /usr/local/hadoop/etc/hadoop/hdfs-site.xml
```



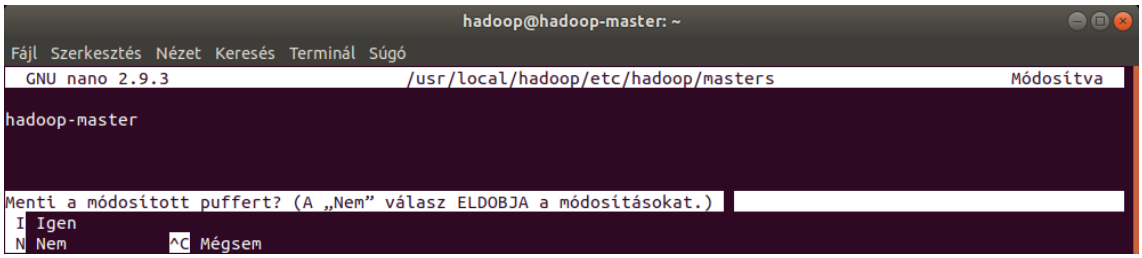
A masteren a workers fájlba a slave gépek host nevét be kell írni.

```
sudo nano /usr/local/hadoop/etc/hadoop/workers
```



A masteren megnyitom a master fájlt és beírom a saját host nevét.

```
sudo nano /usr/local/hadoop/etc/hadoop/masters
```



A mesteren a mapred-site.xml-t konfigurálásával folytatom a telepítést.

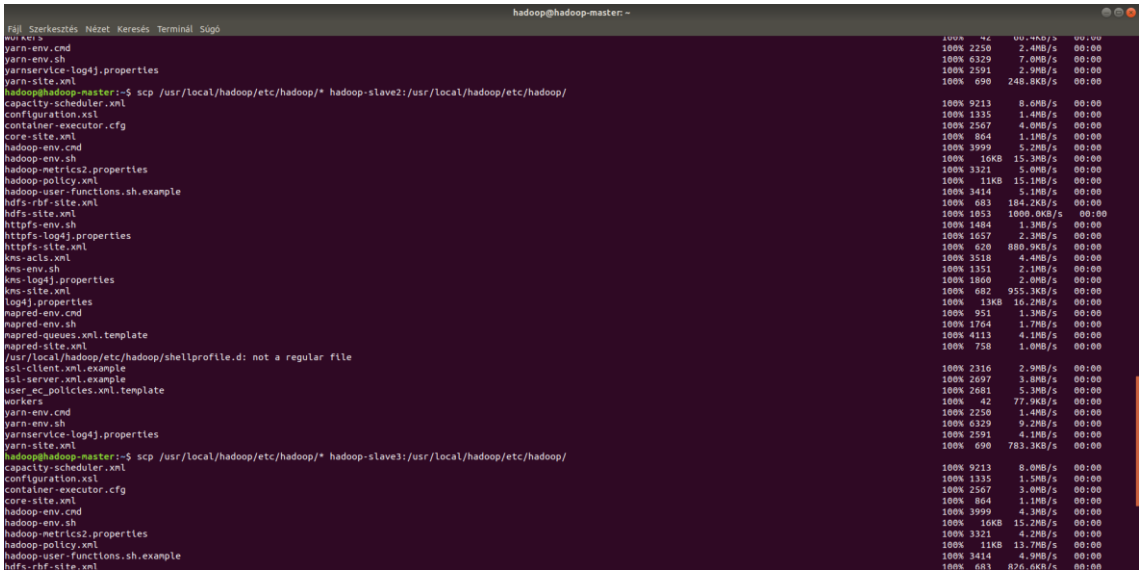
```
sudo nano /usr/local/hadoop/etc/hadoop/mapred-site.xml
```

A következő propertyket adtam hozzá a konfigurációs fájlhoz.

```
<configuration>
<property>
        <name>mapreduce.job.tracker</name>
        <value>hadoop-master:5431</value>
</property>
<property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
</property>

<property>
        <name>yarn.app.mapreduce.am.env</name>
        <value>HADOOP_MAPRED_HOME=${HADOOP_HOME}</value>
</property>
<property>
        <name>mapreduce.map.env</name>
        <value>HADOOP_MAPRED_HOME=${HADOOP_HOME}</value>
</property>
<property>
        <name>mapreduce.reduce.env</name>
        <value>HADOOP_MAPRED_HOME=${HADOOP_HOME}</value>
</property>
<property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
```

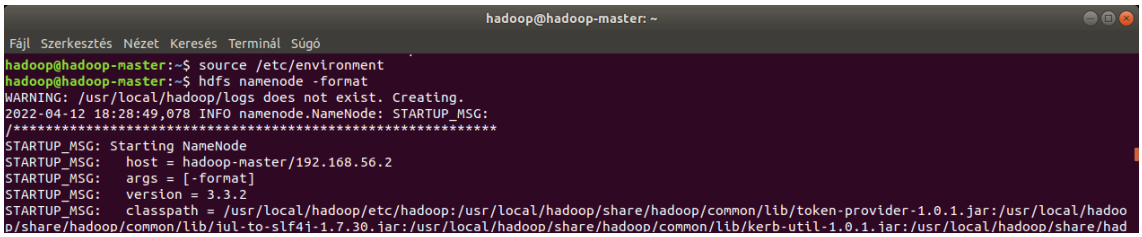
A Hadoop mester konfigurációját átmásolom a slavekre.

```
scp /usr/local/hadoop/etc/hadoop/* hadoop-slave1:/usr/local/hadoop/etc/hadoop/

scp /usr/local/hadoop/etc/hadoop/* hadoop-slave2:/usr/local/hadoop/etc/hadoop/

scp /usr/local/hadoop/etc/hadoop/* hadoop-slave3:/usr/local/hadoop/etc/hadoop/
```



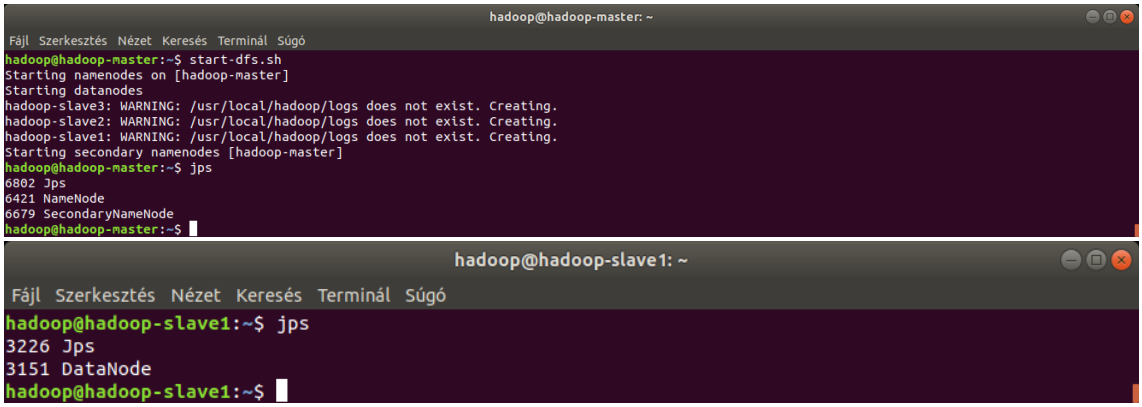
Megformázom a HDFS fájlrendszert.

```
source /etc/environment

hdfs namenode -format
```



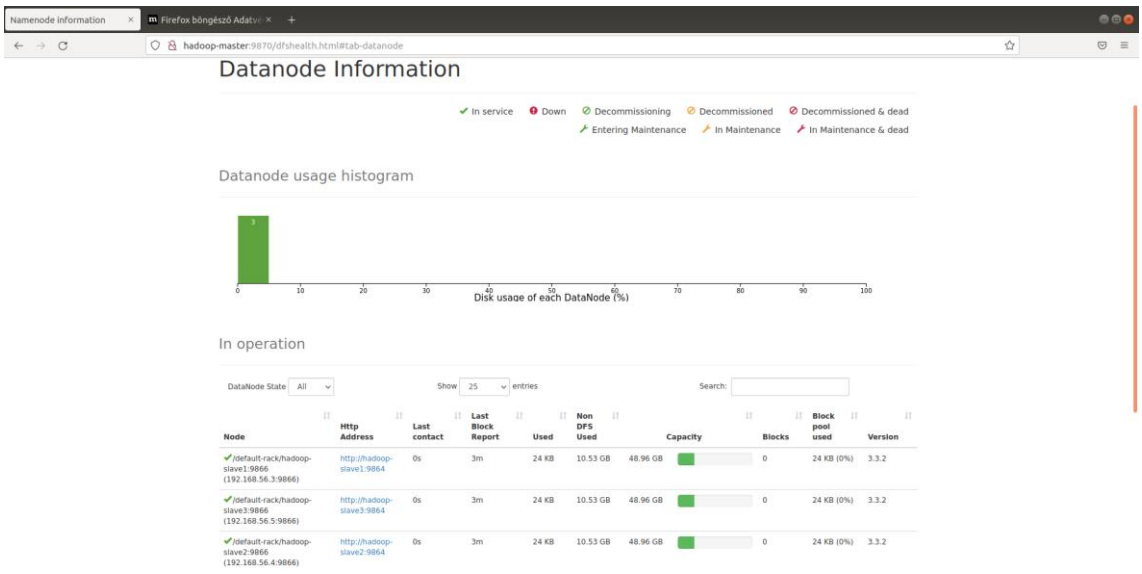
Elindítjuk a HDFS-t, majd ellenőrizzük a jps paranccsal, hogy megfelelően elindult-e.

```
start-dfs.sh

jps
```



Megállapíthatom, hogy mindegyik gépen megfelelően fut a Hadoop.

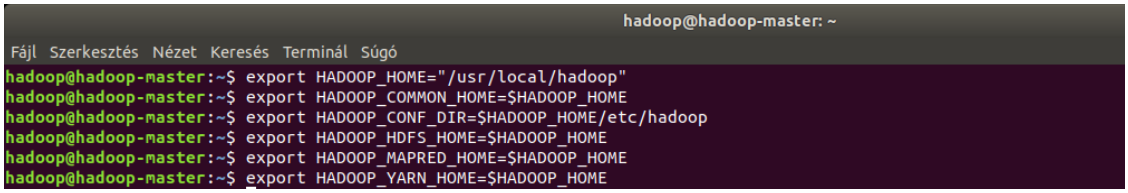
A mesteren megnézem, hogy a webes felülete betölt-e a Hadoopnak.



Ezután beállítom a YARN-ot.

```
export HADOOP_HOME=/usr/local/hadoop

export HADOOP_COMMON_HOME=$HADOOP_HOME
```

```
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop

export HADOOP_HDFS_HOME=$HADOOP_HOME

export HADOOP_MAPRED_HOME=$HADOOP_HOME

export HADOOP_YARN_HOME=$HADOOP_HOME
```

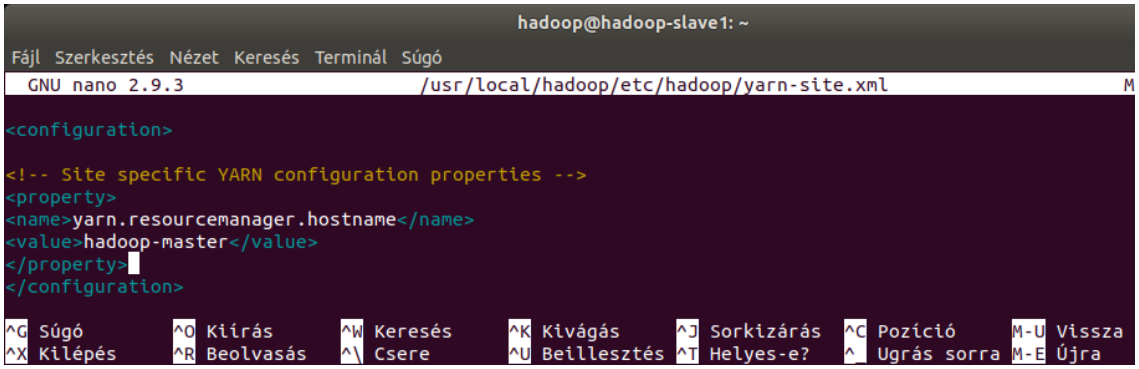


Mindhárom slaven konfigurálnom kell a yarn-site.xml-t.

```
sudo nano /usr/local/hadoop/etc/hadoop/yarn-site.xml
```

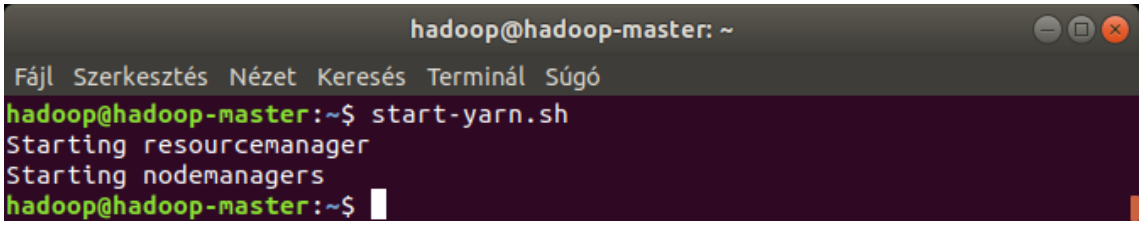
Hozzá kell adni egy propertyt, amit jelzi melyik gép lesz felelős az erőforrás kezelésért.

```
<property>

<name>yarn.resourcemanager.hostname</name>

<value>hadoop-master</value>

</property>
```

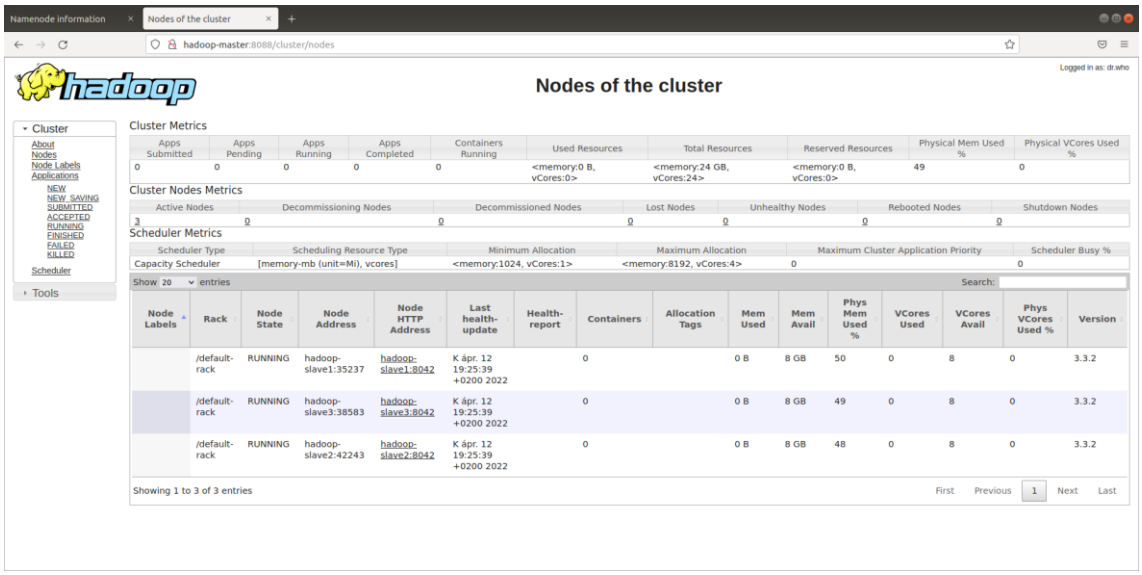


Elindítom a YARN-ot a clusteren.

```
start-yarn.sh
```



A hadoop-master:8088/cluster címen láthatjuk, hogy minden node tökéletesen elindult.



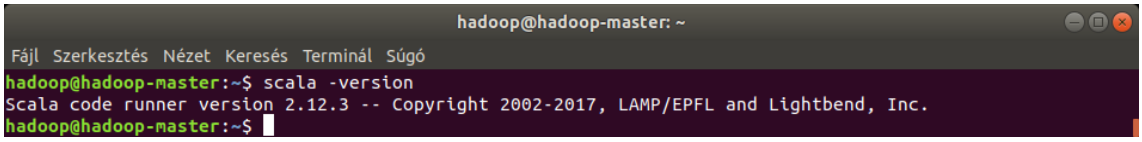
6.1.6. Spark telepítése

A Hadoop installálása során a Spark installálásának sok előkövetelményét teljesítettem. A master és slave gépek be vannak már állítva a hosts fájlban, a Java már fel van installálva, a hálózat és az SSH is konfigurálva lett. Az egyetlen előkövetelmény ami még el kell végezni, az a Scala feltelepítése az összes gépre.

```
sudo wget https://downloads.lightbend.com/scala/2.12.3/scala-2.12.3.deb

sudo dpkg -i scala-2.12.3.deb
```

A telepítés után ellenőrzöm, hogy megfelelően sikerült-e az installálás.

```
scala -version
```

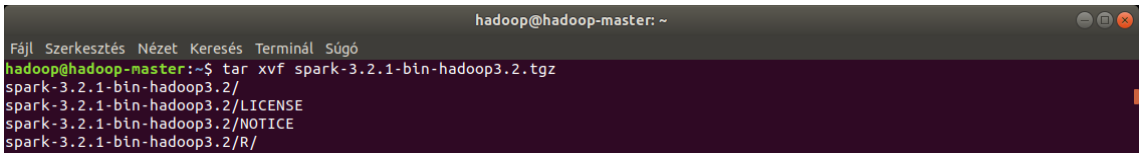


Letöltöm a Sparkot az összes gépre.

```
wget https://dlcdn.apache.org/spark/spark-3.2.1/spark-3.2.1-bin-hadoop3.2.tgz
```

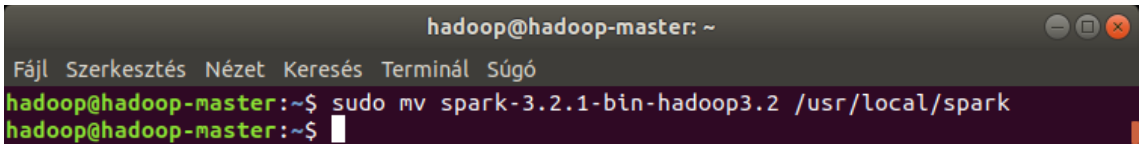
A letöltés befejezését követően ki kell csomagolnom.

```
tar xvf spark-3.2.1-bin-hadoop3.2.tgz
```



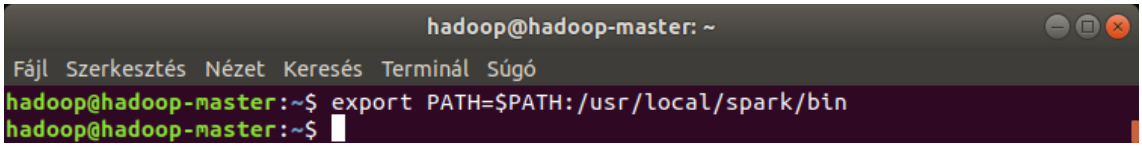
Mind a négy gépen áthelyezem a sparkot a /usr/local/spark mappába.

```
sudo mv spark-3.2.1-bin-hadoop3.2 /usr/local/spark
```



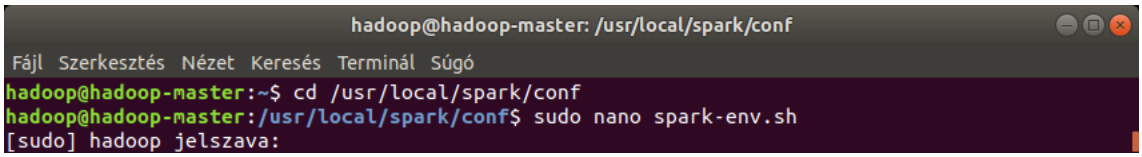
Beállítom a hozzá tartozó környezeti változót.

```
export PATH=$PATH:/usr/local/spark/bin
```



Mivel szeretnénk a HDFS-t használni ezért pár beállítást el kell végeznem a gépeken. A spark-env fájlhoz hozzá kell adjam a hadoop elérési útját és meg kell adni a Spark Master IP címét is az összes gépen. Először a beállításokat a masteren végzem el.

```
cd /usr/local/spark/conf

sudo nano spark-env.sh
```




```
HADOOP_CONF_DIR=/usr/local/hadoop/etc/hadoop

SPARK_MASTER_HOST=”192.168.56.2”
```

```
# Options read when launching programs locally with
# ./bin/run-example or ./bin/spark-submit
# - HADOOP_CONF_DIR, to point Spark towards Hadoop configuration files
HADOOP_CONF_DIR=/usr/local/hadoop/etc/hadoop
# - SPARK_LOCAL_IP, to set the IP address Spark binds to on this node
# - SPARK_PUBLIC_DNS, to set the public dns name of the driver program

# Options read by executors and drivers running inside the cluster
# - SPARK_LOCAL_IP, to set the IP address Spark binds to on this node
# - SPARK_PUBLIC_DNS, to set the public DNS name of the driver program
# - SPARK_LOCAL_DIRS, storage directories to use on this node for shuffle and RDD data
# - MESOS_NATIVE_JAVA_LIBRARY, to point to your libmesos.so if you use Mesos

# Options read in YARN client/cluster mode
# - SPARK_CONF_DIR, Alternate conf dir. (Default: ${SPARK_HOME}/conf)
# - HADOOP_CONF_DIR, to point Spark towards Hadoop configuration files
# - YARN_CONF_DIR, to point Spark towards YARN configuration files when you use YARN
# - SPARK_EXECUTOR_CORES, Number of cores for the executors (Default: 1).
# - SPARK_EXECUTOR_MEMORY, Memory per Executor (e.g. 1000M, 2G) (Default: 1G)
# - SPARK_DRIVER_MEMORY, Memory for Driver (e.g. 1000M, 2G) (Default: 1G)

# Options for the daemons used in the standalone deploy mode
# - SPARK_MASTER_HOST, to bind the master to a different IP address or hostname
SPARK_MASTER_HOST="192.168.56.2"
```

Ez után a beállítások fájlt átmásolom a Slavekre.

```
hadoop@hadoop-master: ~
Fájl  Szerkesztés  Nézet  Keresés  Terminál  Súgó
hadoop@hadoop-master:~$ scp -r /usr/local/spark/conf/spark-env.sh hadoop-slave1:/usr/local/spark/conf/
spark-env.sh
                                                      100% 4504     3.0MB/s   00:00
hadoop@hadoop-master:~$ scp -r /usr/local/spark/conf/spark-env.sh hadoop-slave2:/usr/local/spark/conf/
spark-env.sh
                                                      100% 4504     6.3MB/s   00:00
hadoop@hadoop-master:~$ scp -r /usr/local/spark/conf/spark-env.sh hadoop-slave3:/usr/local/spark/conf/
spark-env.sh
                                                      100% 4504     5.8MB/s   00:00
hadoop@hadoop-master:~$
```

A master gépen a slaves fájlba bele kell írjuk a slave gépek host nevét.

```
cd /usr/local/spark/conf

sudo nano slaves
```

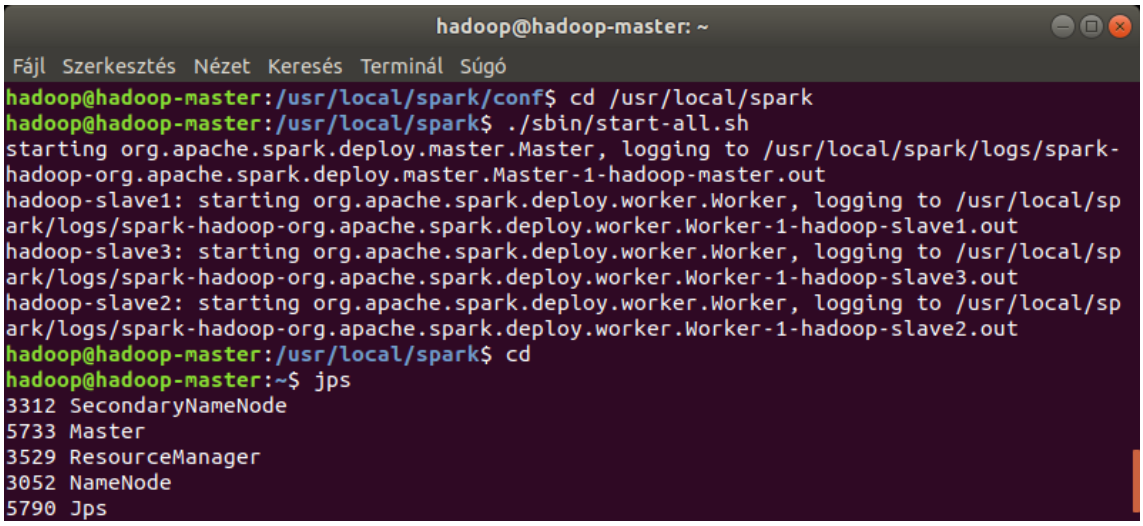
```
hadoop@hadoop-master: /usr/local/spark/conf
Fájl  Szerkesztés  Nézet  Keresés  Terminál  Súgó
  GNU nano 2.9.3                  slaves                       Módosítva

hadoop-slave1
hadoop-slave2
hadoop-slave3
```

Ez után már elindíthatjuk a Sparkot.

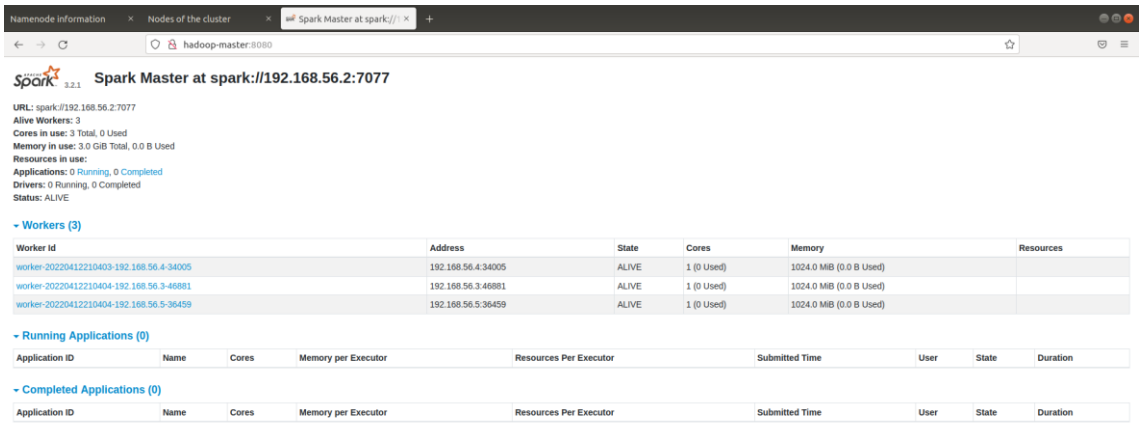
```
cd /usr/local/spark

./sbin/start-all.sh
```

JPS paranccsal leellenőrzőm, hogy minden gépen elindulta-e a kellő folyamatok.

```
jps
```



A hadoop-master:7077-en elérjük a Spark grafikus felületét.



6.2. Tesztek

A két rendszert futásidő alapján fogom összehasonlítani. Ahhoz, hogy az összehasonlítás pontos legyen a Hadoop és a Spark is ugyanazokon a gépeken futnak, mindkettő HDFS fájlrendszert használ és a teszt programokat ugyanazon a programozásinyelven írom meg.

A teszteket ötször fogom lefuttatni, hogy megbízható átlagos futásidőt kapjak. A tesztek eredményeit a Hadoop és a Spark webes felületéről mentem le a meresek.xlsx táblázatba és számítom ki az átlagokat.

6.2.1. Hadoop - WordCount

A WordCount algoritmus megszámolja, hogy egy fájlban az egyes szavak hányszor szerepelnek. A kimeneti fájlban a szavak ABC sorrendbe rendezve szerepelnek és mellettük szerepel az előfordulásuk száma.

```
Aenean    61600
Aliquam   78100
Class     8800
```

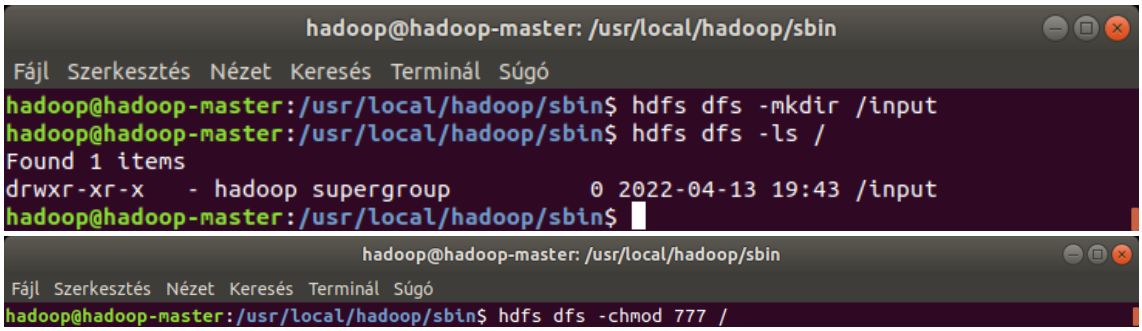
A következő táblázat tartalmazza a bemeneti fájlok nevét és a méretét. Ezeket a fájlokat Lorem Ipsum generátorral hoztam létre. Minden WordCount teszt alapjaként ezek a fájlok fognak szolgálni Hadoopban és Sparkban is.

Fájl neve	Fájl mérte
lorem100mb.txt	105 MB
lorem500mb.txt	527 MB
lorem1gb.txt	1,03 GB

A fájlokat a Dokumentumok közül felmásolom a HDFS-re.

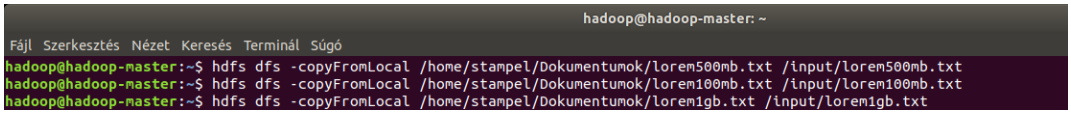
Létrehozok a HDFS-en egy input könyvtárat, ahova a fájlok kerülni fognak.

```
cd /usr/local/hadoop/sbin

hdfs dfs -mkdir /input

hdfs dfs -ls /

hdfs dfs -chmod 777 /
```

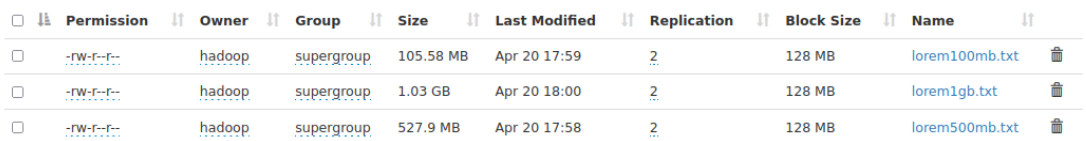


Ezután átmásolom a fájlokat az alábbi képen látható parancsokkal.

```
hdfs dfs -copyFromLocal /home/stampel/Dokumentumok/lorem100mb.txt /input/lorem100mb.txt

hdfs dfs -copyFromLocal /home/stampel/Dokumentumok/lorem500mb.txt /input/ lorem500mb.txt

hdfs dfs -copyFromLocal /home/stampel/Dokumentumok/lorem1gb.txt /input/ lorem1gb.txt
```

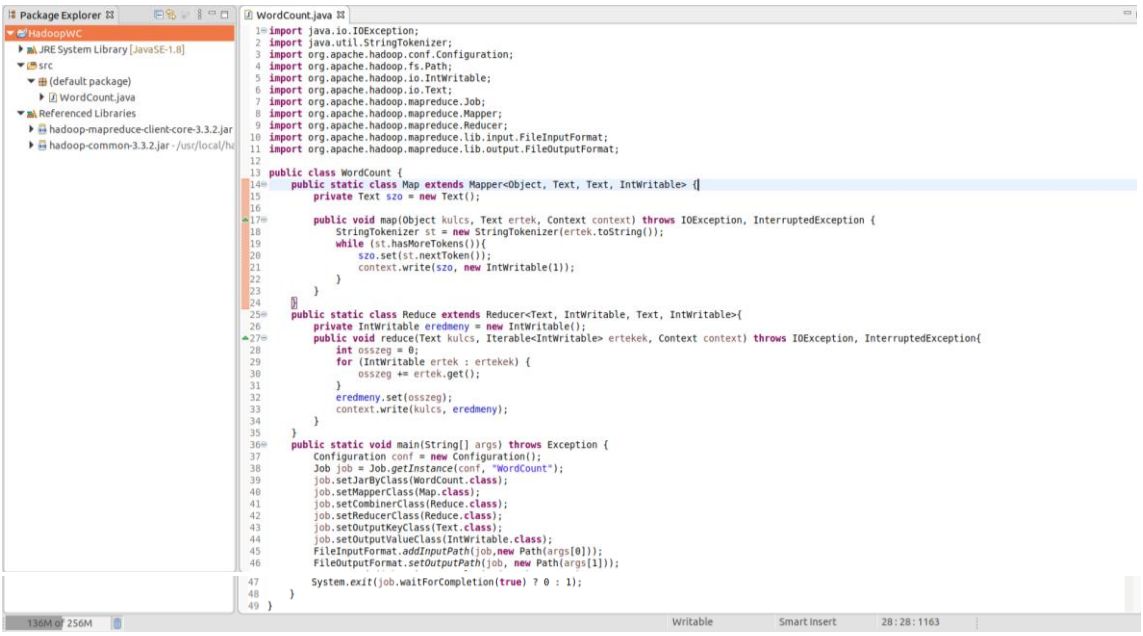


Ezután láthatjuk a HDFS-en is megjelentek a feltöltött fájlok:



Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	hadoop	supergroup	105.58 MB	Apr 20 17:59	2	128 MB	lorem100mb.txt
-rw-r--r--	hadoop	supergroup	1.03 GB	Apr 20 18:00	2	128 MB	lorem1gb.txt
-rw-r--r--	hadoop	supergroup	527.9 MB	Apr 20 17:58	2	128 MB	lorem500mb.txt

Eclipseben írtam meg a Hadoop Word Count implementációt Java nyelven.



A programot JAR-ként exportálom a /usr/local/hadoop/userscripts mappába.

A teszteket az alábbi parancsokkal futtatom le az alapértelmezett beállításokkal. Először a 100 MB-os fájlt fogom feldolgozni, utána az 500 MB-os fájlt, végül az 1GB-ost. A tesztek eredményét a webes UI-ról olvasom le és Excelbe mentem el.

100 MB-os teszt:

```
hadoop jar /usr/local/hadoop/userscripts/HadoopWC.jar /input/lorem100mb.txt /lorem_100mb_noopt_1
hadoop jar /usr/local/hadoop/userscripts/HadoopWC.jar /input/lorem100mb.txt /lorem_100mb_noopt_2
hadoop jar /usr/local/hadoop/userscripts/HadoopWC.jar /input/lorem100mb.txt /lorem_100mb_noopt_3
hadoop jar /usr/local/hadoop/userscripts/HadoopWC.jar /input/lorem100mb.txt /lorem_100mb_noopt_4
hadoop jar /usr/local/hadoop/userscripts/HadoopWC.jar /input/lorem100mb.txt /lorem_100mb_noopt_5
```

500 MB-os teszt:

```
hadoop jar /usr/local/hadoop/userscripts/HadoopWC.jar /input/lorem500mb.txt /lorem_500mb_noopt_1
hadoop jar /usr/local/hadoop/userscripts/HadoopWC.jar /input/lorem500mb.txt /lorem_500mb_noopt_2
hadoop jar /usr/local/hadoop/userscripts/HadoopWC.jar /input/lorem500mb.txt /lorem_500mb_noopt_3
hadoop jar /usr/local/hadoop/userscripts/HadoopWC.jar /input/lorem500mb.txt /lorem_500mb_noopt_4
hadoop jar /usr/local/hadoop/userscripts/HadoopWC.jar /input/lorem500mb.txt /lorem_500mb_noopt_5
```

application_1650472136404_0005	hadoop	WordCount	MAPREDUCE	default	0	Wed Apr 20 18:36:16 +0200 2022	Wed Apr 20 18:36:16 +0200 2022	Wed Apr 20 18:37:18 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A
application_1650472136404_0004	hadoop	WordCount	MAPREDUCE	default	0	Wed Apr 20 18:35:15 +0200 2022	Wed Apr 20 18:35:15 +0200 2022	Wed Apr 20 18:36:04 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A
application_1650472136404_0003	hadoop	WordCount	MAPREDUCE	default	0	Wed Apr 20 18:33:47 +0200 2022	Wed Apr 20 18:33:48 +0200 2022	Wed Apr 20 18:34:43 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A
application_1650472136404_0002	hadoop	WordCount	MAPREDUCE	default	0	Wed Apr 20 18:30:59 +0200 2022	Wed Apr 20 18:30:59 +0200 2022	Wed Apr 20 18:32:11 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A
application_1650472136404_0001	hadoop	WordCount	MAPREDUCE	default	0	Wed Apr 20 18:29:22 +0200 2022	Wed Apr 20 18:29:27 +0200 2022	Wed Apr 20 18:30:34 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A

1 GB-os teszt:

```
hadoop jar /usr/local/hadoop/userscripts/HadoopWC.jar /input/lorem1gb.txt /lorem_1gb_noopt_1

hadoop jar /usr/local/hadoop/userscripts/HadoopWC.jar /input/lorem1gb.txt /lorem_1gb_noopt_2

hadoop jar /usr/local/hadoop/userscripts/HadoopWC.jar /input/lorem1gb.txt /lorem_1gb_noopt_3

hadoop jar /usr/local/hadoop/userscripts/HadoopWC.jar /input/lorem1gb.txt /lorem_1gb_noopt_4

hadoop jar /usr/local/hadoop/userscripts/HadoopWC.jar /input/lorem1gb.txt /lorem_1gb_noopt_5
```

application_1650472863812_0005	hadoop	WordCount	MAPREDUCE	default	0	Wed Apr 20 19:25:47 +0200 2022	Wed Apr 20 19:25:47 +0200 2022	Wed Apr 20 19:29:18 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A
application_1650472863812_0004	hadoop	WordCount	MAPREDUCE	default	0	Wed Apr 20 19:22:21 +0200 2022	Wed Apr 20 19:22:21 +0200 2022	Wed Apr 20 19:24:45 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A
application_1650472863812_0003	hadoop	WordCount	MAPREDUCE	default	0	Wed Apr 20 19:17:30 +0200 2022	Wed Apr 20 19:17:30 +0200 2022	Wed Apr 20 19:20:18 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A
application_1650472863812_0002	hadoop	WordCount	MAPREDUCE	default	0	Wed Apr 20 18:47:59 +0200 2022	Wed Apr 20 18:47:59 +0200 2022	Wed Apr 20 18:51:40 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A
application_1650472863812_0001	hadoop	WordCount	MAPREDUCE	default	0	Wed Apr 20 18:41:34 +0200 2022	Wed Apr 20 18:41:36 +0200 2022	Wed Apr 20 18:46:04 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A

A lefutott tesztek futás idejének az átlaga a következő táblázatban látható.

Hadoop Word Count optimalizálás nélkül			
Adat mérete	100MB	500MB	1GB
Átlagos futásidő	33 mp	1 perc 1 mp	3 perc 22 mp

6.2.2. Hadoop – Word Count és többszöri rendezés

Ez a tesztet három részfeladat alkotja. Az első feladat a szavak megszámlálása, amelyhez az előbbi algoritmus kódját fogom felhasználni. Miután megkapjuk az eredményeket a kulcsokat (szavakat) és értékeket (darabszámok) meg kell cserélnünk, azaz a kulcsok a számok lesznek és az értékek a szavak, itt még fontos, hogy sorba rendezzük az előfordulások alapján. A harmadik feladat az lesz, hogy a darabszám szerint csoportosítsuk és a csoporton belül a szavakat rendezzük ABC sorrendbe. A következő képen látható a kapott kimenet.

```
165000 vitae
159500 id
158400 a
157300 quis
157300 sed
152900 eget
152900 nec
```

A tesztelés során az előző feladatban felhasznált tesztfájlokat használom fel újra.

Eclipseben írtam meg a Hadoop Word Count Secondary Sort implementációt Java nyelven.

JAR-ként exportálom a /usr/local/hadoop/userscripts mappába.

WordCount.java | WCSecondarySort.java

```
1  import java.util.*;
2  import java.io.*;
3  import org.apache.hadoop.fs.FileSystem;
4  import org.apache.hadoop.io.*;
5  import org.apache.hadoop.conf.Configuration;
6  import org.apache.hadoop.fs.Path;
7  import org.apache.hadoop.mapreduce.lib.input.*;
8  import org.apache.hadoop.mapreduce.lib.output.*;
9  import org.apache.hadoop.mapreduce.*;
10 import org.apache.hadoop.mapreduce.Partitioner;
11 import org.apache.hadoop.mapreduce.lib.partition.HashPartitioner;
12 import org.apache.hadoop.mapreduce.lib.map.InverseMapper;
13
14 public class WCSecondarySort {
15
16     public static class Map extends
17     Mapper<Object, Text, Text, IntWritable> {
18         private Text szo = new Text();
19         public void map(Object key, Text value, Context context)
20                 throws IOException, InterruptedException {
21             StringTokenizer st = new StringTokenizer(value.toString());
22     while (st.hasMoreTokens()) {
23         szo.set(st.nextToken());
24         context.write(szo, new IntWritable(1));
25     }
26         }
27 }
28     public static class Reduce extends
29     Reducer<Text, IntWritable, Text, IntWritable> {
30         private IntWritable eredmeny = new IntWritable();
31         public void reduce(Text key, Iterable<IntWritable> values,
32                 Context context) throws IOException, InterruptedException {
33             int sum = 0;
34             for (IntWritable val : values) {
35                 sum += val.get();
36             }
37             eredmeny.set(sum);
38             context.write(key, eredmeny);
39         }
40     }
41     public static class SecondaryMapper extends
42     Mapper<IntWritable, Text, CompositeKey, Text> {
43         private Text szo = new Text();
44         public void map(IntWritable value, Text key, Context context)
45         throws IOException, InterruptedException {
46             context.write(new CompositeKey((Text) key, value), szo);
```

```
47         }
48     }
49     private static class IntWritableComparator extends IntWritable.Comparator {
50         public int compare(WritableComparable a, WritableComparable b) {
51             return -super.compare(a, b);
52         }
53         public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
54             return -super.compare(b1, s1, l1, b2, s2, l2);
55         }
56     }
57     public static void main(String[] args) throws Exception {
58         Configuration conf = new Configuration();
59         Job job = Job.getInstance(conf, "WordCount");
60         Path tempDir = new Path("temp_wc_" + System.currentTimeMillis());
61         job.setJarByClass(WCSecondarySort.class);
62         job.setMapperClass(Map.class);
63         job.setCombinerClass(Reduce.class);
64         job.setReducerClass(Reduce.class);
65         job.setOutputKeyClass(Text.class);
66         job.setOutputValueClass(IntWritable.class);
67         FileInputFormat.addInputPath(job, new Path(args[0]));
68         FileOutputFormat.setOutputPath(job, tempDir);
69         job.setOutputFormatClass(SequenceFileOutputFormat.class);
70         if (job.waitForCompletion(true)) {
71             Job job2 = new Job(conf, "Sort by count");
72             job2.setJarByClass(WCSecondarySort.class);
73             FileInputFormat.addInputPath(job2, tempDir);
74             job2.setInputFormatClass(SequenceFileInputFormat.class);
75             job2.setMapperClass(InverseMapper.class);
76             FileOutputFormat.setOutputPath(job2, new Path(args[1]));
77             job2.setOutputKeyClass(IntWritable.class);
78             job2.setOutputValueClass(Text.class);
79             job2.setSortComparatorClass(IntWritableComparator.class);
80             FileSystem.get(conf).deleteOnExit(tempDir);
81             tempDir = new Path("temp_wc_" + System.currentTimeMillis());
82             FileOutputFormat.setOutputPath(job2, tempDir);
83             job2.setOutputFormatClass(SequenceFileOutputFormat.class);
84             if (job2.waitForCompletion(true)) {
85                 Job job3 = new Job(conf, "Sort by alphabet");
86                 job3.setJarByClass(WCSecondarySort.class);
87                 FileInputFormat.addInputPath(job3, tempDir);
88                 job3.setInputFormatClass(SequenceFileInputFormat.class);
89                 job3.setMapperClass(SecondaryMapper.class);
90                 job3.setMapOutputKeyClass(CompositeKey.class);
91                 job3.setPartitionerClass(KeyPartitioner.class);
92                 job3.setSortComparatorClass(CompositeKeyComparator.class);
```

WordCount.java | *WCSecondarySort.java

```
92                  job3.setSortComparatorClass(CompositeKeyComparator.class);
93                  job3.setGroupingComparatorClass(KeyGroupingComparator.class);
94                  FileOutputFormat.setOutputPath(job3, new Path(args[1]));
95                  job3.setOutputKeyClass(IntWritable.class);
96                  job3.setOutputValueClass(Text.class);
97                  System.exit(job3.waitForCompletion(true) ? 0 : 1);
98              }
99          }
100         FileSystem.get(conf).deleteOnExit(tempDir);
101     }
102 }
103 class KeyPartitioner extends Partitioner<CompositeKey, Text> {
104     HashPartitioner<IntWritable, Text> hashPartitioner =
105             new HashPartitioner<IntWritable, Text>();
106     IntWritable newKey = new IntWritable();
107     @Override
108     public int getPartition(CompositeKey key, Text value, int numReduceTasks) {
109         try {
110             return hashPartitioner.getPartition(key.getGyakorisag(), value,
111                     numReduceTasks);
112         } catch (Exception e) {
113             e.printStackTrace();
114             return (int) (Math.random() * numReduceTasks);
115         }
116     }
117 }
118 class KeyGroupingComparator extends WritableComparator {
119     protected KeyGroupingComparator() {
120         super(CompositeKey.class, true);
121     }
122     @SuppressWarnings("rawtypes")
123     @Override
124     public int compare(WritableComparable w1, WritableComparable w2) {
125         CompositeKey kulcs1 = (CompositeKey) w1;
126         CompositeKey kulcs2 = (CompositeKey) w2;
127         return kulcs2.getGyakorisag().compareTo(kulcs1.getGyakorisag());
128     }
129 }
130 class CompositeKeyComparator extends WritableComparator {
131     protected CompositeKeyComparator() {
132         super(CompositeKey.class, true);
133     }
134     @SuppressWarnings("rawtypes")
135     @Override
136     public int compare(WritableComparable w1, WritableComparable w2) {
137         CompositeKey kulcs1 = (CompositeKey) w1;
138         CompositeKey kulcs2 = (CompositeKey) w2;
```

WordCount.java | *WCSecondarySort.java

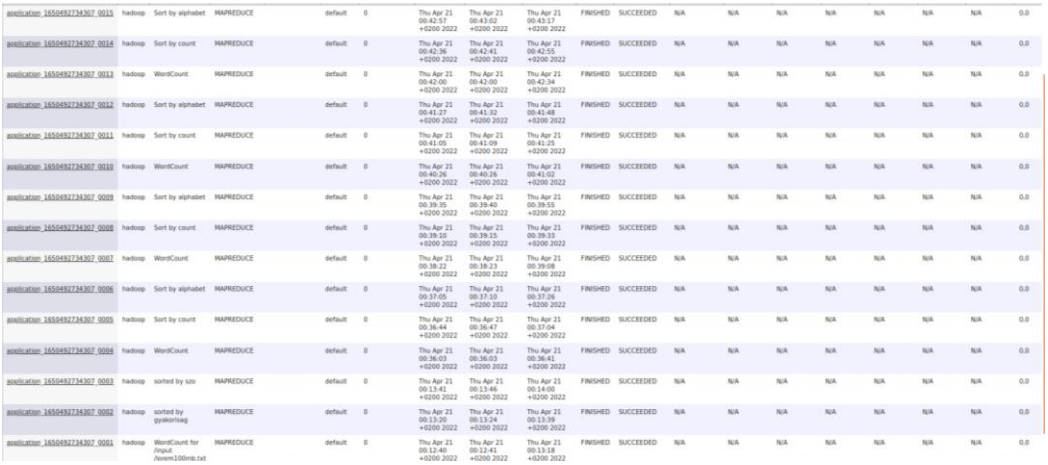
```
138         CompositeKey kulcs2 = (CompositeKey) w2;
139         int cmp = kulcs2.getGyakorisag().compareTo(kulcs1.getGyakorisag());
140         if (cmp != 0)
141             return cmp;
142         return kulcs1.getSzo().compareTo(kulcs2.getSzo());
143     }
144 }
145 class CompositeKey implements WritableComparable<CompositeKey> {
146     private Text szo;
147     private IntWritable gyakorisag;
148     public CompositeKey() {
149         set(new Text(), new IntWritable());
150     }
151     public CompositeKey(String szo, int gyakorisag) {
152         set(new Text(szo), new IntWritable(gyakorisag));
153     }
154     public CompositeKey(Text w, IntWritable f) {
155     set(w, f);
156     }
157     public void set(Text t, IntWritable n) {
158         this.szo = t;
159         this.gyakorisag = n;
160     }
161     @Override
162     public String toString() {return (new StringBuilder()).append(gyakorisag).append(' ').append(szo)
163             .toString();
164     }
165     @Override
166     public boolean equals(Object o) {
167         if (o instanceof CompositeKey) {
168             CompositeKey comp = (CompositeKey) o;
169             return szo.equals(comp.szo) && gyakorisag.equals(comp.gyakorisag);
170         }
171         return false;
172     }
173     @Override
174     public void readFields(DataInput in) throws IOException {
175         szo.readFields(in);
176         gyakorisag.readFields(in);
177     }
178     @Override
179     public void write(DataOutput out) throws IOException {
180         szo.write(out);
181         gyakorisag.write(out);
182     }
183     @Override
184     public int compareTo(CompositeKey o) {
```

```
183     @Override
184     public int compareTo(CompositeKey o) {
185         int eredmeny = szo.compareTo(o.szo);
186         if (eredmeny != 0) {
187             return eredmeny;
188         }
189         return eredmeny = gyakorisag.compareTo(o.gyakorisag);
190     }
191     public Text getSzo() {
192         return szo;
193     }
194         public IntWritable getGyakorisag() {
195         return gyakorisag;
196     }
197 }
```

A teszteket az alábbi parancsokkal futtatom le az alapértelmezett beállításokkal. Először a 100 MB-os fájlt dolgozom fel, utána az 500 MB-os fájlt, végül az 1GB-ost.

100 MB-os teszt:

```
hadoop jar /usr/local/hadoop/userscripts/HadoopSecondarySort.jar /input/lorem100mb.txt /lorem_100mb_wcsecondarysort_noopt_1

hadoop jar /usr/local/hadoop/userscripts/HadoopSecondarySort.jar /input/lorem100mb.txt /lorem_100mb_wcsecondarysort_noopt_2

hadoop jar /usr/local/hadoop/userscripts/HadoopSecondarySort.jar /input/lorem100mb.txt /lorem_100mb_wcsecondarysort_noopt_3

hadoop jar /usr/local/hadoop/userscripts/HadoopSecondarySort.jar /input/lorem100mb.txt /lorem_100mb_wcsecondarysort_noopt_4

hadoop jar /usr/local/hadoop/userscripts/HadoopSecondarySort.jar /input/lorem100mb.txt /lorem_100mb_wcsecondarysort_noopt_5
```

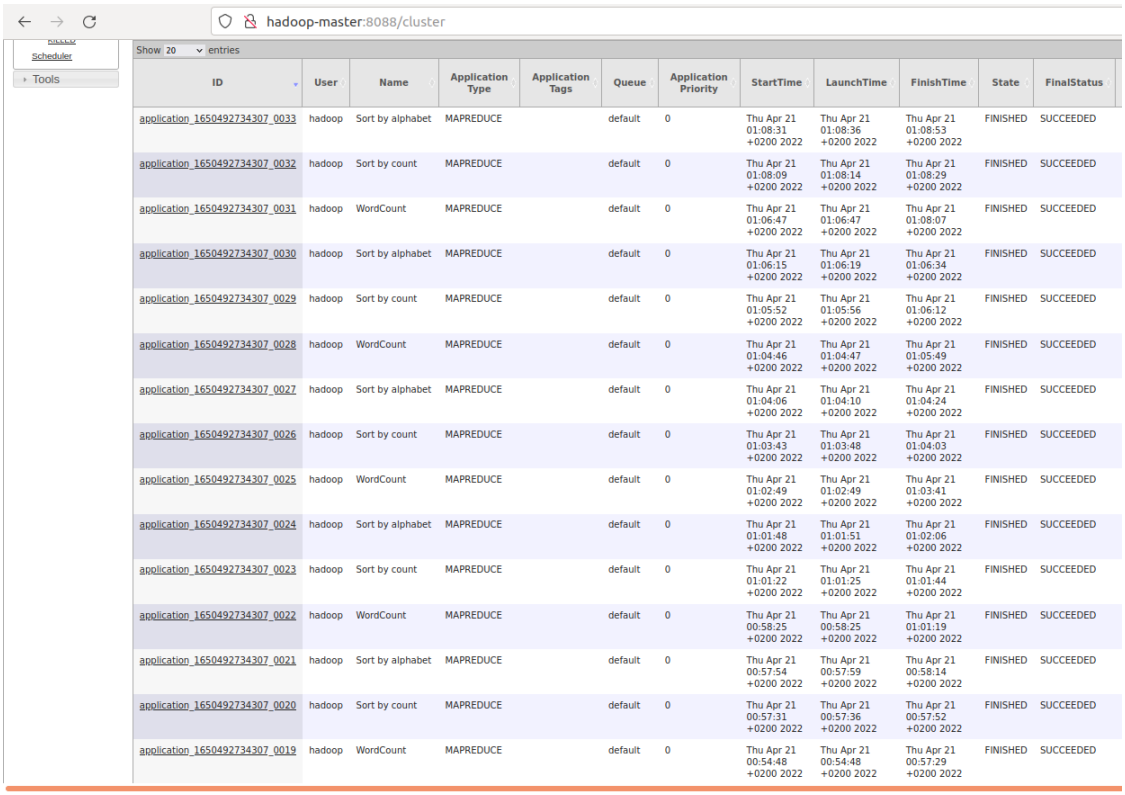


Ezen a képen láthatók a 100MB-os fájl feldolgozásának eredményei. Fontos észben tartani, hogy 3 munkafolyamat számít egynek, ezért azokat össze kell adni és utána átlagolni.

500 MB-os teszt:

```
hadoop jar /usr/local/hadoop/userscripts/HadoopSecondarySort.jar /input/lorem500mb.txt /lorem_500mb_wcsecondarysort_noopt_1

hadoop jar /usr/local/hadoop/userscripts/HadoopSecondarySort.jar /input/lorem500mb.txt /lorem_500mb_wcsecondarysort_noopt_2
```

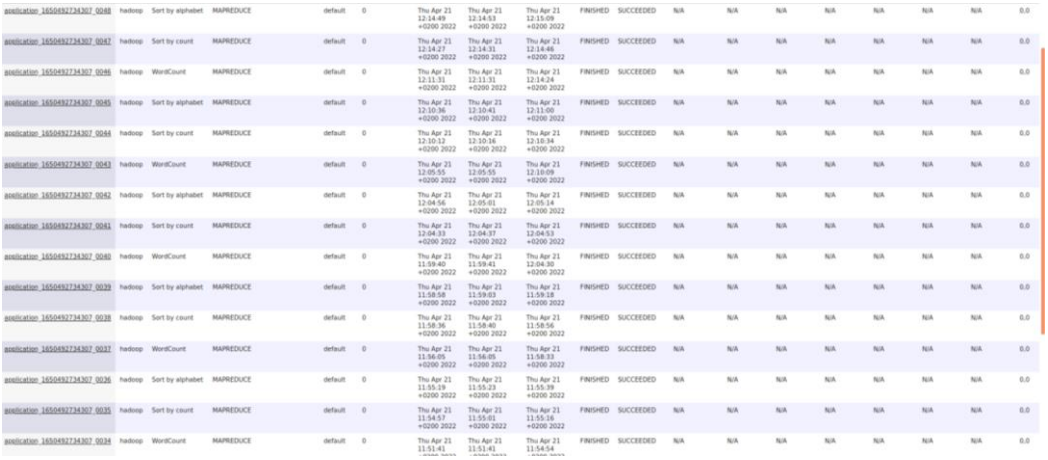
```
hadoop jar /usr/local/hadoop/userscripts/HadoopSecondarySort.jar /input/lorem500mb.txt /lorem_500mb_wcsecondarysort_noopt_3

hadoop jar /usr/local/hadoop/userscripts/HadoopSecondarySort.jar /input/lorem500mb.txt /lorem_500mb_wcsecondarysort_noopt_4

hadoop jar /usr/local/hadoop/userscripts/HadoopSecondarySort.jar /input/lorem500mb.txt /lorem_500mb_wcsecondarysort_noopt_5
```



Ezen a képen láthatók a 500 MB-os fájl feldolgozásának eredményei. Fontos észben tartani, hogy 3 munkafolyamat számít egynek, ezért azokat össze kell adni és utána átlagolni.

1 GB-os teszt:

```
hadoop jar /usr/local/hadoop/userscripts/HadoopSecondarySort.jar /input/lorem1gb.txt /lorem_1gb_wcsecondarysort_noopt_1

hadoop jar /usr/local/hadoop/userscripts/HadoopSecondarySort.jar /input/lorem1gb.txt /lorem_1gb_wcsecondarysort_noopt_2

hadoop jar /usr/local/hadoop/userscripts/HadoopSecondarySort.jar /input/lorem1gb.txt /lorem_1gb_wcsecondarysort_noopt_3

hadoop jar /usr/local/hadoop/userscripts/HadoopSecondarySort.jar /input/lorem1gb.txt /lorem_1gb_wcsecondarysort_noopt_4

hadoop jar /usr/local/hadoop/userscripts/HadoopSecondarySort.jar /input/lorem1gb.txt /lorem_1gb_wcsecondarysort_noopt_5
```



Ezen a képen láthatók a 1GB-os fájl feldolgozásának eredményei. Fontos észben tartani, hogy 3 munkafolyamat számít egynek, ezért azokat össze kell adni és utána átlagolni.

A lefutott tesztek futás idejének az átlaga a következő táblázatban látható.

Hadoop WC és Secondary Sort optimalizálás nélkül			
Adat mérete	100MB	500MB	1GB
Átlagos futásidő	1 perc 9 mp	2 perc 17 mp	4 perc 3 mp

6.2.3. Hadoop – Page Rank

A Page Rank algoritmust a Google dolgozta ki azzal a céllal, hogy rangsorolja a weboldalakat. Az egyes oldalakra mutató linkek mennyisége és minősége alapján történik a pontozás. A logika alapja, hogy egy fontosabb weboldalra több hivatkozás található és súlyozza azt is, hogy a weblap, ami hivatkozik rá milyen fontosságú.

A forrásadatokban két oszlop van, az első a hivatkozás forrása, a másik a hivatkozás célja. Kezdetekben minden linknek 1 az értéke és utána az algoritmus minden iterációja után frissül. Az alábbi képletet használom, amit a Google használt 1998 óta.

$$R_i = d * \sum_{j \in S} \frac{R_j}{N_j} + (1 - d)$$

Az R_i az értéke az i linknek, a R_j az értéke a j linknek; az N_j a j linkről kimenő hivatkozások száma; az S azoknak a linkeknek a halmaza, amik az i linkre mutatnak; a d a csillapítási faktor, jelen esetben az értéke 0,85. [29]

Az algoritmusunknak megadhatjuk, hogy a képletet hányszor futtassa le. Minél többször fut le a képlet annál jobban közelítjük a pontos értéket. A kimenetre minta:

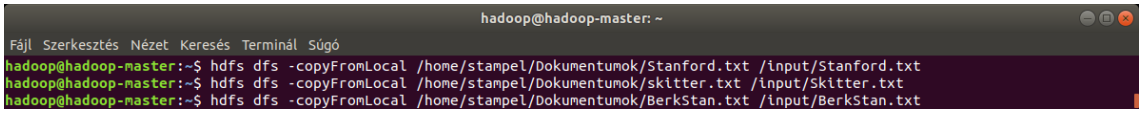
```
100061  1.006360948339204
100062  0.6977135410064511
100063  0.8363916915845452
100064  1.3190953613916538
100065  1.5195427911179893
100066  0.8055799163635339
100067  0.8136950252991546
```

A tesztfájlokat a http://snap.stanford.edu/data/-ról töltöttem le. [30]

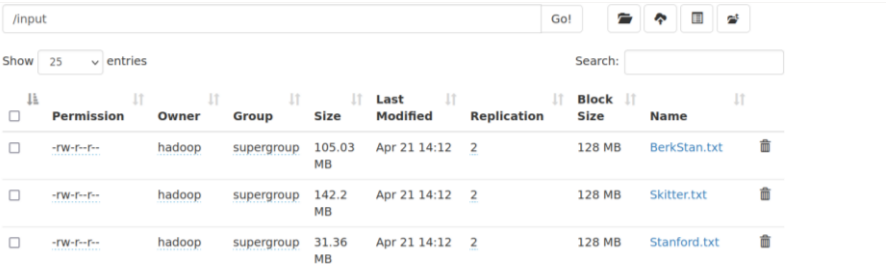
Fájl neve	Fájl mérete	Csúcsok száma	Élek száma
Skitter.txt	142 MB	1.696.415	11.095.298
Stanford.txt	31,3 MB	281.903	2.312.497
BerkStan.txt	105 MB	685.230	7.600.595

A forrásállományokat felmásolom a HDFS-re.

```
hdfs dfs -copyFromLocal /home/stampel/Dokumentumok/BerkStan.txt /input/BerkStan.txt

hdfs dfs -copyFromLocal /home/stampel/Dokumentumok/skitter.txt /input/Skitter.txt

hdfs dfs -copyFromLocal /home/stampel/Dokumentumok/Stanford.txt /input/Stanford.txt
```

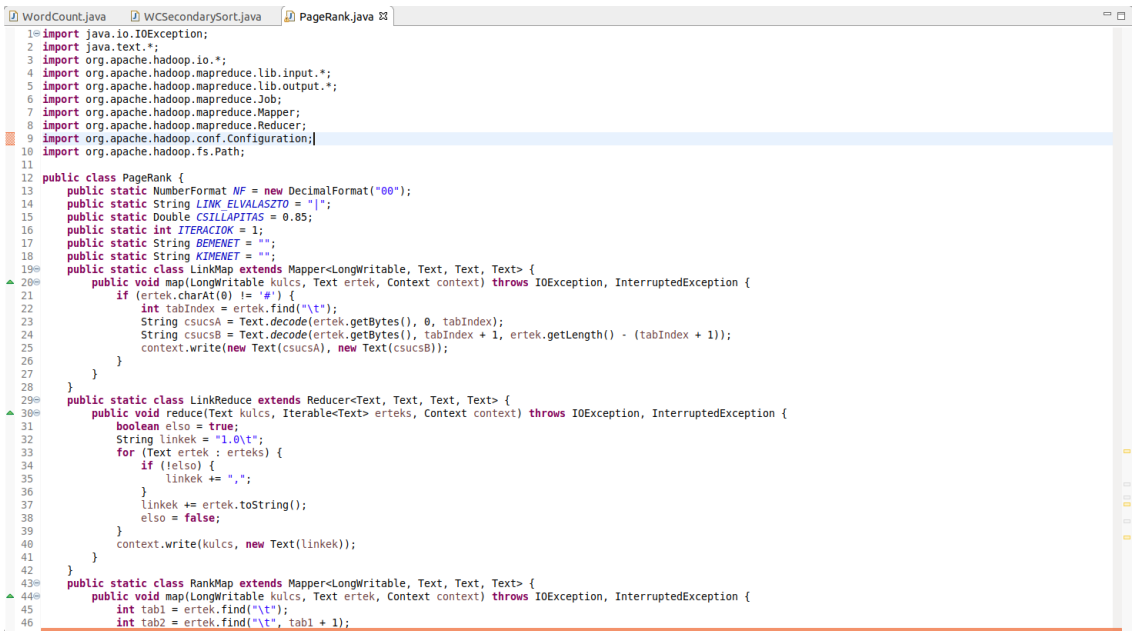


Ellenőrzöm, hogy a fájlok sikeresen felkerültek a HDFS-re.



Eclipseben írtam meg a Hadoop Page Rank implementációt Java nyelven.

JAR-ként exportálom a /usr/local/hadoop/userscripts mappába.



```
            String page = Text.decode(ertek.getBytes(), 0, tab1);
            String pageRank = Text.decode(ertek.getBytes(), tab1 + 1, tab2 - (tab1 + 1));
            if (tab2 == -1) {
                return;
            }
            String linkek = Text.decode(ertek.getBytes(), tab2 + 1, ertek.getLength() - (tab2 + 1));
            String[] masOldalak = linkek.split(",");
            for (String masOldal : masOldalak) {
                Text pageRankOsszesLinkkel = new Text(pageRank + "\t" + masOldalak.length);
                context.write(new Text(masOldal), pageRankOsszesLinkkel);
            }
            context.write(new Text(page), new Text(PageRank.LINK_ELVALASZTO + linkek));
        }
    }
    public static class RankReduce extends Reducer<Text, Text, Text, Text> {
        public void reduce(Text kulcs, Iterable<Text> erteks, Context context) throws IOException, InterruptedException {
            String linkek = "";
            double osszesPageRankReszesedes = 0.0;
            for (Text ertek : erteks) {
                String content = ertek.toString();
                if (content.startsWith(PageRank.LINK_ELVALASZTO)) {
                    linkek += content.substring(PageRank.LINK_ELVALASZTO.length());
                }
                else {
                    String[] split = content.split("\\t");
                    double pageRank = Double.parseDouble(split[0]);
                    if (split[1] != null && !split[1].equals("null")) {
                        int osszesLink = Integer.parseInt(split[1]);
                        osszesPageRankReszesedes += (pageRank / osszesLink);
                    }
                }
            }
            double newRank = PageRank.CSILLAPITAS * osszesPageRankReszesedes + (1 - PageRank.CSILLAPITAS);
            if (newRank > 0.150000000000000002 && !kulcs.toString().trim().equals("")) {
                context.write(kulcs, new Text(newRank + "\t" + linkek));
            }
        }
    }
    public static class SortMap extends Mapper<LongWritable, Text, Text, DoubleWritable> {
        public void map(LongWritable kulcs, Text ertek, Context context) throws IOException, InterruptedException {
            int tab1 = ertek.find("\t");
            int tab2 = ertek.find("\t", tab1 + 1);
            String page = Text.decode(ertek.getBytes(), 0, tab1);
            double pageRank = Double.parseDouble(Text.decode(ertek.getBytes(), tab1 + 1, tab2 - (tab1 + 1)));
            context.write(new Text(page), new DoubleWritable(pageRank));
        }
```

```
    }
    public static void main(String[] args) throws Exception {
        PageRank.BEMENET = args[0];
        PageRank.KIMENET = args[1];
        PageRank.ITERACIOK = Integer.parseInt(args[2]);
        String bemenetiUt = null;
        String utolsoKimenetiUt = null;
        PageRank pagerank = new PageRank();
        System.out.println("Kapcsolodó linkek fetchelése");
        boolean kesz = pagerank.job("szomszedokKeresese", LinkMap.class,
                LinkReduce.class, BEMENET, KIMENET + "/iter00");
        if (!kesz) {
            System.exit(1);
        }
        for (int futasokSzama = 0; futasokSzama < ITERACIOK; futasokSzama++) {
            bemenetiUt = KIMENET + "/iter" + NF.format(futasokSzama);
            utolsoKimenetiUt = KIMENET + "/iter" + NF.format(futasokSzama + 1);
            System.out.println("Számolás kezdése [" + (futasokSzama + 1) + "/" + PageRank.ITERACIOK + "]");
            kesz = pagerank.job("jobRankokSzamolasa", RankMap.class,
                    RankReduce.class, bemenetiUt, utolsoKimenetiUt);
            if (!kesz) {
                System.exit(1);
            }
        }
        System.out.println("Rendezés kezdése");
        kesz = pagerank.job("jobRankokRendezese", SortMap.class,
                SortMap.class, utolsoKimenetiUt, KIMENET + "/result");
        if (!kesz) {
            System.exit(1);
        }

        System.out.println("Kesz!");
        System.exit(0);
    }
    public boolean job(String jobNeve, Class a, Class b, String be, String ki)
            throws IOException, ClassNotFoundException, InterruptedException {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, jobNeve);
        job.setJarByClass(PageRank.class);

        FileInputFormat.addInputPath(job, new Path(be));
        job.setInputFormatClass(TextInputFormat.class);
        if (jobNeve.equals("jobRankokRendezese")) {
            job.setOutputKeyClass(Text.class);
            job.setMapOutputValueClass(DoubleWritable.class);
        }
```

```
        else {
            job.setMapOutputKeyClass(Text.class);
            job.setMapOutputValueClass(Text.class);
        }
        job.setMapperClass(a);
        FileOutputFormat.setOutputPath(job, new Path(ki));
        job.setOutputFormatClass(TextOutputFormat.class);
        if (jobNeve.equals("jobRankokRendezese")) {
            job.setOutputKeyClass(Text.class);
            job.setMapOutputValueClass(DoubleWritable.class);
        }
        else {
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(Text.class);
            job.setReducerClass(b);
        }
        return job.waitForCompletion(true);
    }
}
```

A Page Rank tesztet csak három iteráción keresztül fogom futtatni és az előző tesztektől eltérően csak négyszer, mivel a művelet hossza miatt konzisztensebb a futásidő.

A teszteket az alábbi parancsokkal futtatom le az alapértelmezett beállításokkal. Először a Skitter.txt fájlt dolgozom fel, utána az Stanford.txt fájlt, végül az BerkStan.txt fájlt.

Skitter.txt teszt:

```
hadoop jar /usr/local/hadoop/userscripts/HadoopPageRank.jar /input/Skitter.txt /pagerank_skitter_noopt_1 3

hadoop jar /usr/local/hadoop/userscripts/HadoopPageRank.jar /input/Skitter.txt /pagerank_skitter_noopt_2 3

hadoop jar /usr/local/hadoop/userscripts/HadoopPageRank.jar /input/Skitter.txt /pagerank_skitter_noopt_3 3

hadoop jar /usr/local/hadoop/userscripts/HadoopPageRank.jar /input/Skitter.txt /pagerank_skitter_noopt_4 3
```

Stanford.txt teszt:

```
hadoop jar /usr/local/hadoop/userscripts/HadoopPageRank.jar /input/Stanford.txt /pagerank_stanford_noopt_1 3

hadoop jar /usr/local/hadoop/userscripts/HadoopPageRank.jar /input/Stanford.txt /pagerank_stanford_noopt_2 3

hadoop jar /usr/local/hadoop/userscripts/HadoopPageRank.jar /input/Stanford.txt /pagerank_stanford_noopt_3 3

hadoop jar /usr/local/hadoop/userscripts/HadoopPageRank.jar /input/Stanford.txt /pagerank_stanford_noopt_4 3
```

Active Nodes	Decommissioning Nodes	Decommissioned Nodes
3	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation
Capacity Scheduler	[memory-mb (unit=Mi), vcores]	<memory:1024, vCores:1>

Show 20 entries

ID	User	Name	Application Type	Application Tags	Queue	Application Priority	StartTime	LaunchTime	FinishTime
application_1650565152405_0042	hadoop	jobRankokRendezese	MAPREDUCE		default	0	Thu Apr 21 21:23:43 +0200 2022	Thu Apr 21 21:23:48 +0200 2022	Thu Apr 21 21:24:03 +0200 2022
application_1650565152405_0041	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:23:11 +0200 2022	Thu Apr 21 21:23:16 +0200 2022	Thu Apr 21 21:23:41 +0200 2022
application_1650565152405_0040	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:22:40 +0200 2022	Thu Apr 21 21:22:44 +0200 2022	Thu Apr 21 21:23:09 +0200 2022
application_1650565152405_0039	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:22:08 +0200 2022	Thu Apr 21 21:22:13 +0200 2022	Thu Apr 21 21:22:38 +0200 2022
application_1650565152405_0038	hadoop	szomszedokKeresese	MAPREDUCE		default	0	Thu Apr 21 21:21:45 +0200 2022	Thu Apr 21 21:21:45 +0200 2022	Thu Apr 21 21:22:06 +0200 2022
application_1650565152405_0037	hadoop	jobRankokRendezese	MAPREDUCE		default	0	Thu Apr 21 21:20:49 +0200 2022	Thu Apr 21 21:20:54 +0200 2022	Thu Apr 21 21:21:10 +0200 2022
application_1650565152405_0036	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:20:17 +0200 2022	Thu Apr 21 21:20:22 +0200 2022	Thu Apr 21 21:20:47 +0200 2022
application_1650565152405_0035	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:19:44 +0200 2022	Thu Apr 21 21:19:50 +0200 2022	Thu Apr 21 21:20:15 +0200 2022
application_1650565152405_0034	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:19:13 +0200 2022	Thu Apr 21 21:19:19 +0200 2022	Thu Apr 21 21:19:43 +0200 2022
application_1650565152405_0033	hadoop	szomszedokKeresese	MAPREDUCE		default	0	Thu Apr 21 21:18:51 +0200 2022	Thu Apr 21 21:18:51 +0200 2022	Thu Apr 21 21:19:12 +0200 2022
application_1650565152405_0032	hadoop	jobRankokRendezese	MAPREDUCE		default	0	Thu Apr 21 21:16:06 +0200 2022	Thu Apr 21 21:16:10 +0200 2022	Thu Apr 21 21:16:25 +0200 2022
application_1650565152405_0031	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:15:22 +0200 2022	Thu Apr 21 21:15:27 +0200 2022	Thu Apr 21 21:16:04 +0200 2022
application_1650565152405_0030	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:14:38 +0200 2022	Thu Apr 21 21:14:41 +0200 2022	Thu Apr 21 21:15:20 +0200 2022
application_1650565152405_0029	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:14:07 +0200 2022	Thu Apr 21 21:14:11 +0200 2022	Thu Apr 21 21:14:34 +0200 2022
application_1650565152405_0028	hadoop	szomszedokKeresese	MAPREDUCE		default	0	Thu Apr 21 21:13:44 +0200 2022	Thu Apr 21 21:13:44 +0200 2022	Thu Apr 21 21:14:04 +0200 2022
application_1650565152405_0027	hadoop	jobRankokRendezese	MAPREDUCE		default	0	Thu Apr 21 21:11:57 +0200 2022	Thu Apr 21 21:12:02 +0200 2022	Thu Apr 21 21:12:18 +0200 2022
application_1650565152405_0026	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:11:23 +0200 2022	Thu Apr 21 21:11:27 +0200 2022	Thu Apr 21 21:11:55 +0200 2022
application_1650565152405_0025	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:10:50 +0200 2022	Thu Apr 21 21:10:54 +0200 2022	Thu Apr 21 21:11:20 +0200 2022
application_1650565152405_0024	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:10:18 +0200 2022	Thu Apr 21 21:10:22 +0200 2022	Thu Apr 21 21:10:47 +0200 2022
application_1650565152405_0023	hadoop	szomszedokKeresese	MAPREDUCE		default	0	Thu Apr 21 21:09:52 +0200 2022	Thu Apr 21 21:09:53 +0200 2022	Thu Apr 21 21:10:15 +0200 2022

BerkStan.txt teszt:

```
hadoop jar /usr/local/hadoop/userscripts/HadoopPageRank.jar /input/BerkStan.txt /pagerank_berkstan_noopt_1

hadoop jar /usr/local/hadoop/userscripts/HadoopPageRank.jar /input/BerkStan.txt /pagerank_berkstan_noopt_2

hadoop jar /usr/local/hadoop/userscripts/HadoopPageRank.jar /input/BerkStan.txt /pagerank_berkstan_noopt_3

hadoop jar /usr/local/hadoop/userscripts/HadoopPageRank.jar /input/BerkStan.txt /pagerank_berkstan_noopt_4
```

ID	User	Name	Application Type	Application Tags	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	Allocated GPUs	Reserved CPU VCores	Reserved Memory MB	Reserved GPUs	% of Queue	C
application_1650565152405_0062	hadoop	jobRankokRendezese	MAPREDUCE		default	0	Thu Apr 21 21:47:08 +0200 2022	Thu Apr 21 21:47:14 +0200 2022	Thu Apr 21 21:47:39 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0061	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:46:03 +0200 2022	Thu Apr 21 21:46:08 +0200 2022	Thu Apr 21 21:47:07 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0060	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:45:13 +0200 2022	Thu Apr 21 21:45:18 +0200 2022	Thu Apr 21 21:46:01 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0059	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:44:25 +0200 2022	Thu Apr 21 21:44:29 +0200 2022	Thu Apr 21 21:45:11 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0058	hadoop	szomszedokKeresese	MAPREDUCE		default	0	Thu Apr 21 21:43:32 +0200 2022	Thu Apr 21 21:43:33 +0200 2022	Thu Apr 21 21:44:22 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0057	hadoop	jobRankokRendezese	MAPREDUCE		default	0	Thu Apr 21 21:42:48 +0200 2022	Thu Apr 21 21:42:53 +0200 2022	Thu Apr 21 21:43:15 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0056	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:41:24 +0200 2022	Thu Apr 21 21:41:30 +0200 2022	Thu Apr 21 21:42:46 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0055	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:40:33 +0200 2022	Thu Apr 21 21:40:38 +0200 2022	Thu Apr 21 21:41:23 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0054	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:39:44 +0200 2022	Thu Apr 21 21:39:49 +0200 2022	Thu Apr 21 21:40:31 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0053	hadoop	szomszedokKeresese	MAPREDUCE		default	0	Thu Apr 21 21:38:51 +0200 2022	Thu Apr 21 21:38:52 +0200 2022	Thu Apr 21 21:39:42 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0052	hadoop	jobRankokRendezese	MAPREDUCE		default	0	Thu Apr 21 21:37:45 +0200 2022	Thu Apr 21 21:37:49 +0200 2022	Thu Apr 21 21:38:07 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0051	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:36:32 +0200 2022	Thu Apr 21 21:36:37 +0200 2022	Thu Apr 21 21:37:43 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0050	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:35:39 +0200 2022	Thu Apr 21 21:35:46 +0200 2022	Thu Apr 21 21:36:30 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0049	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:34:50 +0200 2022	Thu Apr 21 21:34:55 +0200 2022	Thu Apr 21 21:35:38 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0048	hadoop	szomszedokKeresese	MAPREDUCE		default	0	Thu Apr 21 21:33:59 +0200 2022	Thu Apr 21 21:33:59 +0200 2022	Thu Apr 21 21:34:48 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0047	hadoop	jobRankokRendezese	MAPREDUCE		default	0	Thu Apr 21 21:33:24 +0200 2022	Thu Apr 21 21:33:29 +0200 2022	Thu Apr 21 21:33:47 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0046	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:32:06 +0200 2022	Thu Apr 21 21:32:11 +0200 2022	Thu Apr 21 21:33:22 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0045	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:31:18 +0200 2022	Thu Apr 21 21:31:23 +0200 2022	Thu Apr 21 21:32:04 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0044	hadoop	jobRankokSzamolasa	MAPREDUCE		default	0	Thu Apr 21 21:30:22 +0200 2022	Thu Apr 21 21:30:26 +0200 2022	Thu Apr 21 21:31:16 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0
application_1650565152405_0043	hadoop	szomszedokKeresese	MAPREDUCE		default	0	Thu Apr 21 21:29:33 +0200 2022	Thu Apr 21 21:29:33 +0200 2022	Thu Apr 21 21:30:19 +0200 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0,0	0,0

/cluster/app/application_1650565152405_0060

A lefutott tesztek futás idejének az átlaga a következő táblázatban látható.

Hadoop Page Rank optimalizálás nélkül			
Név	Skitter	BerkStan	Stanford
Futási idő	7 perc 26 mp	3 perc 45 mp	1 perc 58 mp

6.2.4. Spark – Word Count

A teszt elméleti háttere teljesen megegyezik a 6.2.1-es pontban olvashatóval.

Eclipseben írtam meg a Spark Word Count implementációt Java nyelven.

JAR-ként exportálom a /usr/local/spark/userscripts mappába.


```
cd /usr/local/spark
```

A Spark teszteket a spark-submit paranccsal tudjuk lefuttatni. A --mester után meg kell adni az IP címét, utána a feldolgozandó fájlt a HDFS-ről és a kimeneti mappát.

100 MB-os teszt:

```
./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWordCount.jar /input/lorem100mb.txt /spark/wc_lorem100mb_noopt_1

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWordCount.jar /input/lorem100mb.txt /spark/wc_lorem100mb_noopt_2

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWordCount.jar /input/lorem100mb.txt /spark/wc_lorem100mb_noopt_3

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWordCount.jar /input/lorem100mb.txt /spark/wc_lorem100mb_noopt_4

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWordCount.jar /input/lorem100mb.txt /spark/wc_lorem100mb_noopt_5
```

Completed Applications (8)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20220423132133-0007	Word Count	6	1024.0 MiB		2022/04/23 13:21:33	hadoop	FINISHED	13 s
app-20220423132106-0006	Word Count	6	1024.0 MiB		2022/04/23 13:21:06	hadoop	FINISHED	12 s
app-20220423132045-0005	Word Count	6	1024.0 MiB		2022/04/23 13:20:45	hadoop	FINISHED	12 s
app-20220423132019-0004	Word Count	6	1024.0 MiB		2022/04/23 13:20:19	hadoop	FINISHED	12 s
app-20220423131942-0003	Word Count	6	1024.0 MiB		2022/04/23 13:19:42	hadoop	FINISHED	12 s

500 MB-os teszt:

```
./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWordCount.jar /input/lorem500mb.txt /spark/wc_lorem500mb_noopt_1

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWordCount.jar /input/lorem500mb.txt /spark/wc_lorem500mb_noopt_2

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWordCount.jar /input/lorem500mb.txt /spark/wc_lorem500mb_noopt_3

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWordCount.jar /input/lorem500mb.txt /spark/wc_lorem500mb_noopt_4

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWordCount.jar /input/lorem500mb.txt /spark/wc_lorem500mb_noopt_5
```

Completed Applications (13)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20220423133818-0012	Word Count	6	1024.0 MiB		2022/04/23 13:38:18	hadoop	FINISHED	18 s
app-20220423133751-0011	Word Count	6	1024.0 MiB		2022/04/23 13:37:51	hadoop	FINISHED	17 s
app-20220423133722-0010	Word Count	6	1024.0 MiB		2022/04/23 13:37:22	hadoop	FINISHED	17 s
app-20220423133647-0009	Word Count	6	1024.0 MiB		2022/04/23 13:36:47	hadoop	FINISHED	17 s
app-20220423133613-0008	Word Count	6	1024.0 MiB		2022/04/23 13:36:13	hadoop	FINISHED	18 s

1 GB-os teszt:

```
./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWordCount.jar /input/lorem1gb.txt /spark/wc_lorem1gb_noopt_1

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWordCount.jar /input/lorem1gb.txt /spark/wc_lorem1gb_noopt_2

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWordCount.jar /input/lorem1gb.txt /spark/wc_lorem1gb_noopt_3

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWordCount.jar /input/lorem1gb.txt /spark/wc_lorem1gb_noopt_4

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWordCount.jar /input/lorem1gb.txt /spark/wc_lorem1gb_noopt_5
```

Completed Applications (18)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20220423134608-0017	Word Count	6	1024.0 MiB		2022/04/23 13:46:08	hadoop	FINISHED	24 s
app-20220423134312-0016	Word Count	6	1024.0 MiB		2022/04/23 13:43:12	hadoop	FINISHED	22 s
app-20220423134202-0015	Word Count	6	1024.0 MiB		2022/04/23 13:42:02	hadoop	FINISHED	22 s
app-20220423134123-0014	Word Count	6	1024.0 MiB		2022/04/23 13:41:23	hadoop	FINISHED	22 s
app-20220423134033-0013	Word Count	6	1024.0 MiB		2022/04/23 13:40:33	hadoop	FINISHED	22 s

A lefutott tesztek futás idejének az átlaga a következő táblázatban látható.

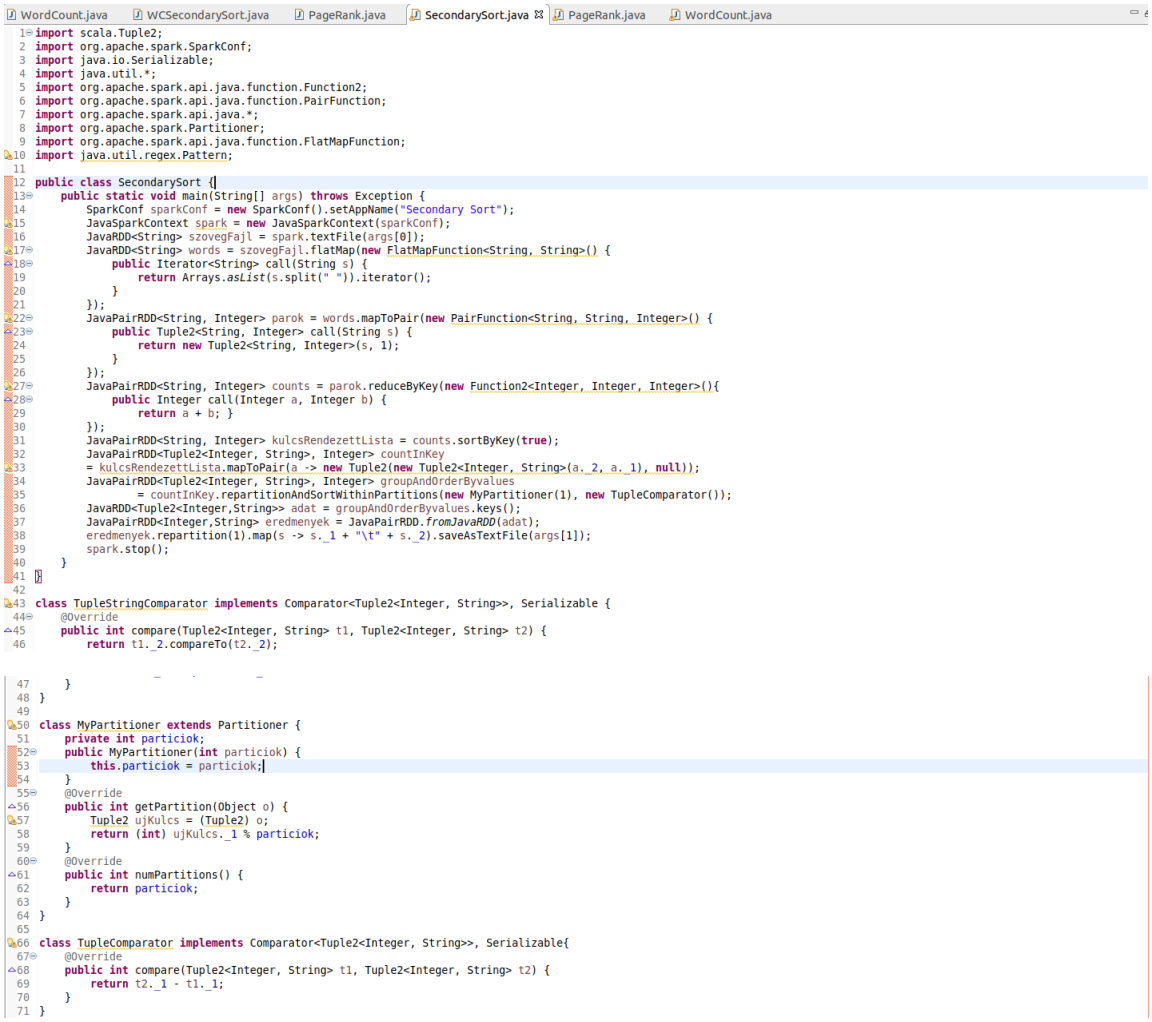
Spark Word Count optimalizálás nélkül			
Adat mérete	100MB	500MB	1GB
Átlagos futásidő	12,2 mp	17 mp	22,4 mp

6.2.5. Spark - Word Count és többszöri rendezés

A teszt elméleti háttere teljesen megegyezik a 6.2.2-es pontban olvashatóval.

Eclipseben írtam meg a Spark Word Count Secondary Sort implementációt Java nyelven.

JAR-ként exportálom a /usr/local/spark/userscripts mappába.




```java
import scala.Tuple2;
import org.apache.spark.SparkConf;
import java.io.Serializable;
import java.util.*;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.function.PairFunction;
import org.apache.spark.api.java.*;
import org.apache.spark.Partitioner;
import org.apache.spark.api.java.function.FlatMapFunction;
import java.util.regex.Pattern;

public class SecondarySort {
    public static void main(String[] args) throws Exception {
        SparkConf sparkConf = new SparkConf().setAppName("Secondary Sort");
        JavaSparkContext spark = new JavaSparkContext(sparkConf);
        JavaRDD<String> szovegFajl = spark.textFile(args[0]);
        JavaRDD<String> words = szovegFajl.flatMap(new FlatMapFunction<String, String>() {
            public Iterator<String> call(String s) {
                return Arrays.asList(s.split(" ")).iterator();
            }
        });
        JavaPairRDD<String, Integer> parok = words.mapToPair(new PairFunction<String, String, Integer>() {
            public Tuple2<String, Integer> call(String s) {
                return new Tuple2<String, Integer>(s, 1);
            }
        });
        JavaPairRDD<String, Integer> counts = parok.reduceByKey(new Function2<Integer, Integer, Integer>(){
            public Integer call(Integer a, Integer b) {
                return a + b; }
        });
        JavaPairRDD<String, Integer> kulcsRendezettLista = counts.sortByKey(true);
        JavaPairRDD<Tuple2<Integer, String>, Integer> countInKey
        = kulcsRendezettLista.mapToPair(a -> new Tuple2(new Tuple2<Integer, String>(a._2, a._1), null));
        JavaPairRDD<Tuple2<Integer, String>, Integer> groupAndOrderByvalues
                = countInKey.repartitionAndSortWithinPartitions(new MyPartitioner(1), new TupleComparator());
        JavaRDD<Tuple2<Integer,String>> adat = groupAndOrderByvalues.keys();
        JavaPairRDD<Integer,String> eredmenyek = JavaPairRDD.fromJavaRDD(adat);
        eredmenyek.repartition(1).map(s -> s._1 + "\t" + s._2).saveAsTextFile(args[1]);
        spark.stop();
    }
}

class TupleStringComparator implements Comparator<Tuple2<Integer, String>>, Serializable {
    @Override
    public int compare(Tuple2<Integer, String> t1, Tuple2<Integer, String> t2) {
        return t1._2.compareTo(t2._2);
    }
}

class MyPartitioner extends Partitioner {
    private int particiok;
    public MyPartitioner(int particiok) {
        this.particiok = particiok;
    }
    @Override
    public int getPartition(Object o) {
        Tuple2 ujKulcs = (Tuple2) o;
        return (int) ujKulcs._1 % particiok;
    }
    @Override
    public int numPartitions() {
        return particiok;
    }
}

class TupleComparator implements Comparator<Tuple2<Integer, String>>, Serializable{
    @Override
    public int compare(Tuple2<Integer, String> t1, Tuple2<Integer, String> t2) {
        return t2._1 - t1._1;
    }
}
```

100 MB-os teszt:

```
./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWCSecondarySort.jar /input/lorem100mb.txt /spark/wcss_lorem100mb_noopt_1

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWCSecondarySort.jar /input/lorem100mb.txt /spark/wcss_lorem100mb_noopt_2
```

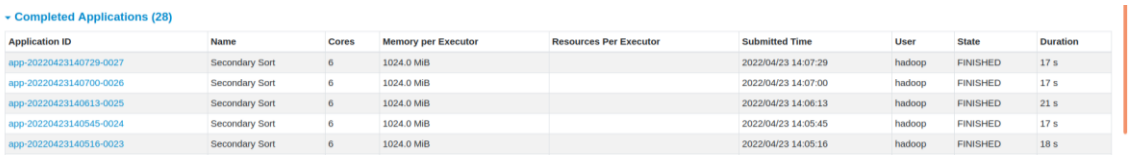
```
./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWCSecondarySort.jar /input/lorem100mb.txt /spark/wcss_lorem100mb_noopt_3

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWCSecondarySort.jar /input/lorem100mb.txt /spark/wcss_lorem100mb_noopt_4

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWCSecondarySort.jar /input/lorem100mb.txt /spark/wcss_lorem100mb_noopt_5
```

▾ Completed Applications (23)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20220423140219-0022	Secondary Sort	6	1024.0 MiB		2022/04/23 14:02:19	hadoop	FINISHED	14 s
app-20220423140151-0021	Secondary Sort	6	1024.0 MiB		2022/04/23 14:01:51	hadoop	FINISHED	15 s
app-20220423140123-0020	Secondary Sort	6	1024.0 MiB		2022/04/23 14:01:23	hadoop	FINISHED	14 s
app-20220423140035-0019	Secondary Sort	6	1024.0 MiB		2022/04/23 14:00:35	hadoop	FINISHED	13 s
app-20220423135927-0018	Secondary Sort	6	1024.0 MiB		2022/04/23 13:59:27	hadoop	FINISHED	13 s

500 MB-os teszt:

```
./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWCSecondarySort.jar /input/lorem500mb.txt /spark/wcss_lorem500mb_noopt_1

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWCSecondarySort.jar /input/lorem500mb.txt /spark/wcss_lorem500mb_noopt_2

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWCSecondarySort.jar /input/lorem500mb.txt /spark/wcss_lorem500mb_noopt_3

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWCSecondarySort.jar /input/lorem500mb.txt /spark/wcss_lorem500mb_noopt_4

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWCSecondarySort.jar /input/lorem500mb.txt /spark/wcss_lorem500mb_noopt_5
```



▾ Completed Applications (28)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20220423140729-0027	Secondary Sort	6	1024.0 MiB		2022/04/23 14:07:29	hadoop	FINISHED	17 s
app-20220423140700-0026	Secondary Sort	6	1024.0 MiB		2022/04/23 14:07:00	hadoop	FINISHED	17 s
app-20220423140613-0025	Secondary Sort	6	1024.0 MiB		2022/04/23 14:06:13	hadoop	FINISHED	21 s
app-20220423140545-0024	Secondary Sort	6	1024.0 MiB		2022/04/23 14:05:45	hadoop	FINISHED	17 s
app-20220423140516-0023	Secondary Sort	6	1024.0 MiB		2022/04/23 14:05:16	hadoop	FINISHED	18 s

1 GB-os teszt:

```
./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWCSecondarySort.jar /input/lorem1gb.txt /spark/wcss_lorem1gb_noopt_1

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWCSecondarySort.jar /input/lorem1gb.txt /spark/wcss_lorem1gb_noopt_2

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWCSecondarySort.jar /input/lorem1gb.txt /spark/wcss_lorem1gb_noopt_3

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWCSecondarySort.jar /input/lorem1gb.txt /spark/wcss_lorem1gb_noopt_4

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkWCSecondarySort.jar /input/lorem1gb.txt /spark/wcss_lorem1gb_noopt_5
```

Completed Applications (33)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20220423141402-0032	Secondary Sort	6	1024.0 MiB		2022/04/23 14:14:02	hadoop	FINISHED	23 s
app-20220423141328-0031	Secondary Sort	6	1024.0 MiB		2022/04/23 14:13:28	hadoop	FINISHED	21 s
app-20220423141252-0030	Secondary Sort	6	1024.0 MiB		2022/04/23 14:12:52	hadoop	FINISHED	23 s
app-20220423141203-0029	Secondary Sort	6	1024.0 MiB		2022/04/23 14:12:03	hadoop	FINISHED	23 s
app-20220423141130-0028	Secondary Sort	6	1024.0 MiB		2022/04/23 14:11:30	hadoop	FINISHED	23 s

A lefutott tesztek futás idejének az átlaga a következő táblázatban látható.

Spark WC és Secondary Sort optimalizálás nélkül			
Adat mérete	100MB	500MB	1GB
Átlagos futásidő	13,8 mp	18 mp	22,6 mp

6.2.6. Spark – Page Rank

A teszt elméleti háttere teljesen megegyezik a 6.2.3-as pontban olvashatóval.

Eclipseben írtam meg a Spark Page Rank implementációt Java nyelven.

JAR-ként exportálom a /usr/local/spark/userscripts mappába.

WordCount.java | WCSecondarySort.java | PageRank.java | SecondarySort.java | PageRank.java | WordCount.java

```java
import scala.Tuple2;
import com.google.common.collect.Iterables;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.storage.StorageLevel;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.*;
import java.util.*;
import java.util.regex.Pattern;

public class PageRank {
    private static final Pattern SZOKOZOK = Pattern.compile("\\t+");
    private static final double CSILLAPITAS = 0.85d;
    public static void main(String[] args) throws Exception {
        SparkConf sparkConf = new SparkConf().setAppName("Page Rank");
        JavaSparkContext spark = new JavaSparkContext(sparkConf);
        JavaRDD<String> sorok = spark.textFile(args[0], 1);
        final JavaRDD<String> adat = sorok.filter(new Function<String,Boolean>(){
            public Boolean call(String s) {
                return !s.startsWith("#");
            }
        });
        JavaPairRDD<String, Iterable<String>> linkek = adat.mapToPair(new PairFunction<String, String, String>() {
            @Override
            public Tuple2<String, String> call(String s) {
                String[] reszek = SZOKOZOK.split(s);
                return new Tuple2<String, String>(reszek[0], reszek[1]);
            }
        }).groupByKey().persist(StorageLevel.MEMORY_ONLY());
        JavaPairRDD<String, Double> rankok = linkek.mapValues(new Function<Iterable<String>, Double>() {
            @Override
            public Double call(Iterable<String> rs) {
                return 1.0;
            }
        });
        for (int i = 0; i < Integer.parseInt(args[2]); i++) {
            JavaPairRDD<String, Double> contribs = linkek.join(rankok).values().flatMapToPair(new PairFlatMapFunction<Tuple2<Iterable<String>, Double>,String, Double>() {
                @Override
                public Iterator<Tuple2<String, Double>> call(Tuple2<Iterable<String>, Double> s) {
                    int urlDarab = Iterables.size(s._1);
                    List<Tuple2<String, Double>> eremenyek = new ArrayList<Tuple2<String, Double>>();
                    for (String n : s._1) {
                        eremenyek.add(new Tuple2<String, Double>(n, s._2() / urlDarab));
                    }
                    return eremenyek.iterator();
                }
            });

            rankok = contribs.reduceByKey(new Osszeg()).mapValues(new Function<Double, Double>() {
                @Override
                public Double call(Double osszeg) {
                    return (1.0 - CSILLAPITAS) + osszeg * CSILLAPITAS;
                }
            });
        }
        rankok.sortByKey().repartition(1).map(x -> x._1 + "\t" + x._2).saveAsTextFile(args[1]);
        spark.stop();
    }
    private static class Osszeg implements Function2<Double, Double, Double> {
        @Override
        public Double call(Double a, Double b) { return a + b; }
    }
}
```

Skitter.txt teszt:

```
./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkPageRank.jar /input/Skitter.txt /spark/pr_skitter_3_noopt_1 3

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkPageRank.jar /input/Skitter.txt /spark/pr_skitter_3_noopt_2 3
```

```
./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkPageRank.jar /input/Skitter.txt /spark/pr_skitter_3_noopt_3 3

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkPageRank.jar /input/Skitter.txt /spark/pr_skitter_3_noopt_4 3
```

▾ Completed Applications (37)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20220423150440-0036	Page Rank	6	1024.0 MiB		2022/04/23 15:04:40	hadoop	FINISHED	4,5 min
app-20220423150003-0035	Page Rank	6	1024.0 MiB		2022/04/23 15:00:03	hadoop	FINISHED	4,2 min
app-20220423145045-0034	Page Rank	6	1024.0 MiB		2022/04/23 14:50:45	hadoop	FINISHED	5,2 min
app-20220423144459-0033	Page Rank	6	1024.0 MiB		2022/04/23 14:44:59	hadoop	FINISHED	4,7 min

BerkStan.txt teszt:

```
./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkPageRank.jar /input/BerkStan.txt /spark/pr_berkstan_3_noopt_1 3

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkPageRank.jar /input/BerkStan.txt /spark/pr_berkstan_3_noopt_2 3

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkPageRank.jar /input/BerkStan.txt /spark/pr_berkstan_3_noopt_3 3

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkPageRank.jar /input/BerkStan.txt /spark/pr_berkstan_3_noopt_4 3
```

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20220423152657-0041	Page Rank	6	1024.0 MiB		2022/04/23 15:26:57	hadoop	FINISHED	2,9 min
app-20220423152341-0040	Page Rank	6	1024.0 MiB		2022/04/23 15:23:41	hadoop	FINISHED	3,0 min
app-20220423151818-0038	Page Rank	6	1024.0 MiB		2022/04/23 15:18:18	hadoop	FINISHED	3,0 min
app-20220423151328-0037	Page Rank	6	1024.0 MiB		2022/04/23 15:13:28	hadoop	FINISHED	2,9 min

Stanford.txt teszt:

```
./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkPageRank.jar /input/Stanford.txt /spark/page_stanford_3_noopt_1 3

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkPageRank.jar /input/Stanford.txt /spark/page_stanford_3_noopt_2 3

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkPageRank.jar /input/Stanford.txt /spark/page_stanford_3_noopt_3 3

./bin/spark-submit --master spark://192.168.56.2:7077 /usr/local/spark/userscripts/SparkPageRank.jar /input/Stanford.txt /spark/page_stanford_3_noopt_4 3
```

▾ Completed Applications (6)

Application ID	Name	Cores	Memory per Executor ▴	Resources Per Executor	Submitted Time	User	State	Duration
app-20220423162320-0005	Page Rank	6	1024.0 MiB		2022/04/23 16:23:20	hadoop	FINISHED	53 s
app-20220423162212-0004	Page Rank	6	1024.0 MiB		2022/04/23 16:22:12	hadoop	FINISHED	53 s
app-20220423162100-0003	Page Rank	6	1024.0 MiB		2022/04/23 16:21:00	hadoop	FINISHED	55 s
app-20220423161934-0002	Page Rank	6	1024.0 MiB		2022/04/23 16:19:34	hadoop	FINISHED	55 s

A lefutott tesztek futás idejének az átlaga a következő táblázatban látható.

Spark WC és Secondary Sort optimalizálás nélkül			
Adathalmaz neve	skitter.txt	berkstan.txt	stanford.txt
Átlagos futásidő	4 perc 39 mp	2 perc 57 mp	54 mp

6.3. Optimalizáció

A Hadoop és a Spark is nagy szabadságot biztosít a felhasználóknak az optimalizáció területén. Beállítások százait szabhatjuk a rendszerünk fizikai tulajdonságaihoz és rajta futó feladathoz.

Az optimalizációkat majd a Word Count teszten fogom letesztelni. Az alapértelmezett beállításokkal való futásidőket az alábbi táblázatban láthatjuk.

Hadoop Word Count optimalizálás nélkül			
Adat mérete	100MB	500MB	1GB
Átlagos futásidő	33 mp	1 perc 1 mp	3 perc 22 mp
Spark Word Count optimalizálás nélkül			
Adat mérete	100MB	500MB	1GB
Átlagos futásidő	12,2 mp	17 mp	22,4 mp

6.3.1. Hadoop

Az első módszer, amivel próbálkozhatunk az a kimenet tömörítése. A Hadoop alapból háromféle tömörítést támogat, ezek a következők: LZO, gzip és bzip2. Ezek közül mindegyiknek meg vannak az előnyei és hátrányai. Más-más a tömörítési arányuk és az idő, ami alatt a tömörítés és a kicsomagolás vesz igénybe.

Én a gzipet választom, mert jó a tömörítési és kicsomagolási ideje és mellé 25% körüli tömörítési aránya van. A map kimenetét fogom tömöríteni, így a reduce már tömörített adatokkal fog dolgozni. Ennek eléréséhez ki kell bővítenem a Hadoop Word Countjának a kódját:

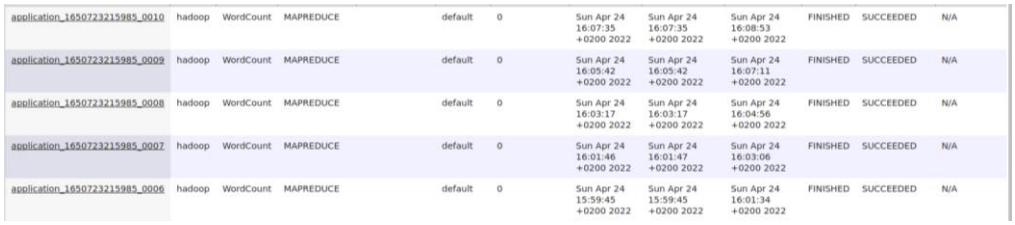
WordCount.java

```
import org.apache.hadoop.io.compress.*;

public class WordCount {
    public static class Map extends Mapper<Object, Text, Text, IntWritable> {
        private Text szo = new Text();

        public void map(Object kulcs, Text ertek, Context context) throws IOException, InterruptedException {
            StringTokenizer st = new StringTokenizer(ertek.toString());
            while (st.hasMoreTokens()){
                szo.set(st.nextToken());
                context.write(szo, new IntWritable(1));
            }
        }
    }
    public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable>{
        private IntWritable eredmeny = new IntWritable();
        public void reduce(Text kulcs, Iterable<IntWritable> ertekek, Context context) throws IOException, InterruptedException{
            int osszeg = 0;
            for (IntWritable ertek : ertekek) {
                osszeg += ertek.get();
            }
            eredmeny.set(osszeg);
            context.write(kulcs, eredmeny);
        }
    }
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        conf.setBoolean(Job.MAP_OUTPUT_COMPRESS,true);
        conf.setClass(Job.MAP_OUTPUT_COMPRESS_CODEC,GzipCodec.class,CompressionCodec.class);
        Job job = Job.getInstance(conf, "WordCount");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(Map.class);
        job.setCombinerClass(Reduce.class);
        job.setReducerClass(Reduce.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job,new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

A teszteket 6.2.1-es pontban írt módon fogom lefuttatni.

100MB-os teszt eredményei tömörítéssel:

Az 500MB-os teszt eredményei tömörítéssel:



Az 1GB-os teszt eredményei tömörítéssel:

Hadoop Word Count tömörítéssel			
Adat mérete	100MB	500MB	1GB
Átlagos futásidő	38 mp	1 perc 31mp	4 perc 25 mp

Mivel a futásidő nagy részét a map fázis teszi ki, ezért a tömörítés rontott a futásidőn, mivel több időt vesz el a tömörítés, mint amit nyerünk vele a reduce fázisban. Ezért ezt az optimalizációt nem visszük tovább.

A következő optimalizációs kísérletben a YARN beállításait fogom megváltoztatni. Amit megfigyelhettünk, hogy a map fázis tart tovább, de amint egyre nagyobb fájlt dolgozunk fel a reduce fázis is egyre hosszabb arányaiban.

A Hadoop úgy működik, hogy a map eredményeit egy bizonyos méretig a memóriában tárolja és amint egy limitet elér kiírja a HDFS-re. Minden merevlemezre való írás időköltséges, ezért a YARN beállításaiban megnöveljük a buffer méretét, a párhuzamos másolatok számát és a limitet, amit után a bufferből kiírja az adatokat a HDFS-re. Ezeket a mapred-site.xml-ben kell beállítani.

```
sudo nano /usr/local/hadoop/etc/hadoop/mapred-site.xml
```

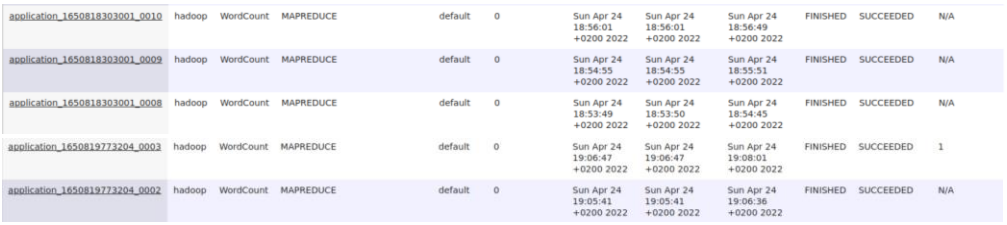
```
<property>
        <name>mapreduce.reduce.shuffle.parallelcopies</name>
        <value>6</value>
</property>
<property>
        <name>mapreduce.task.io.sort.mb</name>
        <value>150</value>
</property>
<property>
        <name>mapreduce.map.sort.spill.percent </name>
        <value>0.85</value>
</property>
```

A teszteket a 6.2.1-ben leírtak alapján futtatom.

100MB-os eredményei mapred-site.xml változtatása után:

application_1650723215985_0021	hadoop	WordCount	MAPREDUCE	default	0	Sun Apr 24 17:20:18 +0200 2022	Sun Apr 24 17:20:19 +0200 2022	Sun Apr 24 17:20:51 +0200 2022	FINISHED	SUCCEEDED	N/A
application_1650723215985_0020	hadoop	WordCount	MAPREDUCE	default	0	Sun Apr 24 17:17:25 +0200 2022	Sun Apr 24 17:17:26 +0200 2022	Sun Apr 24 17:17:59 +0200 2022	FINISHED	SUCCEEDED	N/A
application_1650723215985_0019	hadoop	WordCount	MAPREDUCE	default	0	Sun Apr 24 17:13:48 +0200 2022	Sun Apr 24 17:13:48 +0200 2022	Sun Apr 24 17:14:20 +0200 2022	FINISHED	SUCCEEDED	N/A
application_1650723215985_0018	hadoop	WordCount	MAPREDUCE	default	0	Sun Apr 24 17:11:05 +0200 2022	Sun Apr 24 17:11:06 +0200 2022	Sun Apr 24 17:11:39 +0200 2022	FINISHED	SUCCEEDED	N/A
application_1650723215985_0017	hadoop	WordCount	MAPREDUCE	default	0	Sun Apr 24 17:09:10 +0200 2022	Sun Apr 24 17:09:11 +0200 2022	Sun Apr 24 17:09:43 +0200 2022	FINISHED	SUCCEEDED	N/A

500MB-os eredményei mapred-site.xml változtatása után:



application_1650818303001_0010	hadoop	WordCount	MAPREDUCE	default	0	Sun Apr 24 18:56:01 +0200 2022	Sun Apr 24 18:56:01 +0200 2022	Sun Apr 24 18:56:49 +0200 2022	FINISHED	SUCCEEDED	N/A
application_1650818303001_0009	hadoop	WordCount	MAPREDUCE	default	0	Sun Apr 24 18:54:55 +0200 2022	Sun Apr 24 18:54:55 +0200 2022	Sun Apr 24 18:55:51 +0200 2022	FINISHED	SUCCEEDED	N/A
application_1650818303001_0008	hadoop	WordCount	MAPREDUCE	default	0	Sun Apr 24 18:53:49 +0200 2022	Sun Apr 24 18:53:50 +0200 2022	Sun Apr 24 18:54:45 +0200 2022	FINISHED	SUCCEEDED	N/A
application_1650819773204_0003	hadoop	WordCount	MAPREDUCE	default	0	Sun Apr 24 19:06:47 +0200 2022	Sun Apr 24 19:06:47 +0200 2022	Sun Apr 24 19:08:01 +0200 2022	FINISHED	SUCCEEDED	1
application_1650819773204_0002	hadoop	WordCount	MAPREDUCE	default	0	Sun Apr 24 19:05:41 +0200 2022	Sun Apr 24 19:05:41 +0200 2022	Sun Apr 24 19:06:36 +0200 2022	FINISHED	SUCCEEDED	N/A

1GB-os eredményei mapred-site.xml változtatása után:

application_1650821687344_0006	hadoop	WordCount	MAPREDUCE	default	0	Sun Apr 24 19:50:49 +0200 2022	Sun Apr 24 19:50:50 +0200 2022	Sun Apr 24 19:54:41 +0200 2022	FINISHED	SUCCEEDED	1
application_1650821687344_0005	hadoop	WordCount	MAPREDUCE	default	0	Sun Apr 24 19:47:27 +0200 2022	Sun Apr 24 19:47:27 +0200 2022	Sun Apr 24 19:50:23 +0200 2022	FINISHED	SUCCEEDED	N/A
application_1650821687344_0004	hadoop	WordCount	MAPREDUCE	default	0	Sun Apr 24 19:44:03 +0200 2022	Sun Apr 24 19:44:04 +0200 2022	Sun Apr 24 19:47:10 +0200 2022	FINISHED	SUCCEEDED	N/A
application_1650821687344_0003	hadoop	WordCount	MAPREDUCE	default	0	Sun Apr 24 19:40:38 +0200 2022	Sun Apr 24 19:40:39 +0200 2022	Sun Apr 24 19:43:42 +0200 2022	FINISHED	SUCCEEDED	N/A
application_1650821687344_0002	hadoop	WordCount	MAPREDUCE	default	0	Sun Apr 24 19:37:05 +0200 2022	Sun Apr 24 19:37:07 +0200 2022	Sun Apr 24 19:39:51 +0200 2022	FINISHED	SUCCEEDED	N/A

Hadoop Word Count mapred-site.xml változtatásával			
Adat mérete	100MB	500MB	1GB
Átlagos futásidő	32 mp	58 mp	3 perc 8 mp

A YARN optimalizálása után pár másodperces javulásokat értünk el, ami viszonylag jók a limitált erőforrások mellett.

A harmadik optimalizációs lehetőség a reduce feladatok számának megváltoztatása. Ha növeljük a reducerek számát, akkor a terhelés jobban eloszlik és az esetleges hibák is kevésbé jelentenek problémát. Ehhez kibővítem a Word Count függvényt, harmadik pa-

raméterként meg kell adjuk a reducerek számát. Emellett még megpróbáljuk optimalizálni a map és reduce fázisokhoz rendelt memóriát. A slave gépeken a futó programok mellett kb. 1 GB szabad RAM található ezt a map és a reducerek között 300-600MB arányban osztom el.

```
<property>
        <name>mapreduce.map.memory.mb </name>
        <value>300</value>
</property>
<property>
        <name>mapreduce.reduce.memory.mb </name>
        <value>600</value>
</property>
```

WordCount.java

```
import java.io.IOException;

public class WordCount {
    public static class Map extends Mapper<Object, Text, Text, IntWritable> {
        private Text szo = new Text();

        public void map(Object kulcs, Text ertek, Context context) throws IOException, InterruptedException {
            StringTokenizer st = new StringTokenizer(ertek.toString());
            while (st.hasMoreTokens()){
                szo.set(st.nextToken());
                context.write(szo, new IntWritable(1));
            }
        }
    }
    public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable>{
        private IntWritable eredmeny = new IntWritable();
        public void reduce(Text kulcs, Iterable<IntWritable> ertekek, Context context) throws IOException, InterruptedException{
            int osszeg = 0;
            for (IntWritable ertek : ertekek) {
                osszeg += ertek.get();
            }
            eredmeny.set(osszeg);
            context.write(kulcs, eredmeny);
        }
    }
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "WordCount");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(Map.class);
        job.setCombinerClass(Reduce.class);
        job.setReducerClass(Reduce.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setNumReduceTasks(Integer.parseInt(args[2]));
        FileInputFormat.addInputPath(job,new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
```

A teszteket a 6.2.1-es ponthoz hasonlóan futtatom, azzal a különbséggel, hogy már meg kell adjam a reducerek számát. A 100MB és az 500MB-os fájlokon 1 reducerrel kapjuk a legjobb eredményt, bár ez nem tér el sokkal az optimalizáció nélkülitől, viszont az 1GB-os fájlnál már 3 reducer eredményezi a legjobb teljesítményt és több mint 1 perces javulást idéz elő.

Hadoop Word Count reducerek beállításával			
Adat mérete	100MB	500MB	1GB
Átlagos futásidő	33 mp	57 mp	1 perc 49 mp
Reducerek száma	1	1	3

6.3.2. Spark

A Sparknál csak két optimalizációt fogunk csak elvégezni, mert már alapértelmezetten Snappy tömörítést használ, ezért a tömörítés általi optimalizációt nem kell megírnunk.

Az első optimalizációban a Spark növelem a driver magok számát, továbbá a gép szabad memóriáját (körülbelül 1 GB) a driver és executor között kézzel allokálom.

A Sparkban az ilyen beállításokat úgy tudjuk megtenni, hogy a spark-submit parancs után --conf után megadom a beállításokat, amikkel az elindított program le fog futni.

A tesztek a 6.2.4-ben megírtakhoz fog hasonlítani, azzal a különbséggel, hogy három --conf beállítás bele fog még kerülni. Például:

```
./bin/spark-submit
--master spark://192.168.56.2:7077
--conf "spark.driver.cores = 2"
--conf "spark.driver.memory = 400m"
--conf "spark.executor.memory = 600m"
/usr/local/spark/userscripts/SparkWordCount.jar
/input/lorem1gb.txt /[teszt_neve_teszt_szam]
```

Ez a kód elindítja a SparkWordCount.jar-t a Spark clusterünkön, a driver coreok számát 2-re állítja, a driver memóriáját 400 MB-ra és az executor memóriáját 600 MB-ra. Személyes tapasztalat alapján egy 2:3-hoz arányú eloszlás eredményezi a legjobb futásidőt.

100MB-os teszt eredményei az első optimalizációval:

app-20220425155510-0029	Word Count	6	1024.0 MiB		2022/04/25 15:55:10	hadoop	FINISHED	12 s
app-20220425155448-0028	Word Count	6	1024.0 MiB		2022/04/25 15:54:48	hadoop	FINISHED	12 s
app-20220425155427-0027	Word Count	6	1024.0 MiB		2022/04/25 15:54:27	hadoop	FINISHED	12 s
app-20220425155403-0026	Word Count	6	1024.0 MiB		2022/04/25 15:54:03	hadoop	FINISHED	11 s
app-20220425155301-0025	Word Count	6	1024.0 MiB		2022/04/25 15:53:01	hadoop	FINISHED	11 s

500MB-os teszt eredményei az első optimalizációval:

app-20220425155047-0024	Word Count	6	1024.0 MiB		2022/04/25 15:50:47	hadoop	FINISHED	17 s
app-20220425155022-0023	Word Count	6	1024.0 MiB		2022/04/25 15:50:22	hadoop	FINISHED	17 s
app-20220425154955-0022	Word Count	6	1024.0 MiB		2022/04/25 15:49:55	hadoop	FINISHED	17 s
app-20220425154925-0021	Word Count	6	1024.0 MiB		2022/04/25 15:49:25	hadoop	FINISHED	16 s
app-20220425154856-0020	Word Count	6	1024.0 MiB		2022/04/25 15:48:56	hadoop	FINISHED	17 s

1GB-os teszt eredményei az első optimalizációval:

app-20220425153557-0009	Word Count	6	1024.0 MiB		2022/04/25 15:35:57	hadoop	FINISHED	24 s
app-20220425153524-0008	Word Count	6	1024.0 MiB		2022/04/25 15:35:24	hadoop	FINISHED	22 s
app-20220425153452-0007	Word Count	6	1024.0 MiB		2022/04/25 15:34:52	hadoop	FINISHED	21 s
app-20220425153413-0006	Word Count	6	1024.0 MiB		2022/04/25 15:34:13	hadoop	FINISHED	21 s
app-20220425153342-0005	Word Count	6	1024.0 MiB		2022/04/25 15:33:42	hadoop	FINISHED	22 s

Spark Word Count az első optimalizáció után			
Adat mérete	100MB	500MB	1GB
Átlagos futásidő	11,6 mp	16,8 mp	22 mp

Az első optimalizáció minimálisan javította az átlagos futásidőt minden tesztnél.

A második optimalizáció az elsőre fog épülni. Az előzőben beállítottuk a driver magok számát és a memória allokációt a driver és executor között. Úgy tudnánk még gyorsítani a folyamaton, ha a RDD-t annyi részre osztanánk ahány driver mag van. Ehhez a Spark WordCount kódot kiegészítem egy harmadik argumentummal (az első kettő a bemeneti fájl és a kimeneti mappa) amivel a partíciók számát adjuk meg. Fontos megjegyezni, hogy az újra partícionálás elég erőforrás igényes művelet, ezért nem minden esetben várunk el javulást a futásidőn.

WordCount.java | WordCountOpt2.java

```
import scala.Tuple2;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.function.PairFunction;
import java.util.Arrays;
import java.util.Iterator;

public class WordCountOpt2 {
    public static void main(String[] args) throws Exception {
        SparkConf sparkConf = new SparkConf().setAppName("Word Count");
        JavaSparkContext spark = new JavaSparkContext(sparkConf);
        JavaRDD<String> szoveg = spark.textFile(args[0]);
        JavaRDD<String> szavak = szoveg.flatMap(new FlatMapFunction<String, String>() {
            public Iterator<String> call(String s) {
                return Arrays.asList(s.split(" ")).iterator();
            }
        });
        JavaPairRDD<String, Integer> szavakMap = szavak.mapToPair(new PairFunction<String, String, Integer>() {
            public Tuple2<String, Integer> call(String s) {
                return new Tuple2<String, Integer>(s, 1);
            }
        });
        JavaPairRDD<String, Integer> freqPair = szavakMap.reduceByKey(new Function2<Integer, Integer, Integer>() {
            public Integer call(Integer a, Integer b) {
                return a + b;
            }
        });
        freqPair.repartition(Integer.parseInt(args[2])).sortByKey().map(x -> x._1 + "\t" + x._2).saveAsTextFile(args[1]);
        spark.stop();
    }
}
```

100 MB-os teszt a második optimalizációval:

app-20220425160844-0036	Word Count	6	1024.0 MiB		2022/04/25 16:08:44	hadoop	FINISHED	12 s
app-20220425160822-0035	Word Count	6	1024.0 MiB		2022/04/25 16:08:22	hadoop	FINISHED	12 s
app-20220425160757-0034	Word Count	6	1024.0 MiB		2022/04/25 16:07:57	hadoop	FINISHED	12 s
app-20220425160734-0033	Word Count	6	1024.0 MiB		2022/04/25 16:07:34	hadoop	FINISHED	13 s
app-20220425160710-0032	Word Count	6	1024.0 MiB		2022/04/25 16:07:10	hadoop	FINISHED	12 s

500 MB-os teszt a második optimalizációval:

app-20220425161117-0041	Word Count	6	1024.0 MiB		2022/04/25 16:11:17	hadoop	FINISHED	17 s
app-20220425161050-0040	Word Count	6	1024.0 MiB		2022/04/25 16:10:50	hadoop	FINISHED	16 s
app-20220425161023-0039	Word Count	6	1024.0 MiB		2022/04/25 16:10:23	hadoop	FINISHED	18 s
app-20220425160956-0038	Word Count	6	1024.0 MiB		2022/04/25 16:09:56	hadoop	FINISHED	17 s
app-20220425160929-0037	Word Count	6	1024.0 MiB		2022/04/25 16:09:29	hadoop	FINISHED	17 s

1 GB-os teszt a második optimalizációval:

app-20220425161445-0046	Word Count	6	1024.0 MiB		2022/04/25 16:14:45	hadoop	FINISHED	24 s
app-20220425161413-0045	Word Count	6	1024.0 MiB		2022/04/25 16:14:13	hadoop	FINISHED	21 s
app-20220425161342-0044	Word Count	6	1024.0 MiB		2022/04/25 16:13:42	hadoop	FINISHED	21 s
app-20220425161311-0043	Word Count	6	1024.0 MiB		2022/04/25 16:13:11	hadoop	FINISHED	21 s
app-20220425161236-0042	Word Count	6	1024.0 MiB		2022/04/25 16:12:36	hadoop	FINISHED	21 s

Spark Word Count az első optimalizáció után			
Adat mérete	100MB	500MB	1GB
Átlagos futásidő	12,2 mp	17 mp	21,6 mp

Az első két tesztnél rosszabb futásidőt értünk el, de az 1 GB-os tesztnél már javult a futásidő, mert a reparticionálás időveszteségе megtérült a gyorsabb feldolgozás miatt.

7. KONKLÚZIÓ

A konklúziók levonásához először is összehasonlítom az egyes tesztek és optimalizációk átlagos futásidejét és számolok egy arányt, ahol a Hadoop futásidejét elosztom a Spark futásidejével, így kapok egy arányt, amely mérőszámként szolgál a teljesítmények összehasonítására.

7.1. Word Count

Az első teszt a szavak megszámlálása volt és itt a Spark a 100 MB-os teszten 2,7-szer gyorsabb volt a Hadoopnál és az adathalmaz növekedésével ez a szám felment több mint 9-szeresre.

Word Count optimalizálás nélkül			
Adat mérete	100MB	500MB	1GB
Hadoop átlagos futásidő	33 mp	1 perc 1 mp	3 perc 22 mp
Spark átlagos futásidő	12,2 mp	17 mp	22,4 mp
Teljesítmény arány	2,70	3,59	9,02

7.2. Word Count - optimalizált

Miután ezt a tesztet mind a két programon optimalizáltam, utána a legjobb eredményeiket hasonlítottam össze. A 100 MB-os adathalmazon ugyan az maradt az eredmény, az 500 MB-os adathalmazon a Hadoop arányaiban javult 7%-ot és az 1 GB-os teszten 45%-ot. A Spark így is 5-ször gyorsabban dolgozta fel az 1 GB-os adathalmazt.

Word Count optimalizálással			
Adat mérete	100MB	500MB	1GB
Hadoop átlagos futásidő	33 mp	57 mp	1 perc 49 mp
Spark átlagos futásidő	12,2 mp	17 mp	21,6 mp
Teljesítmény arány	2,70	3,35	5,05

7.3. Word Count és többszöri rendezés

A Word Count és Secondary Sort teszt 3 munkafolyamatból állt, amiből az első a Word Count volt és utána megcseréltük a kulcs érték párokat, csoportosítottuk és rendeztük a csoportokon belül. Ez az algoritmus sokkal több iterációt tartalmazott az alapjaként szolgáló Word Countnál.

Word Count és Secondary Sort			
Adat mérete	100MB	500MB	1GB
Hadoop átlagos futásidő	1 perc 9 mp	2 perc 17 mp	4 perc 3 mp
Spark átlagos futásidő	13,8 mp	18 mp	22,6 mp
Teljesítmény arány	5	7,61	10,75

Az eredményeken láthatjuk, hogy a Hadoop a sok iterációt tartalmazó algoritmusokban még lassabb a Sparknál, abból adódóan, hogy a Spark a memóriában tárolja el a részeredményeket, miközben a Hadoop minden részeredményt kiír a HDFS-re.

7.4. Page Rank

Az utolsó teszt a Page Rank volt, amit 3 iteráción keresztül futtattunk és egy nagyon számolásigényes.

Page Rank			
Adat mérete	Skitter	BerkStan	Stanford
Hadoop átlagos futásidő	7 perc 26 mp	3 perc 45 mp	1 perc 58 mp
Spark átlagos futásidő	4 perc 39 mp	2 perc 57 mp	54 mp
Teljesítmény arány	1,6	1,27	2,19

Ezen a teszten is a Spark gyorsabb volt a Hadoopnál.

7.5. Összefoglalás

Az általam futtatott tesztek és mérések alapján kijelenthetem, hogy a Spark a Hadoopnál minden esetben sokkal jobban teljesített. Főleg a második tesztnél, ahol a sok iteráció miatt több mint tízszeres eltérést mérhettünk az 1 GB-os tesztnél. Ez abból adódik, hogy a Spark a RAM-ban tárolja az adatokat, ahonnan, ha később újra el szeretné érni őket akkor gyorsabban teheti meg, így gyorsabb futásidőt kapunk, ellentétben a Hadoop inkább a nagy mennyiségű adatátvitelre van fejlesztve és a merevlemezre ír. Ebből adódóan a Hadoopnál I/O műveletek teszik ki a futási idő nagyrészét.

Az optimalizációk sajnos limitáltak voltak a szakdolgozatomban, mivel a gépeknek erősen limitált volt az erőforrása. A 2 mag és 2 GB RAM amit minden virtuális gép adott nem hagyott sok lehetőséget az optimalizációval való játszadozásra, mert voltak olyan beállítások, amiket nem lehetett az alapértelmezettnél magasabbra állítani.

A Spark több optimalizációs lehetőséget biztosít a Hadoopnál, ezért elméletben jobban optimalizálható kéne legyen, bár ez nem látszik az arányszámok alapján az optimalizációnál, mivel alapból annyira gyorsabb, hogy olyan mértékű javulást nehéz lenne elérni.

A végső konklúzióm az, hogy a Spark általánosságban jobb, ha csak az adatfeldolgozási sebesség számít, de sokkal nagyobb a memóriaigénye. Azt is szem előtt kell tartani, hogy a Spark nem a Hadoop leváltására jött létre, hanem kiegészítésére, mivel az RDD-k működéséből adódóan nem jó választás, ha aszinkron módon kell a tárhelybe kis frissítéseket írni. Ezek mellett, ha az idő nem fontos tényező és a memória egy szűkös erőforrás, akkor a Hadoopot javasolnám.

8. HIVATKOZÁSOK

[21] Akil, B., Zhou, Y., Rohm, U.: On the Usability of Hadoop MapReduce, Apache Spark & Apache Flink for Data Science, *IEEE International Conference on Big Data*, 2017

[22] Apache: Cluster Mode Overview, https://spark.apache.org/docs/latest/cluster-overview.html, 2021. 11. 10.

[29] Bianchini, M., Gori, M., Scarselli, F.: Inside PageRank *ACM Trans. Inter. Tech. TOIT ACM Transactions on Internet Technology*, 2005, oldal. 92-128

[14] Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. (https://www.usenix.org/legacy/event/osdi04/tech/full_papers/dean/dean_html/), utoljára megtekintve: 2021. 12. 01.

[16] Hazarika, A., Jain, E., Ram, G.: Performance comparision of Hadoop and spark engine, *International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, 2017

[30] http://snap.stanford.edu/data/, utoljára látogatva 2022. 04. 01.

[18] Ikken, S., Renault, É., Kechadi, T., Tari, A.: Toward Scheduling I/O Request of Mapreduce Tasks Based on Markov Model, *International Conference on Mobile, Secure and Programmable Networking*, 2015

[17] Rattanaopas, K., Kaewkeeree, S.: Improving Hadoop MapReduce performance with data compression: A study using wordcount job, *International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, 2017

[13] Singh, A., Khamparia, A., & Luhach, A. K.: Performance comparison of Apache Hadoop and Apache Spark, *Proceedings of the Third International Conference on Advanced Informatics for Computing Research*, 2019

[15] Son, S., Gil, M., Moon, Y.: Anomaly Detection for Big Log Data Using a Hadoop Ecosystem, *2017 IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2017

[20] Wijayanto, A., Winarko, E.: Implementation of Multi-criteria Collaborative Filtering on Cluster Using Apache Spark, *2nd International Conference on Science and Technology-Computer (ICST)*, 2016

[19] Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets, *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010